CDlib Documentation

Release 0.1.9

Giulio Rossetti

Sep 22, 2020

Contents

`CDlib` is a Python software package that allows to extract, compare and evaluate communities from complex networks.

The library provides a standardized input/output for several existing Community Discovery algorithms. The implementations of all CD algorithms are inherited from existing projects, each one of them acknowledged in the dedicated method reference page.

Date	Python Versions	Main Author	GitHub	pypl
2020-09-22	3.7-3.8	Giulio Rossetti	Source	Distribution

CHAPTER 1

CDlib Dev Team

Name	Contribution
Giulio Rossetti	Library Design/Documentation
Letizia Milli	Community Models Integration
Rémy Cazabet	Visualization
Salvatore Citraro	Community Models Integration

1.1 Overview

`CDlib` is a Python language software package for the extraction, comparison, and evaluation of communities from complex networks.

1.1.1 Who uses CDlib?

The potential audience for `CDlib` includes mathematicians, physicists, biologists, computer scientists, and social scientists.

1.1.2 Goals

`CDlib` is built upon the NetworkX python library and is intended to provide:

- a standard programming interface and community discovery implementations that are suitable for many applications,
- a rapid development environment for collaborative, multidisciplinary, projects.

1.1.3 The Python CDlib library

`CDlib` is a powerful Python package that allows simple and flexible partitioning of complex networks.

Most importantly, `CDlib`, as well as the Python programming language, is free, well-supported, and a joy to use.

1.1.4 Free software

`CDlib` is free software; you can redistribute it and/or modify it under the terms of the BSD License. We welcome contributions from the community.

1.1.5 EU H2020

`CDlib` is a result of an European H2020 project:

- SoBigData "Social Mining & Big Data Ecosystem": under the scheme "INFRAIA-1-2014-2015: Research Infrastructures", grant agreement #654024.

1.2 Download

1.2.1 Software

Source and binary releases: https://pypi.python.org/pypi/cdlib

Github (latest development): https://github.com/GiulioRossetti/cdlib

1.2.2 Documentation

1.3 Installing CDlib

Before installing `CDlib`, you need to have setuptools installed.

1.3.1 Quick install

Get `CDlib` from the Python Package Index at pypl.

or install it with

```
pip install cdlib
```

and an attempt will be made to find and install an appropriate version that matches your operating system and Python version.

You can install the development version with

```
pip install git://github.com/GiulioRossetti/cdlib.git
```

1.3.2 Optional Dependencies

`CDlib` relies on a few packages calling C code (namely: `python-igraph`, `leidenalg`, `angel_cd` and `infomap`). The default installation will not set up such requirements since their configuration under non unix-like systems is not trivial and cannot be easily automated.

Such a choice has been made to allow (even) Windows user to install the library and get access to its core functionalities.

To made available (most of) the optional packages you can either:

- (Windows) manually install the optional packages (versions details are specified in `requirements_optional.txt`) following the original projects guidelines, or
- (Linux/OSX) run the command:

```
pip install cdlib[C]
```

Such caveat will install everything that can be easily automated under Linux/OSX.

(Advanced) Graph-tool

The only optional dependency that will remain unsatisfied following the previous procedures will be **graph-tool** (used to add SBM models). If you need it up and running, refer to the official documentation.

1.3.3 Installing from source

You can install from source by downloading a source archive file (tar.gz or zip) or by checking out the source files from the GitHub source code repository.

`CDlib` is a pure Python package; you don't need a compiler to build or install it.

Source archive file

Download the source (tar.gz or zip file) from pypl or get the latest development version from GitHub

Unpack and change directory to the source directory (it should have the files README.txt and setup.py).

Run python setup.py install to build and install

GitHub

Clone the CDlib repostitory (see GitHub for options)

```
git clone https://github.com/GiulioRossetti/cdlib.git
```

Change directory to CDlib

Run python setup.py install to build and install

If you don't have permission to install software on your system, you can install into another directory using the –user, –prefix, or –home flags to setup.py.

For example

```
python setup.py install --prefix=/home/username/python
```

or

```
python setup.py install --home=~
```

or

```
python setup.py install --user
```

If you didn't install in the standard Python site-packages directory you will need to set your PYTHONPATH variable to the alternate location. See http://docs.python.org/2/install/index.html#search-path for further details.

1.3.4 Requirements

Python

To use CDlib you need Python 3.6 or later.

The easiest way to get Python and most optional packages is to install the Enthought Python distribution "Canopy" or using Anaconda.

There are several other distributions that contain the key packages you need for scientific computing.

1.4 Tutorial

NClib is built upon networkx and is designed to extract, compare and evaluate network partitions.

You can find a few basilar examples to get started with `cdlib` in this notebook

1.5 Reference

`CDlib` composes of several modules, each one fulfilling a different task related to community detection.

1.5.1 Community Objects

`cdlib` provides data structures and methods for storing communities.

The choice of community class depends on the structure of the community generated by the selected algorithm.

Which community should I use?

Community Type	cdlib class
Node Partition	NodeClustering, FuzzyNodeClustering, AttrNodeClustering, BiNodeClustering
Edge Partition	EdgeClustering

Community Types

Node Clustering

Overview

class NodeClustering(*communities*, *graph*, *method_name*, *method_parameters=None*, *overlap=False*)

Node Communities representation.

Parameters

- **communities** – list of communities
- **graph** – a networkx/igraph object
- **method_name** – community discovery algorithm name
- **method_parameters** – configuration for the community discovery algorithm used
- **overlap** – boolean, whether the partition is overlapping or not

adjusted_mutual_information(*clustering*)

Adjusted Mutual Information between two clusterings.

Adjusted Mutual Information (AMI) is an adjustment of the Mutual Information (MI) score to account for chance. It accounts for the fact that the MI is generally higher for two clusterings with a larger number of clusters, regardless of whether there is actually more information shared. For two clusterings U and V, the AMI is given as:

```
AMI(U, V) = [MI(U, V) - E(MI(U, V))] / [max(H(U), H(V)) - E(MI(U, V))]
```

This metric is independent of the absolute values of the labels: a permutation of the class or cluster label values won't change the score value in any way.

This metric is furthermore symmetric: switching `label_true` with `label_pred` will return the same score value. This can be useful to measure the agreement of two independent label assignments strategies on the same dataset when the real ground truth is not known.

Be mindful that this function is an order of magnitude slower than other metrics, such as the Adjusted Rand Index.

Parameters **clustering** – NodeClustering object

Returns AMI score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> leiden_communities = algorithms.leiden(g)
>>> mod = communities.adjusted_mutual_information(leiden_communities)
```

Reference

1. Vinh, N. X., Epps, J., & Bailey, J. (2010). **Information theoretic measures for clusterings comparison: Variants, properties, normalization and correction for chance.** Journal of Machine Learning Research, 11(Oct), 2837-2854.

adjusted_rand_index(*clustering*)

Rand index adjusted for chance.

The Rand Index computes a similarity measure between two clusterings by considering all pairs of samples and counting pairs that are assigned in the same or different clusters in the predicted and true clusterings.

The raw RI score is then "adjusted for chance" into the ARI score using the following scheme:

```
ARI = (RI - Expected_RI) / (max(RI) - Expected_RI)
```

The adjusted Rand index is thus ensured to have a value close to 0.0 for random labeling independently of the number of clusters and samples and exactly 1.0 when the clusterings are identical (up to a permutation).

ARI is a symmetric measure:

```
adjusted_rand_index(a, b) == adjusted_rand_index(b, a)
```

Parameters **clustering** – NodeClustering object

Returns ARI score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> leiden_communities = algorithms.leiden(g)
>>> mod = communities.adjusted_rand_index(leiden_communities)
```

Reference

1. Hubert, L., & Arabie, P. (1985). **Comparing partitions**. Journal of classification, 2(1), 193-218.

average_internal_degree(***kwargs*)

The average internal degree of the algorithms set.

$$f(S) = \frac{2m_S}{n_S}$$

where m_S is the number of algorithms internal edges and n_S is the number of algorithms nodes.

Parameters **summary** – (optional, default True) if **True**, an overall summary is returned for the partition (min, max, avg, std); if **False** a list of community-wise score

Returns a FitnessResult object/a list of community-wise score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.average_internal_degree()
```

avg_odf(***kwargs*)

Average fraction of edges of a node of a algorithms that point outside the algorithms itself.

$$\frac{1}{n_S} \sum_{u \in S} \frac{|\{(u,v) \in E : v \notin S\}|}{d(u)}$$

where E is the graph edge set, v is a node in S, $d(u)$ is the degree of u and n_S is the set of algorithms nodes.

Parameters **summary** – (optional, default True) if **True**, an overall summary is returned for the partition (min, max, avg, std); if **False** a list of community-wise score

Returns a FitnessResult object/a list of community-wise score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>>
>>> communities = eva(g, alpha=alpha)
>>> pur = communities.purity()
```

conductance(***kwargs*)

Fraction of total edge volume that points outside the algorithms.

$$f(S) = \frac{c_S}{2m_S + c_S}$$

where c_S is the number of algorithms nodes and, m_S is the number of algorithms edges

Parameters **summary** – (optional, default True) if **True**, an overall summary is returned for the partition (min, max, avg, std); if **False** a list of community-wise score

Returns a FitnessResult object/a list of community-wise score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.conductance()
```

cut_ratio(***kwargs*)

Fraction of existing edges (out of all possible edges) leaving the algorithms.

..math:: f(S) = frac{c_S}{n_S (n n_S)}

where c_S is the number of algorithms nodes and, n_S is the number of edges on the algorithms boundary

Parameters **summary** – (optional, default True) if **True**, an overall summary is returned for the partition (min, max, avg, std); if **False** a list of community-wise score

Returns a FitnessResult object/a list of community-wise score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.cut_ratio()
```

edges_inside(***kwargs*)

Number of edges internal to the algorithms.

Parameters **summary** – (optional, default True) if **True**, an overall summary is returned for the partition (min, max, avg, std); if **False** a list of community-wise score

Returns a FitnessResult object/a list of community-wise score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.edges_inside()
```

erdos_renyi_modularity()

Erdos-Renyi modularity is a variation of the Newman-Girvan one. It assumes that vertices in a network are connected randomly with a constant probability p.

$$Q(S) = \frac{1}{m}\sum_{c \in S}(m_S \frac{mn_S(n_S 1)}{n(n1)})$$

where m is the number of graph edges, m_S is the number of algorithms edges, l_S is the number of edges from nodes in S to nodes outside S.

Returns the Erdos-Renyi modularity score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.erdos_renyi_modularity()
```

References

Erdos, P., & Renyi, A. (1959). **On random graphs I.** Publ. Math. Debrecen, 6, 290-297.

expansion(***kwargs*)

Number of edges per algorithms node that point outside the cluster.

$$f(S) = \frac{c_S}{n_S}$$

where n_S is the number of edges on the algorithms boundary, c_S is the number of algorithms nodes.

Parameters **summary** – (optional, default True) if **True**, an overall summary is returned for the partition (min, max, avg, std); if **False** a list of community-wise score

Returns a FitnessResult object/a list of community-wise score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.expansion()
```

f1(*clustering*)

Compute the average F1 score of the optimal algorithms matches among the partitions in input. Works on overlapping/non-overlapping complete/partial coverage partitions.

Parameters **clustering** – NodeClustering object

Returns F1 score (harmonic mean of precision and recall)

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> leiden_communities = algorithms.leiden(g)
>>> mod = communities.f1(leiden_communities)
```

Reference

1. Rossetti, G., Pappalardo, L., & Rinzivillo, S. (2016). **A novel approach to evaluate algorithms detection internal on ground truth.** In Complex Networks VII (pp. 133-144). Springer, Cham.

flake_odf (***kwargs*)

Fraction of nodes in S that have fewer edges pointing inside than to the outside of the algorithms.

$$f(S) = \frac{|\{u : u \in S, |\{(u, v) \in E : v \in S\}| < d(u)/2\}|}{n_S}$$

where E is the graph edge set, v is a node in S, $d(u)$ is the degree of u and n_S is the set of algorithms nodes.

Parameters **summary** – (optional, default True) if **True**, an overall summary is returned for the partition (min, max, avg, std); if **False** a list of community-wise score

Returns a FitnessResult object/a list of community-wise score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.flake_odf()
```

fraction_over_median_degree (***kwargs*)

Fraction of algorithms nodes of having internal degree higher than the median degree value.

$$f(S) = \frac{|\{u : u \in S, |\{(u, v) : v \in S\}| > d_m\}|}{n_S}$$

where d_m is the internal degree median value

Parameters **summary** – (optional, default True) if **True**, an overall summary is returned for the partition (min, max, avg, std); if **False** a list of community-wise score

Returns a FitnessResult object/a list of community-wise score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.fraction_over_median_degree()
```

get_description (*parameters_to_display=None*, *precision=3*)

Return a description of the clustering, with the name of the method and its numeric parameters.

Parameters

- **parameters_to_display** – parameters to display. By default, all float parameters.
- **precision** – precision used to plot parameters. default: 3

Returns a string description of the method.

internal_edge_density(***kwargs*)

The internal density of the algorithms set.

$$f(S) = \frac{m_S}{n_S(n_S 1)/2}$$

where m_S is the number of algorithms internal edges and n_S is the number of algorithms nodes.

Parameters **summary** – (optional, default True) if **True**, an overall summary is returned for the partition (min, max, avg, std); if **False** a list of community-wise score

Returns a FitnessResult object/a list of community-wise score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.internal_edge_density()
```

link_modularity()

Quality function designed for directed graphs with overlapping communities.

Returns the link modularity score

Example

```
>>> from cdlib import evaluation
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.link_modularity()
```

max_odf(***kwargs*)

Maximum fraction of edges of a node of a algorithms that point outside the algorithms itself.

$$max_{u \in S} \frac{|\{(u,v) \in E : v \notin S\}|}{d(u)}$$

where E is the graph edge set, v is a node in S and $d(u)$ is the degree of u

Parameters **summary** – (optional, default True) if **True**, an overall summary is returned for the partition (min, max, avg, std); if **False** a list of community-wise score

Returns a FitnessResult object/a list of community-wise score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.max_odf()
```

modularity_density()

The modularity density is one of several propositions that envisioned to palliate the resolution limit issue of modularity based measures. The idea of this metric is to include the information about algorithms size into the expected density of algorithms to avoid the negligence of small and dense communities. For each algorithms C in partition S, it uses the average modularity degree calculated by $d(C) = d^{int(C)} d^{ext(C)}$

where $d^{int(C)}$ and $d^{ext(C)}$ are the average internal and external degrees of C respectively to evaluate the fitness of C in its network. Finally, the modularity density can be calculated as follows:

$$Q(S) = \sum_{C \in S} \frac{1}{n_C} (\sum_{i \in C} k_{iC}^{int} - \sum_{i \in C} k_{iC}^{out})$$

where n_C is the number of nodes in C, k_{iC}^{int} is the degree of node i within C and k_{iC}^{out} is the deree of node i outside C.

Returns the modularity density score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.modularity_density()
```

References

Li, Z., Zhang, S., Wang, R. S., Zhang, X. S., & Chen, L. (2008). **Quantitative function for algorithms detection.** Physical review E, 77(3), 036109.

newman_girvan_modularity()

Difference the fraction of intra algorithms edges of a partition with the expected number of such edges if distributed according to a null model.

In the standard version of modularity, the null model preserves the expected degree sequence of the graph under consideration. In other words, the modularity compares the real network structure with a corresponding one where nodes are connected without any preference about their neighbors.

$$Q(S) = \frac{1}{m} \sum_{c \in S} (m_S - \frac{(2m_S + l_S)^2}{4m})$$

where m is the number of graph edges, m_S is the number of algorithms edges, l_S is the number of edges from nodes in S to nodes outside S.

Returns the Newman-Girvan modularity score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.newman_girvan_modularity()
```

References

Newman, M.E.J. & Girvan, M. **Finding and evaluating algorithms structure in networks.** Physical Review E 69, 26113(2004).

nf1(*clustering*)

Compute the Normalized F1 score of the optimal algorithms matches among the partitions in input. Works on overlapping/non-overlapping complete/partial coverage partitions.

Parameters **clustering** – NodeClustering object

Returns MatchingResult instance

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> leiden_communities = algorithms.leiden(g)
>>> mod = communities.nf1(leiden_communities)
```

Reference

1. Rossetti, G., Pappalardo, L., & Rinzivillo, S. (2016). **A novel approach to evaluate algorithms detection internal on ground truth.**
2. Rossetti, G. (2017). : **RDyn: graph benchmark handling algorithms dynamics. Journal of Complex Networks.** 5(6), 893-912.

normalized_cut (***kwargs*)

Normalized variant of the Cut-Ratio

$$: f(S) = \frac{c_S}{2m_S + c_S} + \frac{c_S}{2(mm_S) + c_S}$$

where m is the number of graph edges, m_S is the number of algorithms internal edges and c_S is the number of algorithms nodes.

Parameters **summary** – (optional, default True) if **True**, an overall summary is returned for the partition (min, max, avg, std); if **False** a list of community-wise score

Returns a FitnessResult object/a list of community-wise score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.normalized_cut()
```

normalized_mutual_information (*clustering*)

Normalized Mutual Information between two clusterings.

Normalized Mutual Information (NMI) is an normalization of the Mutual Information (MI) score to scale the results between 0 (no mutual information) and 1 (perfect correlation). In this function, mutual information is normalized by `sqrt(H(labels_true) * H(labels_pred))`

Parameters **clustering** – NodeClustering object

Returns normalized mutual information score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> leiden_communities = algorithms.leiden(g)
>>> mod = communities.normalized_mutual_information(leiden_communities)
```

omega (*clustering*)

Index of resemblance for overlapping, complete coverage, network clusterings.

Parameters **clustering** – NodeClustering object

Returns omega index

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> leiden_communities = algorithms.leiden(g)
>>> mod = communities.omega(leiden_communities)
```

Reference

1. Gabriel Murray, Giuseppe Carenini, and Raymond Ng. 2012. **Using the omega index for evaluating abstractive algorithms detection.** In Proceedings of Workshop on Evaluation Metrics and System Comparison for Automatic Summarization. Association for Computational Linguistics, Stroudsburg, PA, USA, 10-18.

overlapping_normalized_mutual_information_LFK(*clustering*)

Overlapping Normalized Mutual Information between two clusterings.

Extension of the Normalized Mutual Information (NMI) score to cope with overlapping partitions. This is the version proposed by Lancichinetti et al.

Parameters **clustering** – NodeClustering object

Returns onmi score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> leiden_communities = algorithms.leiden(g)
>>> mod = communities.overlapping_normalized_mutual_information_LFK(leiden_
↪communities)
```

Reference

1. Lancichinetti, A., Fortunato, S., & Kertesz, J. (2009). Detecting the overlapping and hierarchical community structure in complex networks. New Journal of Physics, 11(3), 033015.

overlapping_normalized_mutual_information_MGH(*clustering*, *normalization='max'*)

Overlapping Normalized Mutual Information between two clusterings.

Extension of the Normalized Mutual Information (NMI) score to cope with overlapping partitions. This is the version proposed by McDaid et al. using a different normalization than the original LFR one. See ref. for more details.

Parameters

- **clustering** – NodeClustering object
- **normalization** – one of "max" or "LFK". Default "max" (corresponds to the main method described in the article)

Returns onmi score

Example

```
>>> from cdlib import evaluation, algorithms
>>> g = nx.karate_club_graph()
>>> louvain_communities = algorithms.louvain(g)
>>> leiden_communities = algorithms.leiden(g)
>>> evaluation.overlapping_normalized_mutual_information_MGH(louvain_
↪communities,leiden_communities)
:Reference:
```

1. McDaid, A. F., Greene, D., & Hurley, N. (2011). Normalized mutual information to evaluate overlapping community finding algorithms. arXiv preprint arXiv:1110.2515. Chicago

significance()

Significance estimates how likely a partition of dense communities appear in a random graph.

Returns the significance score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.significance()
```

References

Traag, V. A., Aldecoa, R., & Delvenne, J. C. (2015). **Detecting communities using asymptotical surprise.** Physical Review E, 92(2), 022816.

size(***kwargs*)

Size is the number of nodes in the community

Parameters **summary** – (optional, default True) if **True**, an overall summary is returned for the partition (min, max, avg, std); if **False** a list of community-wise score

Returns a FitnessResult object/a list of community-wise score

Example:

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.size()
```

surprise()

Surprise is statistical approach proposes a quality metric assuming that edges between vertices emerge randomly according to a hyper-geometric distribution.

According to the Surprise metric, the higher the score of a partition, the less likely it is resulted from a random realization, the better the quality of the algorithms structure.

Returns the surprise score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.surprise()
```

References

Traag, V. A., Aldecoa, R., & Delvenne, J. C. (2015). **Detecting communities using asymptotical surprise.** Physical Review E, 92(2), 022816.

to_json()
Generate a JSON representation of the algorithms object

Returns a JSON formatted string representing the object

to_node_community_map()
Generate a <node, list(communities)> representation of the current clustering

Returns dict of the form <node, list(communities)>

triangle_participation_ratio(***kwargs*)
Fraction of algorithms nodes that belong to a triad.

$$f(S) = \frac{|\{u : u \in S, \{(v, w) : v, w \in S, (u, v) \in E, (u, w) \in E, (v, w) \in E\} \neq \emptyset\}|}{n_S}$$

where n_S is the set of algorithms nodes.

Parameters **summary** – (optional, default True) if **True**, an overall summary is returned for the partition (min, max, avg, std); if **False** a list of community-wise score

Returns a FitnessResult object/a list of community-wise score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.triangle_participation_ratio()
```

variation_of_information(*clustering*)
Variation of Information among two nodes partitions.

$$ H(p)+H(q)-2MI(p, q) $$

where MI is the mutual information, H the partition entropy and p,q are the algorithms sets

Parameters **clustering** – NodeClustering object

Returns VI score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> leiden_communities = algorithms.leiden(g)
>>> mod = communities.variation_of_information(leiden_communities)
```

Reference

1. Meila, M. (2007). **Comparing clusterings - an information based distance.** Journal of Multivariate Analysis, 98, 873-895. doi:10.1016/j.jmva.2006.11.013

z_modularity()
Z-modularity is another variant of the standard modularity proposed to avoid the resolution limit. The concept of this version is based on an observation that the difference between the fraction of edges inside

communities and the expected number of such edges in a null model should not be considered as the only contribution to the final quality of algorithms structure.

Returns the z-modularity score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.z_modularity()
```

References

Miyauchi, Atsushi, and Yasushi Kawase. **Z-score-based modularity for algorithms detection in networks.** PloS one 11.1 (2016): e0147805.

Methods

Data transformation and IO

NodeClustering.to_json()	Generate a JSON representation of the algorithms object
NodeClustering.to_node_community_map()	Generate a <node, list(communities)> representation of the current clustering

Evaluating Node Clustering

NodeClustering.link_modularity()	Quality function designed for directed graphs with overlapping communities.
NodeClustering.normalized_cut(**kwargs)	Normalized variant of the Cut-Ratio
NodeClustering.internal_edge_density(**kwargs)	The internal density of the algorithms set.
NodeClustering.average_internal_degree(**kwargs)	The average internal degree of the algorithms set.
NodeClustering.fraction_over_median_degree(**kwargs)	Fraction of algorithms nodes of having internal degree higher than the median degree value.
NodeClustering.expansion(**kwargs)	Number of edges per algorithms node that point outside the cluster.
NodeClustering.cut_ratio(**kwargs)	Fraction of existing edges (out of all possible edges) leaving the algorithms.
NodeClustering.edges_inside(**kwargs)	Number of edges internal to the algorithms.
NodeClustering.conductance(**kwargs)	Fraction of total edge volume that points outside the algorithms.
NodeClustering.max_odf(**kwargs)	Maximum fraction of edges of a node of a algorithms that point outside the algorithms itself.
NodeClustering.avg_odf(**kwargs)	Average fraction of edges of a node of a algorithms that point outside the algorithms itself.
NodeClustering.flake_odf(**kwargs)	Fraction of nodes in S that have fewer edges pointing inside than to the outside of the algorithms.
NodeClustering.triangle_participation_ratio()	Fraction of algorithms nodes that belong to a triad.

Continued on next page

Table 2 – continued from previous page

NodeClustering.newman_girvan_modularity()	Difference the fraction of intra algorithms edges of a partition with the expected number of such edges if distributed according to a null model.
NodeClustering.erdos_renyi_modularity()	Erdos-Renyi modularity is a variation of the Newman-Girvan one.
NodeClustering.modularity_density()	The modularity density is one of several propositions that envisioned to palliate the resolution limit issue of modularity based measures.
NodeClustering.z_modularity()	Z-modularity is another variant of the standard modularity proposed to avoid the resolution limit.
NodeClustering.surprise()	Surprise is statistical approach proposes a quality metric assuming that edges between vertices emerge randomly according to a hyper-geometric distribution.
NodeClustering.significance()	Significance estimates how likely a partition of dense communities appear in a random graph.

Comparing Node Clusterings

NodeClustering.normalized_mutual_information(clustering)	Normalized Mutual Information between two clusterings.
NodeClustering.overlapping_normalized_mutual_information_LFK(clustering)	Overlapping Normalized Mutual Information between two clusterings.
NodeClustering.overlapping_normalized_mutual_information_MGH(clustering)	Overlapping Normalized Mutual Information between two clusterings.
NodeClustering.omega(clustering)	Index of resemblance for overlapping, complete coverage, network clusterings.
NodeClustering.f1(clustering)	Compute the average F1 score of the optimal algorithms matches among the partitions in input.
NodeClustering.nf1(clustering)	Compute the Normalized F1 score of the optimal algorithms matches among the partitions in input.
NodeClustering.adjusted_rand_index(clustering)	Rand index adjusted for chance.
NodeClustering.adjusted_mutual_information(clustering)	Adjusted Mutual Information between two clusterings.
NodeClustering.variation_of_information(clustering)	Variation of Information among two nodes partitions.

Fuzzy Node Clustering

Overview

class FuzzyNodeClustering(*communities*, *node_allocation*, *graph*, *method_name*, *method_parameters=None*, *overlap=False*)

Fuzzy Node Communities representation.

Parameters

- **communities** – list of communities
- **node_allocation** – dictionary specifying for each node the allocation of probability toward the communities it is placed in
- **graph** – a networkx/igraph object
- **method_name** – community discovery algorithm name

- **method_parameters** – configuration for the community discovery algorithm used
- **overlap** – boolean, whether the partition is overlapping or not

adjusted_mutual_information (*clustering*)

Adjusted Mutual Information between two clusterings.

Adjusted Mutual Information (AMI) is an adjustment of the Mutual Information (MI) score to account for chance. It accounts for the fact that the MI is generally higher for two clusterings with a larger number of clusters, regardless of whether there is actually more information shared. For two clusterings U and V, the AMI is given as:

```
AMI(U, V) = [MI(U, V) - E(MI(U, V))] / [max(H(U), H(V)) - E(MI(U, V))]
```

This metric is independent of the absolute values of the labels: a permutation of the class or cluster label values won't change the score value in any way.

This metric is furthermore symmetric: switching `label_true` with `label_pred` will return the same score value. This can be useful to measure the agreement of two independent label assignments strategies on the same dataset when the real ground truth is not known.

Be mindful that this function is an order of magnitude slower than other metrics, such as the Adjusted Rand Index.

Parameters **clustering** – NodeClustering object

Returns AMI score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> leiden_communities = algorithms.leiden(g)
>>> mod = communities.adjusted_mutual_information(leiden_communities)
```

Reference

1. Vinh, N. X., Epps, J., & Bailey, J. (2010). **Information theoretic measures for clusterings comparison: Variants, properties, normalization and correction for chance.** Journal of Machine Learning Research, 11(Oct), 2837-2854.

adjusted_rand_index (*clustering*)

Rand index adjusted for chance.

The Rand Index computes a similarity measure between two clusterings by considering all pairs of samples and counting pairs that are assigned in the same or different clusters in the predicted and true clusterings.

The raw RI score is then "adjusted for chance" into the ARI score using the following scheme:

```
ARI = (RI - Expected_RI) / (max(RI) - Expected_RI)
```

The adjusted Rand index is thus ensured to have a value close to 0.0 for random labeling independently of the number of clusters and samples and exactly 1.0 when the clusterings are identical (up to a permutation).

ARI is a symmetric measure:

```
adjusted_rand_index(a, b) == adjusted_rand_index(b, a)
```

Parameters **clustering** – NodeClustering object

Returns ARI score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> leiden_communities = algorithms.leiden(g)
>>> mod = communities.adjusted_rand_index(leiden_communities)
```

Reference

1. Hubert, L., & Arabie, P. (1985). **Comparing partitions**. Journal of classification, 2(1), 193-218.

average_internal_degree(***kwargs*)

The average internal degree of the algorithms set.

$$f(S) = \frac{2m_S}{n_S}$$

where m_S is the number of algorithms internal edges and n_S is the number of algorithms nodes.

Parameters **summary** – (optional, default True) if **True**, an overall summary is returned for the partition (min, max, avg, std); if **False** a list of community-wise score

Returns a FitnessResult object/a list of community-wise score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.average_internal_degree()
```

avg_odf(***kwargs*)

Average fraction of edges of a node of a algorithms that point outside the algorithms itself.

$$\frac{1}{n_S} \sum_{u \in S} \frac{|\{(u,v) \in E : v \notin S\}|}{d(u)}$$

where E is the graph edge set, v is a node in S, $d(u)$ is the degree of u and n_S is the set of algorithms nodes.

Parameters **summary** – (optional, default True) if **True**, an overall summary is returned for the partition (min, max, avg, std); if **False** a list of community-wise score

Returns a FitnessResult object/a list of community-wise score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>>
>>> communities = eva(g, alpha=alpha)
>>> pur = communities.purity()
```

conductance(***kwargs*)

Fraction of total edge volume that points outside the algorithms.

$$f(S) = \frac{c_S}{2m_S + c_S}$$

where c_S is the number of algorithms nodes and, m_S is the number of algorithms edges

Parameters **summary** – (optional, default True) if **True**, an overall summary is returned for the partition (min, max, avg, std); if **False** a list of community-wise score

Returns a FitnessResult object/a list of community-wise score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.conductance()
```

cut_ratio(***kwargs*)

Fraction of existing edges (out of all possible edges) leaving the algorithms.

..math:: f(S) = frac{c_S}{n_S (n n_S)}

where c_S is the number of algorithms nodes and, n_S is the number of edges on the algorithms boundary

Parameters **summary** – (optional, default True) if **True**, an overall summary is returned for the partition (min, max, avg, std); if **False** a list of community-wise score

Returns a FitnessResult object/a list of community-wise score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.cut_ratio()
```

edges_inside(***kwargs*)

Number of edges internal to the algorithms.

Parameters **summary** – (optional, default True) if **True**, an overall summary is returned for the partition (min, max, avg, std); if **False** a list of community-wise score

Returns a FitnessResult object/a list of community-wise score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.edges_inside()
```

erdos_renyi_modularity()

Erdos-Renyi modularity is a variation of the Newman-Girvan one. It assumes that vertices in a network are connected randomly with a constant probability p.

$$Q(S) = \frac{1}{m}\sum_{c \in S}(m_S \frac{m n_S(n_S 1)}{n(n1)})$$

where m is the number of graph edges, m_S is the number of algorithms edges, l_S is the number of edges from nodes in S to nodes outside S.

Returns the Erdos-Renyi modularity score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.erdos_renyi_modularity()
```

References

Erdos, P., & Renyi, A. (1959). **On random graphs I.** Publ. Math. Debrecen, 6, 290-297.

expansion (***kwargs*)

Number of edges per algorithms node that point outside the cluster.

$$f(S) = \frac{c_S}{n_S}$$

where n_S is the number of edges on the algorithms boundary, c_S is the number of algorithms nodes.

Parameters **summary** – (optional, default True) if **True**, an overall summary is returned for the partition (min, max, avg, std); if **False** a list of community-wise score

Returns a FitnessResult object/a list of community-wise score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.expansion()
```

f1 (*clustering*)

Compute the average F1 score of the optimal algorithms matches among the partitions in input. Works on overlapping/non-overlapping complete/partial coverage partitions.

Parameters **clustering** – NodeClustering object

Returns F1 score (harmonic mean of precision and recall)

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> leiden_communities = algorithms.leiden(g)
>>> mod = communities.f1(leiden_communities)
```

Reference

1. Rossetti, G., Pappalardo, L., & Rinzivillo, S. (2016). **A novel approach to evaluate algorithms detection internal on ground truth.** In Complex Networks VII (pp. 133-144). Springer, Cham.

flake_odf (***kwargs*)

Fraction of nodes in S that have fewer edges pointing inside than to the outside of the algorithms.

$$f(S) = \frac{|\{u : u \in S, |\{(u, v) \in E : v \in S\}| < d(u)/2\}|}{n_S}$$

where E is the graph edge set, v is a node in S, $d(u)$ is the degree of u and n_S is the set of algorithms nodes.

Parameters **summary** – (optional, default True) if **True**, an overall summary is returned for the partition (min, max, avg, std); if **False** a list of community-wise score

Returns a FitnessResult object/a list of community-wise score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.flake_odf()
```

fraction_over_median_degree (***kwargs*)
Fraction of algorithms nodes of having internal degree higher than the median degree value.

$$f(S) = \frac{|\{u : u \in S, |\{(u, v) : v \in S\}| > d_m\}|}{n_S}$$

where d_m is the internal degree median value

Parameters **summary** – (optional, default True) if **True**, an overall summary is returned for the partition (min, max, avg, std); if **False** a list of community-wise score

Returns a FitnessResult object/a list of community-wise score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.fraction_over_median_degree()
```

get_description (*parameters_to_display=None*, *precision=3*)
Return a description of the clustering, with the name of the method and its numeric parameters.

Parameters

- **parameters_to_display** – parameters to display. By default, all float parameters.
- **precision** – precision used to plot parameters. default: 3

Returns a string description of the method.

internal_edge_density (***kwargs*)
The internal density of the algorithms set.

$$f(S) = \frac{m_S}{n_S(n_S 1)/2}$$

where m_S is the number of algorithms internal edges and n_S is the number of algorithms nodes.

Parameters **summary** – (optional, default True) if **True**, an overall summary is returned for the partition (min, max, avg, std); if **False** a list of community-wise score

Returns a FitnessResult object/a list of community-wise score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.internal_edge_density()
```

link_modularity()
Quality function designed for directed graphs with overlapping communities.

Returns the link modularity score

Example

```
>>> from cdlib import evaluation
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.link_modularity()
```

max_odf(***kwargs*)
Maximum fraction of edges of a node of a algorithms that point outside the algorithms itself.

$$max_{u \in S} \frac{|\{(u,v) \in E : v \notin S\}|}{d(u)}$$

where E is the graph edge set, v is a node in S and $d(u)$ is the degree of u

Parameters **summary** – (optional, default True) if **True**, an overall summary is returned for the partition (min, max, avg, std); if **False** a list of community-wise score

Returns a FitnessResult object/a list of community-wise score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.max_odf()
```

modularity_density()
The modularity density is one of several propositions that envisioned to palliate the resolution limit issue of modularity based measures. The idea of this metric is to include the information about algorithms size into the expected density of algorithms to avoid the negligence of small and dense communities. For each algorithms C in partition S, it uses the average modularity degree calculated by $d(C) = d^{int(C)} d^{ext(C)}$ where $d^{int(C)}$ and $d^{ext(C)}$ are the average internal and external degrees of C respectively to evaluate the fitness of C in its network. Finally, the modularity density can be calculated as follows:

$$Q(S) = \sum_{C \in S} \frac{1}{n_C} (\sum_{i \in C} k_{iC}^{int} - \sum_{i \in C} k_{iC}^{out})$$

where n_C is the number of nodes in C, k_{iC}^{int} is the degree of node i within C and k_{iC}^{out} is the deree of node i outside C.

Returns the modularity density score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.modularity_density()
```

References

Li, Z., Zhang, S., Wang, R. S., Zhang, X. S., & Chen, L. (2008). **Quantitative function for algorithms detection.** Physical review E, 77(3), 036109.

newman_girvan_modularity()

Difference the fraction of intra algorithms edges of a partition with the expected number of such edges if distributed according to a null model.

In the standard version of modularity, the null model preserves the expected degree sequence of the graph under consideration. In other words, the modularity compares the real network structure with a corresponding one where nodes are connected without any preference about their neighbors.

$$Q(S) = \frac{1}{m}\sum_{c \in S}(m_S - \frac{(2m_S + l_S)^2}{4m})$$

where m is the number of graph edges, m_S is the number of algorithms edges, l_S is the number of edges from nodes in S to nodes outside S.

Returns the Newman-Girvan modularity score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.newman_girvan_modularity()
```

References

Newman, M.E.J. & Girvan, M. **Finding and evaluating algorithms structure in networks.** Physical Review E 69, 26113(2004).

nf1(*clustering*)

Compute the Normalized F1 score of the optimal algorithms matches among the partitions in input. Works on overlapping/non-overlapping complete/partial coverage partitions.

Parameters **clustering** – NodeClustering object

Returns MatchingResult instance

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> leiden_communities = algorithms.leiden(g)
>>> mod = communities.nf1(leiden_communities)
```

Reference

1. Rossetti, G., Pappalardo, L., & Rinzivillo, S. (2016). **A novel approach to evaluate algorithms detection internal on ground truth.**
2. Rossetti, G. (2017). : **RDyn: graph benchmark handling algorithms dynamics. Journal of Complex Networks.** 5(6), 893-912.

normalized_cut(***kwargs*)

Normalized variant of the Cut-Ratio

$$: f(S) = \frac{c_S}{2m_S + c_S} + \frac{c_S}{2(mm_S) + c_S}$$

where m is the number of graph edges, m_S is the number of algorithms internal edges and c_S is the number of algorithms nodes.

Parameters **summary** – (optional, default True) if **True**, an overall summary is returned for the partition (min, max, avg, std); if **False** a list of community-wise score

Returns a FitnessResult object/a list of community-wise score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.normalized_cut()
```

normalized_mutual_information (*clustering*)

Normalized Mutual Information between two clusterings.

Normalized Mutual Information (NMI) is an normalization of the Mutual Information (MI) score to scale the results between 0 (no mutual information) and 1 (perfect correlation). In this function, mutual information is normalized by `sqrt(H(labels_true) * H(labels_pred))`

Parameters **clustering** – NodeClustering object

Returns normalized mutual information score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> leiden_communities = algorithms.leiden(g)
>>> mod = communities.normalized_mutual_information(leiden_communities)
```

omega (*clustering*)

Index of resemblance for overlapping, complete coverage, network clusterings.

Parameters **clustering** – NodeClustering object

Returns omega index

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> leiden_communities = algorithms.leiden(g)
>>> mod = communities.omega(leiden_communities)
```

Reference

1. Gabriel Murray, Giuseppe Carenini, and Raymond Ng. 2012. **Using the omega index for evaluating abstractive algorithms detection.** In Proceedings of Workshop on Evaluation Metrics and System Comparison for Automatic Summarization. Association for Computational Linguistics, Stroudsburg, PA, USA, 10-18.

overlapping_normalized_mutual_information_LFK (*clustering*)

Overlapping Normalized Mutual Information between two clusterings.

Extension of the Normalized Mutual Information (NMI) score to cope with overlapping partitions. This is the version proposed by Lancichinetti et al.

Parameters **clustering** – NodeClustering object

Returns onmi score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> leiden_communities = algorithms.leiden(g)
>>> mod = communities.overlapping_normalized_mutual_information_LFK(leiden_
↪communities)
```

Reference

1. Lancichinetti, A., Fortunato, S., & Kertesz, J. (2009). Detecting the overlapping and hierarchical community structure in complex networks. New Journal of Physics, 11(3), 033015.

overlapping_normalized_mutual_information_MGH(*clustering*, *normalization='max'*)

Overlapping Normalized Mutual Information between two clusterings.

Extension of the Normalized Mutual Information (NMI) score to cope with overlapping partitions. This is the version proposed by McDaid et al. using a different normalization than the original LFR one. See ref. for more details.

Parameters

- **clustering** – NodeClustering object
- **normalization** – one of "max" or "LFK". Default "max" (corresponds to the main method described in the article)

Returns onmi score

Example

```
>>> from cdlib import evaluation, algorithms
>>> g = nx.karate_club_graph()
>>> louvain_communities = algorithms.louvain(g)
>>> leiden_communities = algorithms.leiden(g)
>>> evaluation.overlapping_normalized_mutual_information_MGH(louvain_
↪communities,leiden_communities)
:Reference:
```

1. McDaid, A. F., Greene, D., & Hurley, N. (2011). Normalized mutual information to evaluate overlapping community finding algorithms. arXiv preprint arXiv:1110.2515. Chicago

significance()

Significance estimates how likely a partition of dense communities appear in a random graph.

Returns the significance score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.significance()
```

References

Traag, V. A., Aldecoa, R., & Delvenne, J. C. (2015). **Detecting communities using asymptotical surprise.** Physical Review E, 92(2), 022816.

size(***kwargs*)
Size is the number of nodes in the community

Parameters **summary** – (optional, default True) if **True**, an overall summary is returned for the partition (min, max, avg, std); if **False** a list of community-wise score

Returns a FitnessResult object/a list of community-wise score

Example:

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.size()
```

surprise()
Surprise is statistical approach proposes a quality metric assuming that edges between vertices emerge randomly according to a hyper-geometric distribution.

According to the Surprise metric, the higher the score of a partition, the less likely it is resulted from a random realization, the better the quality of the algorithms structure.

Returns the surprise score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.surprise()
```

References

Traag, V. A., Aldecoa, R., & Delvenne, J. C. (2015). **Detecting communities using asymptotical surprise.** Physical Review E, 92(2), 022816.

to_json()
Generate a JSON representation of the algorithms object

Returns a JSON formatted string representing the object

to_node_community_map()
Generate a <node, list(communities)> representation of the current clustering

Returns dict of the form <node, list(communities)>

triangle_participation_ratio(***kwargs*)
Fraction of algorithms nodes that belong to a triad.

$$f(S) = \frac{|\{u : u \in S, \{(v, w) : v, w \in S, (u, v) \in E, (u, w) \in E, (v, w) \in E\} \neq \emptyset\}|}{n_S}$$

where n_S is the set of algorithms nodes.

Parameters **summary** – (optional, default True) if **True**, an overall summary is returned for the partition (min, max, avg, std); if **False** a list of community-wise score

Returns a FitnessResult object/a list of community-wise score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.triangle_participation_ratio()
```

variation_of_information(*clustering*)

Variation of Information among two nodes partitions.

$$ H(p)+H(q)-2MI(p, q) $$

where MI is the mutual information, H the partition entropy and p,q are the algorithms sets

Parameters **clustering** – NodeClustering object

Returns VI score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> leiden_communities = algorithms.leiden(g)
>>> mod = communities.variation_of_information(leiden_communities)
```

Reference

1. Meila, M. (2007). **Comparing clusterings - an information based distance.** Journal of Multivariate Analysis, 98, 873-895. doi:10.1016/j.jmva.2006.11.013

z_modularity()

Z-modularity is another variant of the standard modularity proposed to avoid the resolution limit. The concept of this version is based on an observation that the difference between the fraction of edges inside communities and the expected number of such edges in a null model should not be considered as the only contribution to the final quality of algorithms structure.

Returns the z-modularity score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.z_modularity()
```

References

Miyauchi, Atsushi, and Yasushi Kawase. **Z-score-based modularity for algorithms detection in networks.** PloS one 11.1 (2016): e0147805.

Methods

Data transformation and IO

FuzzyNodeClustering.to_json()	Generate a JSON representation of the algorithms object

Continued on next page

Table 4 – continued from previous page

FuzzyNodeClustering.to_node_community_map()	Generate a <node, list(communities)> representation of the current clustering

Evaluating Node Clustering

FuzzyNodeClustering.link_modularity()	Quality function designed for directed graphs with overlapping communities.
FuzzyNodeClustering.normalized_cut(**kwargs)	Normalized variant of the Cut-Ratio
FuzzyNodeClustering.internal_edge_density(...)	The internal density of the algorithms set.
FuzzyNodeClustering.average_internal_degree(...)	The average internal degree of the algorithms set.
FuzzyNodeClustering.fraction_over_median_degree(...)	Fraction of algorithms nodes of having internal degree higher than the median degree value.
FuzzyNodeClustering.expansion(**kwargs)	Number of edges per algorithms node that point outside the cluster.
FuzzyNodeClustering.cut_ratio(**kwargs)	Fraction of existing edges (out of all possible edges) leaving the algorithms.
FuzzyNodeClustering.edges_inside(**kwargs)	Number of edges internal to the algorithms.
FuzzyNodeClustering.conductance(**kwargs)	Fraction of total edge volume that points outside the algorithms.
FuzzyNodeClustering.max_odf(**kwargs)	Maximum fraction of edges of a node of a algorithms that point outside the algorithms itself.
FuzzyNodeClustering.avg_odf(**kwargs)	Average fraction of edges of a node of a algorithms that point outside the algorithms itself.
FuzzyNodeClustering.flake_odf(**kwargs)	Fraction of nodes in S that have fewer edges pointing inside than to the outside of the algorithms.
FuzzyNodeClustering.triangle_participation_ratio(...)	Fraction of algorithms nodes that belong to a triad.
FuzzyNodeClustering.newman_girvan_modularity()	Difference the fraction of intra algorithms edges of a partition with the expected number of such edges if distributed according to a null model.
FuzzyNodeClustering.erdos_renyi_modularity()	Erdos-Renyi modularity is a variation of the Newman-Girvan one.
FuzzyNodeClustering.modularity_density()	The modularity density is one of several propositions that envisioned to palliate the resolution limit issue of modularity based measures.
FuzzyNodeClustering.z_modularity()	Z-modularity is another variant of the standard modularity proposed to avoid the resolution limit.
FuzzyNodeClustering.surprise()	Surprise is statistical approach proposes a quality metric assuming that edges between vertices emerge randomly according to a hyper-geometric distribution.
FuzzyNodeClustering.significance()	Significance estimates how likely a partition of dense communities appear in a random graph.

Attributed Node Clustering

Overview

class AttrNodeClustering(*communities*, *graph*, *method_name*, *coms_labels=None*, *method_parameters=None*, *overlap=False*)

Attribute Node Communities representation.

Parameters

- **communities** – list of communities
- **graph** – a networkx/igraph object
- **method_name** – community discovery algorithm name
- **coms_labels** – dictionary specifying for each community the frequency of the attribute values
- **method_parameters** – configuration for the community discovery algorithm used
- **overlap** – boolean, whether the partition is overlapping or not

adjusted_mutual_information(*clustering*)

Adjusted Mutual Information between two clusterings.

Adjusted Mutual Information (AMI) is an adjustment of the Mutual Information (MI) score to account for chance. It accounts for the fact that the MI is generally higher for two clusterings with a larger number of clusters, regardless of whether there is actually more information shared. For two clusterings U and V, the AMI is given as:

```
AMI(U, V) = [MI(U, V) - E(MI(U, V))] / [max(H(U), H(V)) - E(MI(U, V))]
```

This metric is independent of the absolute values of the labels: a permutation of the class or cluster label values won't change the score value in any way.

This metric is furthermore symmetric: switching `label_true` with `label_pred` will return the same score value. This can be useful to measure the agreement of two independent label assignments strategies on the same dataset when the real ground truth is not known.

Be mindful that this function is an order of magnitude slower than other metrics, such as the Adjusted Rand Index.

Parameters **clustering** – NodeClustering object

Returns AMI score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> leiden_communities = algorithms.leiden(g)
>>> mod = communities.adjusted_mutual_information(leiden_communities)
```

Reference

1. Vinh, N. X., Epps, J., & Bailey, J. (2010). **Information theoretic measures for clusterings comparison: Variants, properties, normalization and correction for chance.** Journal of Machine Learning Research, 11(Oct), 2837-2854.

adjusted_rand_index(*clustering*)

Rand index adjusted for chance.

The Rand Index computes a similarity measure between two clusterings by considering all pairs of samples and counting pairs that are assigned in the same or different clusters in the predicted and true clusterings.

The raw RI score is then "adjusted for chance" into the ARI score using the following scheme:

```
ARI = (RI - Expected_RI) / (max(RI) - Expected_RI)
```

The adjusted Rand index is thus ensured to have a value close to 0.0 for random labeling independently of the number of clusters and samples and exactly 1.0 when the clusterings are identical (up to a permutation).

ARI is a symmetric measure:

```
adjusted_rand_index(a, b) == adjusted_rand_index(b, a)
```

Parameters **clustering** – NodeClustering object

Returns ARI score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> leiden_communities = algorithms.leiden(g)
>>> mod = communities.adjusted_rand_index(leiden_communities)
```

Reference

1. Hubert, L., & Arabie, P. (1985). **Comparing partitions**. Journal of classification, 2(1), 193-218.

average_internal_degree(***kwargs*)

The average internal degree of the algorithms set.

$$f(S) = \frac{2m_S}{n_S}$$

where m_S is the number of algorithms internal edges and n_S is the number of algorithms nodes.

Parameters **summary** – (optional, default True) if **True**, an overall summary is returned for the partition (min, max, avg, std); if **False** a list of community-wise score

Returns a FitnessResult object/a list of community-wise score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.average_internal_degree()
```

avg_odf(***kwargs*)

Average fraction of edges of a node of a algorithms that point outside the algorithms itself.

$$\frac{1}{n_S} \sum_{u \in S} \frac{|\{(u,v) \in E : v \notin S\}|}{d(u)}$$

where E is the graph edge set, v is a node in S, $d(u)$ is the degree of u and n_S is the set of algorithms nodes.

Parameters **summary** – (optional, default True) if **True**, an overall summary is returned for the partition (min, max, avg, std); if **False** a list of community-wise score

Returns a FitnessResult object/a list of community-wise score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>>
>>> communities = eva(g, alpha=alpha)
>>> pur = communities.purity()
```

conductance (***kwargs*)

Fraction of total edge volume that points outside the algorithms.

$$f(S) = \frac{c_S}{2m_S + c_S}$$

where c_S is the number of algorithms nodes and, m_S is the number of algorithms edges

Parameters **summary** – (optional, default True) if **True**, an overall summary is returned for the partition (min, max, avg, std); if **False** a list of community-wise score

Returns a FitnessResult object/a list of community-wise score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.conductance()
```

cut_ratio (***kwargs*)

Fraction of existing edges (out of all possible edges) leaving the algorithms.

..math:: f(S) = frac{c_S}{n_S (n n_S)}

where c_S is the number of algorithms nodes and, n_S is the number of edges on the algorithms boundary

Parameters **summary** – (optional, default True) if **True**, an overall summary is returned for the partition (min, max, avg, std); if **False** a list of community-wise score

Returns a FitnessResult object/a list of community-wise score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.cut_ratio()
```

edges_inside (***kwargs*)

Number of edges internal to the algorithms.

Parameters **summary** – (optional, default True) if **True**, an overall summary is returned for the partition (min, max, avg, std); if **False** a list of community-wise score

Returns a FitnessResult object/a list of community-wise score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.edges_inside()
```

erdos_renyi_modularity()

Erdos-Renyi modularity is a variation of the Newman-Girvan one. It assumes that vertices in a network are connected randomly with a constant probability p.

$$Q(S) = \frac{1}{m}\sum_{c \in S}(m_S \frac{mn_S(n_S 1)}{n(n1)})$$

where m is the number of graph edges, m_S is the number of algorithms edges, l_S is the number of edges from nodes in S to nodes outside S.

Returns the Erdos-Renyi modularity score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.erdos_renyi_modularity()
```

References

Erdos, P., & Renyi, A. (1959). **On random graphs I.** Publ. Math. Debrecen, 6, 290-297.

expansion(***kwargs*)

Number of edges per algorithms node that point outside the cluster.

$$f(S) = \frac{c_S}{n_S}$$

where n_S is the number of edges on the algorithms boundary, c_S is the number of algorithms nodes.

Parameters **summary** – (optional, default True) if **True**, an overall summary is returned for the partition (min, max, avg, std); if **False** a list of community-wise score

Returns a FitnessResult object/a list of community-wise score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.expansion()
```

f1(*clustering*)

Compute the average F1 score of the optimal algorithms matches among the partitions in input. Works on overlapping/non-overlapping complete/partial coverage partitions.

Parameters **clustering** – NodeClustering object

Returns F1 score (harmonic mean of precision and recall)

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> leiden_communities = algorithms.leiden(g)
>>> mod = communities.f1(leiden_communities)
```

Reference

1. Rossetti, G., Pappalardo, L., & Rinzivillo, S. (2016). **A novel approach to evaluate algorithms detection internal on ground truth.** In Complex Networks VII (pp. 133-144). Springer, Cham.

flake_odf (***kwargs*)

Fraction of nodes in S that have fewer edges pointing inside than to the outside of the algorithms.

$$f(S) = \frac{|\{u : u \in S, |\{(u, v) \in E : v \in S\}| < d(u)/2\}|}{n_S}$$

where E is the graph edge set, v is a node in S, $d(u)$ is the degree of u and n_S is the set of algorithms nodes.

Parameters **summary** – (optional, default True) if **True**, an overall summary is returned for the partition (min, max, avg, std); if **False** a list of community-wise score

Returns a FitnessResult object/a list of community-wise score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.flake_odf()
```

fraction_over_median_degree (***kwargs*)

Fraction of algorithms nodes of having internal degree higher than the median degree value.

$$f(S) = \frac{|\{u : u \in S, |\{(u, v) : v \in S\}| > d_m\}|}{n_S}$$

where d_m is the internal degree median value

Parameters **summary** – (optional, default True) if **True**, an overall summary is returned for the partition (min, max, avg, std); if **False** a list of community-wise score

Returns a FitnessResult object/a list of community-wise score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.fraction_over_median_degree()
```

get_description (*parameters_to_display=None*, *precision=3*)

Return a description of the clustering, with the name of the method and its numeric parameters.

Parameters

- **parameters_to_display** – parameters to display. By default, all float parameters.
- **precision** – precision used to plot parameters. default: 3

Returns a string description of the method.

internal_edge_density(***kwargs*)

The internal density of the algorithms set.

$$f(S) = \frac{m_S}{n_S(n_S 1)/2}$$

where m_S is the number of algorithms internal edges and n_S is the number of algorithms nodes.

Parameters **summary** – (optional, default True) if **True**, an overall summary is returned for the partition (min, max, avg, std); if **False** a list of community-wise score

Returns a FitnessResult object/a list of community-wise score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.internal_edge_density()
```

link_modularity()

Quality function designed for directed graphs with overlapping communities.

Returns the link modularity score

Example

```
>>> from cdlib import evaluation
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.link_modularity()
```

max_odf(***kwargs*)

Maximum fraction of edges of a node of a algorithms that point outside the algorithms itself.

$$max_{u \in S} \frac{|\{(u,v) \in E : v \notin S\}|}{d(u)}$$

where E is the graph edge set, v is a node in S and $d(u)$ is the degree of u

Parameters **summary** – (optional, default True) if **True**, an overall summary is returned for the partition (min, max, avg, std); if **False** a list of community-wise score

Returns a FitnessResult object/a list of community-wise score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.max_odf()
```

modularity_density()

The modularity density is one of several propositions that envisioned to palliate the resolution limit issue of modularity based measures. The idea of this metric is to include the information about algorithms size into the expected density of algorithms to avoid the negligence of small and dense communities. For each algorithms C in partition S, it uses the average modularity degree calculated by $d(C) = d^{int(C)} d^{ext(C)}$

where $d^{int(C)}$ and $d^{ext(C)}$ are the average internal and external degrees of C respectively to evaluate the fitness of C in its network. Finally, the modularity density can be calculated as follows:

$$Q(S) = \sum_{C \in S} \frac{1}{n_C} (\sum_{i \in C} k_{iC}^{int} - \sum_{i \in C} k_{iC}^{out})$$

where n_C is the number of nodes in C, k_{iC}^{int} is the degree of node i within C and k_{iC}^{out} is the deree of node i outside C.

Returns the modularity density score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.modularity_density()
```

References

Li, Z., Zhang, S., Wang, R. S., Zhang, X. S., & Chen, L. (2008). **Quantitative function for algorithms detection.** Physical review E, 77(3), 036109.

newman_girvan_modularity()

Difference the fraction of intra algorithms edges of a partition with the expected number of such edges if distributed according to a null model.

In the standard version of modularity, the null model preserves the expected degree sequence of the graph under consideration. In other words, the modularity compares the real network structure with a corresponding one where nodes are connected without any preference about their neighbors.

$$Q(S) = \frac{1}{m} \sum_{c \in S} (m_S - \frac{(2m_S + l_S)^2}{4m})$$

where m is the number of graph edges, m_S is the number of algorithms edges, l_S is the number of edges from nodes in S to nodes outside S.

Returns the Newman-Girvan modularity score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.newman_girvan_modularity()
```

References

Newman, M.E.J. & Girvan, M. **Finding and evaluating algorithms structure in networks.** Physical Review E 69, 26113(2004).

nf1(*clustering*)

Compute the Normalized F1 score of the optimal algorithms matches among the partitions in input. Works on overlapping/non-overlapping complete/partial coverage partitions.

Parameters **clustering** – NodeClustering object

Returns MatchingResult instance

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> leiden_communities = algorithms.leiden(g)
>>> mod = communities.nf1(leiden_communities)
```

Reference

1. Rossetti, G., Pappalardo, L., & Rinzivillo, S. (2016). **A novel approach to evaluate algorithms detection internal on ground truth.**
2. Rossetti, G. (2017). : **RDyn: graph benchmark handling algorithms dynamics. Journal of Complex Networks.** 5(6), 893-912.

normalized_cut (***kwargs*)

Normalized variant of the Cut-Ratio

$$: f(S) = \frac{c_S}{2m_S + c_S} + \frac{c_S}{2(mm_S) + c_S}$$

where m is the number of graph edges, m_S is the number of algorithms internal edges and c_S is the number of algorithms nodes.

Parameters **summary** – (optional, default True) if **True**, an overall summary is returned for the partition (min, max, avg, std); if **False** a list of community-wise score

Returns a FitnessResult object/a list of community-wise score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.normalized_cut()
```

normalized_mutual_information (*clustering*)

Normalized Mutual Information between two clusterings.

Normalized Mutual Information (NMI) is an normalization of the Mutual Information (MI) score to scale the results between 0 (no mutual information) and 1 (perfect correlation). In this function, mutual information is normalized by `sqrt(H(labels_true) * H(labels_pred))`

Parameters **clustering** – NodeClustering object

Returns normalized mutual information score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> leiden_communities = algorithms.leiden(g)
>>> mod = communities.normalized_mutual_information(leiden_communities)
```

omega (*clustering*)

Index of resemblance for overlapping, complete coverage, network clusterings.

Parameters **clustering** – NodeClustering object

Returns omega index

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> leiden_communities = algorithms.leiden(g)
>>> mod = communities.omega(leiden_communities)
```

Reference

1. Gabriel Murray, Giuseppe Carenini, and Raymond Ng. 2012. **Using the omega index for evaluating abstractive algorithms detection.** In Proceedings of Workshop on Evaluation Metrics and System Comparison for Automatic Summarization. Association for Computational Linguistics, Stroudsburg, PA, USA, 10-18.

overlapping_normalized_mutual_information_LFK(*clustering*)

Overlapping Normalized Mutual Information between two clusterings.

Extension of the Normalized Mutual Information (NMI) score to cope with overlapping partitions. This is the version proposed by Lancichinetti et al.

Parameters **clustering** – NodeClustering object

Returns onmi score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> leiden_communities = algorithms.leiden(g)
>>> mod = communities.overlapping_normalized_mutual_information_LFK(leiden_
↪communities)
```

Reference

1. Lancichinetti, A., Fortunato, S., & Kertesz, J. (2009). Detecting the overlapping and hierarchical community structure in complex networks. New Journal of Physics, 11(3), 033015.

overlapping_normalized_mutual_information_MGH(*clustering*, *normalization='max'*)

Overlapping Normalized Mutual Information between two clusterings.

Extension of the Normalized Mutual Information (NMI) score to cope with overlapping partitions. This is the version proposed by McDaid et al. using a different normalization than the original LFR one. See ref. for more details.

Parameters

- **clustering** – NodeClustering object
- **normalization** – one of "max" or "LFK". Default "max" (corresponds to the main method described in the article)

Returns onmi score

Example

```
>>> from cdlib import evaluation, algorithms
>>> g = nx.karate_club_graph()
>>> louvain_communities = algorithms.louvain(g)
>>> leiden_communities = algorithms.leiden(g)
>>> evaluation.overlapping_normalized_mutual_information_MGH(louvain_
↪communities,leiden_communities)
:Reference:
```

1. McDaid, A. F., Greene, D., & Hurley, N. (2011). Normalized mutual information to evaluate overlapping community finding algorithms. arXiv preprint arXiv:1110.2515. Chicago

purity()
Purity is the product of the frequencies of the most frequent labels carried by the nodes within the communities :return: FitnessResult object

significance()
Significance estimates how likely a partition of dense communities appear in a random graph.

Returns the significance score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.significance()
```

References

Traag, V. A., Aldecoa, R., & Delvenne, J. C. (2015). **Detecting communities using asymptotical surprise.** Physical Review E, 92(2), 022816.

size(***kwargs*)
Size is the number of nodes in the community

Parameters **summary** – (optional, default True) if **True**, an overall summary is returned for the partition (min, max, avg, std); if **False** a list of community-wise score

Returns a FitnessResult object/a list of community-wise score

Example:

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.size()
```

surprise()
Surprise is statistical approach proposes a quality metric assuming that edges between vertices emerge randomly according to a hyper-geometric distribution.

According to the Surprise metric, the higher the score of a partition, the less likely it is resulted from a random realization, the better the quality of the algorithms structure.

Returns the surprise score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.surprise()
```

References

Traag, V. A., Aldecoa, R., & Delvenne, J. C. (2015). **Detecting communities using asymptotical surprise.** Physical Review E, 92(2), 022816.

to_json()
Generate a JSON representation of the algorithms object

Returns a JSON formatted string representing the object

to_node_community_map()
Generate a <node, list(communities)> representation of the current clustering

Returns dict of the form <node, list(communities)>

triangle_participation_ratio(***kwargs*)
Fraction of algorithms nodes that belong to a triad.

$$f(S) = \frac{|\{u : u \in S, \{(v, w) : v, w \in S, (u, v) \in E, (u, w) \in E, (v, w) \in E\} \neq \emptyset\}|}{n_S}$$

where n_S is the set of algorithms nodes.

Parameters **summary** – (optional, default True) if **True**, an overall summary is returned for the partition (min, max, avg, std); if **False** a list of community-wise score

Returns a FitnessResult object/a list of community-wise score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.triangle_participation_ratio()
```

variation_of_information(*clustering*)
Variation of Information among two nodes partitions.

$$ H(p)+H(q)-2MI(p, q) $$

where MI is the mutual information, H the partition entropy and p,q are the algorithms sets

Parameters **clustering** – NodeClustering object

Returns VI score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> leiden_communities = algorithms.leiden(g)
>>> mod = communities.variation_of_information(leiden_communities)
```

Reference

1. Meila, M. (2007). **Comparing clusterings - an information based distance.** Journal of Multivariate Analysis, 98, 873-895. doi:10.1016/j.jmva.2006.11.013

z_modularity()

Z-modularity is another variant of the standard modularity proposed to avoid the resolution limit. The concept of this version is based on an observation that the difference between the fraction of edges inside communities and the expected number of such edges in a null model should not be considered as the only contribution to the final quality of algorithms structure.

Returns the z-modularity score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.z_modularity()
```

References

Miyauchi, Atsushi, and Yasushi Kawase. **Z-score-based modularity for algorithms detection in networks.** PloS one 11.1 (2016): e0147805.

Methods

Evaluating Node Clustering

AttrNodeClustering.purity()	Purity is the product of the frequencies of the most frequent labels carried by the nodes within the communities :return: FitnessResult object

Biparite Node Clustering

Overview

class BiNodeClustering(*left_communities*, *right_communities*, *graph*, *method_name*, *method_parameters=None*, *overlap=False*)

Bipartite Node Communities representation.

Parameters

- **left_communities** – list of left communities
- **right_communities** – list of right communities
- **graph** – a networkx/igraph object
- **method_name** – community discovery algorithm name
- **method_parameters** – configuration for the community discovery algorithm used
- **overlap** – boolean, whether the partition is overlapping or not

adjusted_mutual_information(*clustering*)

Adjusted Mutual Information between two clusterings.

Adjusted Mutual Information (AMI) is an adjustment of the Mutual Information (MI) score to account for chance. It accounts for the fact that the MI is generally higher for two clusterings with a larger number of clusters, regardless of whether there is actually more information shared. For two clusterings U and V, the AMI is given as:

```
AMI(U, V) = [MI(U, V) - E(MI(U, V))] / [max(H(U), H(V)) - E(MI(U, V))]
```

This metric is independent of the absolute values of the labels: a permutation of the class or cluster label values won't change the score value in any way.

This metric is furthermore symmetric: switching `label_true` with `label_pred` will return the same score value. This can be useful to measure the agreement of two independent label assignments strategies on the same dataset when the real ground truth is not known.

Be mindful that this function is an order of magnitude slower than other metrics, such as the Adjusted Rand Index.

Parameters **clustering** – NodeClustering object

Returns AMI score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> leiden_communities = algorithms.leiden(g)
>>> mod = communities.adjusted_mutual_information(leiden_communities)
```

Reference

1. Vinh, N. X., Epps, J., & Bailey, J. (2010). **Information theoretic measures for clusterings comparison: Variants, properties, normalization and correction for chance.** Journal of Machine Learning Research, 11(Oct), 2837-2854.

adjusted_rand_index(*clustering*)

Rand index adjusted for chance.

The Rand Index computes a similarity measure between two clusterings by considering all pairs of samples and counting pairs that are assigned in the same or different clusters in the predicted and true clusterings.

The raw RI score is then "adjusted for chance" into the ARI score using the following scheme:

```
ARI = (RI - Expected_RI) / (max(RI) - Expected_RI)
```

The adjusted Rand index is thus ensured to have a value close to 0.0 for random labeling independently of the number of clusters and samples and exactly 1.0 when the clusterings are identical (up to a permutation).

ARI is a symmetric measure:

```
adjusted_rand_index(a, b) == adjusted_rand_index(b, a)
```

Parameters **clustering** – NodeClustering object

Returns ARI score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> leiden_communities = algorithms.leiden(g)
>>> mod = communities.adjusted_rand_index(leiden_communities)
```

Reference

1. Hubert, L., & Arabie, P. (1985). **Comparing partitions**. Journal of classification, 2(1), 193-218.

average_internal_degree (***kwargs*)

The average internal degree of the algorithms set.

$$f(S) = \frac{2m_S}{n_S}$$

where m_S is the number of algorithms internal edges and n_S is the number of algorithms nodes.

Parameters **summary** – (optional, default True) if **True**, an overall summary is returned for the partition (min, max, avg, std); if **False** a list of community-wise score

Returns a FitnessResult object/a list of community-wise score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.average_internal_degree()
```

avg_odf (***kwargs*)

Average fraction of edges of a node of a algorithms that point outside the algorithms itself.

$$\frac{1}{n_S} \sum_{u \in S} \frac{|\{(u,v) \in E : v \notin S\}|}{d(u)}$$

where E is the graph edge set, v is a node in S, $d(u)$ is the degree of u and n_S is the set of algorithms nodes.

Parameters **summary** – (optional, default True) if **True**, an overall summary is returned for the partition (min, max, avg, std); if **False** a list of community-wise score

Returns a FitnessResult object/a list of community-wise score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>>
>>> communities = eva(g, alpha=alpha)
>>> pur = communities.purity()
```

conductance (***kwargs*)

Fraction of total edge volume that points outside the algorithms.

$$f(S) = \frac{c_S}{2m_S + c_S}$$

where c_S is the number of algorithms nodes and, m_S is the number of algorithms edges

Parameters **summary** – (optional, default True) if **True**, an overall summary is returned for the partition (min, max, avg, std); if **False** a list of community-wise score

Returns a FitnessResult object/a list of community-wise score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.conductance()
```

cut_ratio(***kwargs*)

Fraction of existing edges (out of all possible edges) leaving the algorithms.

..math:: f(S) = frac{c_S}{n_S (n n_S)}

where c_S is the number of algorithms nodes and, n_S is the number of edges on the algorithms boundary

Parameters **summary** – (optional, default True) if **True**, an overall summary is returned for the partition (min, max, avg, std); if **False** a list of community-wise score

Returns a FitnessResult object/a list of community-wise score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.cut_ratio()
```

edges_inside(***kwargs*)

Number of edges internal to the algorithms.

Parameters **summary** – (optional, default True) if **True**, an overall summary is returned for the partition (min, max, avg, std); if **False** a list of community-wise score

Returns a FitnessResult object/a list of community-wise score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.edges_inside()
```

erdos_renyi_modularity()

Erdos-Renyi modularity is a variation of the Newman-Girvan one. It assumes that vertices in a network are connected randomly with a constant probability p.

$$Q(S) = \frac{1}{m}\sum_{c \in S}(m_S \frac{m n_S(n_S 1)}{n(n1)})$$

where m is the number of graph edges, m_S is the number of algorithms edges, l_S is the number of edges from nodes in S to nodes outside S.

Returns the Erdos-Renyi modularity score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.erdos_renyi_modularity()
```

References

Erdos, P., & Renyi, A. (1959). **On random graphs I.** Publ. Math. Debrecen, 6, 290-297.

expansion (***kwargs*)

Number of edges per algorithms node that point outside the cluster.

$$f(S) = \frac{c_S}{n_S}$$

where n_S is the number of edges on the algorithms boundary, c_S is the number of algorithms nodes.

Parameters **summary** – (optional, default True) if **True**, an overall summary is returned for the partition (min, max, avg, std); if **False** a list of community-wise score

Returns a FitnessResult object/a list of community-wise score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.expansion()
```

f1 (*clustering*)

Compute the average F1 score of the optimal algorithms matches among the partitions in input. Works on overlapping/non-overlapping complete/partial coverage partitions.

Parameters **clustering** – NodeClustering object

Returns F1 score (harmonic mean of precision and recall)

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> leiden_communities = algorithms.leiden(g)
>>> mod = communities.f1(leiden_communities)
```

Reference

1. Rossetti, G., Pappalardo, L., & Rinzivillo, S. (2016). **A novel approach to evaluate algorithms detection internal on ground truth.** In Complex Networks VII (pp. 133-144). Springer, Cham.

flake_odf (***kwargs*)

Fraction of nodes in S that have fewer edges pointing inside than to the outside of the algorithms.

$$f(S) = \frac{|\{u : u \in S, |\{(u, v) \in E : v \in S\}| < d(u)/2\}|}{n_S}$$

where E is the graph edge set, v is a node in S, $d(u)$ is the degree of u and n_S is the set of algorithms nodes.

Parameters **summary** – (optional, default True) if **True**, an overall summary is returned for the partition (min, max, avg, std); if **False** a list of community-wise score

Returns a FitnessResult object/a list of community-wise score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.flake_odf()
```

fraction_over_median_degree (***kwargs*)
Fraction of algorithms nodes of having internal degree higher than the median degree value.

$$f(S) = \frac{|\{u : u \in S, |\{(u, v) : v \in S\}| > d_m\}|}{n_S}$$

where d_m is the internal degree median value

Parameters **summary** – (optional, default True) if **True**, an overall summary is returned for the partition (min, max, avg, std); if **False** a list of community-wise score

Returns a FitnessResult object/a list of community-wise score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.fraction_over_median_degree()
```

get_description (*parameters_to_display=None*, *precision=3*)
Return a description of the clustering, with the name of the method and its numeric parameters.

Parameters

- **parameters_to_display** – parameters to display. By default, all float parameters.
- **precision** – precision used to plot parameters. default: 3

Returns a string description of the method.

internal_edge_density (***kwargs*)
The internal density of the algorithms set.

$$f(S) = \frac{m_S}{n_S(n_S 1)/2}$$

where m_S is the number of algorithms internal edges and n_S is the number of algorithms nodes.

Parameters **summary** – (optional, default True) if **True**, an overall summary is returned for the partition (min, max, avg, std); if **False** a list of community-wise score

Returns a FitnessResult object/a list of community-wise score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.internal_edge_density()
```

link_modularity()

Quality function designed for directed graphs with overlapping communities.

Returns the link modularity score

Example

```
>>> from cdlib import evaluation
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.link_modularity()
```

max_odf(***kwargs*)

Maximum fraction of edges of a node of a algorithms that point outside the algorithms itself.

$$max_{u \in S} \frac{|\{(u,v) \in E : v \notin S\}|}{d(u)}$$

where E is the graph edge set, v is a node in S and $d(u)$ is the degree of u

Parameters **summary** – (optional, default True) if **True**, an overall summary is returned for the partition (min, max, avg, std); if **False** a list of community-wise score

Returns a FitnessResult object/a list of community-wise score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.max_odf()
```

modularity_density()

The modularity density is one of several propositions that envisioned to palliate the resolution limit issue of modularity based measures. The idea of this metric is to include the information about algorithms size into the expected density of algorithms to avoid the negligence of small and dense communities. For each algorithms C in partition S, it uses the average modularity degree calculated by $d(C) = d^{int(C)} d^{ext(C)}$ where $d^{int(C)}$ and $d^{ext(C)}$ are the average internal and external degrees of C respectively to evaluate the fitness of C in its network. Finally, the modularity density can be calculated as follows:

$$Q(S) = \sum_{C \in S} \frac{1}{n_C} (\sum_{i \in C} k_{iC}^{int} - \sum_{i \in C} k_{iC}^{out})$$

where n_C is the number of nodes in C, k_{iC}^{int} is the degree of node i within C and k_{iC}^{out} is the deree of node i outside C.

Returns the modularity density score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.modularity_density()
```

References

Li, Z., Zhang, S., Wang, R. S., Zhang, X. S., & Chen, L. (2008). **Quantitative function for algorithms detection.** Physical review E, 77(3), 036109.

newman_girvan_modularity()

Difference the fraction of intra algorithms edges of a partition with the expected number of such edges if distributed according to a null model.

In the standard version of modularity, the null model preserves the expected degree sequence of the graph under consideration. In other words, the modularity compares the real network structure with a corresponding one where nodes are connected without any preference about their neighbors.

$$Q(S) = \frac{1}{m}\sum_{c \in S}(m_S - \frac{(2m_S + l_S)^2}{4m})$$

where m is the number of graph edges, m_S is the number of algorithms edges, l_S is the number of edges from nodes in S to nodes outside S.

Returns the Newman-Girvan modularity score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.newman_girvan_modularity()
```

References

Newman, M.E.J. & Girvan, M. **Finding and evaluating algorithms structure in networks.** Physical Review E 69, 26113(2004).

nf1(*clustering*)

Compute the Normalized F1 score of the optimal algorithms matches among the partitions in input. Works on overlapping/non-overlapping complete/partial coverage partitions.

Parameters **clustering** – NodeClustering object

Returns MatchingResult instance

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> leiden_communities = algorithms.leiden(g)
>>> mod = communities.nf1(leiden_communities)
```

Reference

1. Rossetti, G., Pappalardo, L., & Rinzivillo, S. (2016). **A novel approach to evaluate algorithms detection internal on ground truth.**
2. Rossetti, G. (2017). : **RDyn: graph benchmark handling algorithms dynamics. Journal of Complex Networks.** 5(6), 893-912.

normalized_cut(***kwargs*)

Normalized variant of the Cut-Ratio

$$: f(S) = \frac{c_S}{2m_S + c_S} + \frac{c_S}{2(mm_S) + c_S}$$

where m is the number of graph edges, m_S is the number of algorithms internal edges and c_S is the number of algorithms nodes.

Parameters **summary** – (optional, default True) if **True**, an overall summary is returned for the partition (min, max, avg, std); if **False** a list of community-wise score

Returns a FitnessResult object/a list of community-wise score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.normalized_cut()
```

normalized_mutual_information(*clustering*)

Normalized Mutual Information between two clusterings.

Normalized Mutual Information (NMI) is an normalization of the Mutual Information (MI) score to scale the results between 0 (no mutual information) and 1 (perfect correlation). In this function, mutual information is normalized by `sqrt(H(labels_true) * H(labels_pred))`

Parameters **clustering** – NodeClustering object

Returns normalized mutual information score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> leiden_communities = algorithms.leiden(g)
>>> mod = communities.normalized_mutual_information(leiden_communities)
```

omega(*clustering*)

Index of resemblance for overlapping, complete coverage, network clusterings.

Parameters **clustering** – NodeClustering object

Returns omega index

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> leiden_communities = algorithms.leiden(g)
>>> mod = communities.omega(leiden_communities)
```

Reference

1. Gabriel Murray, Giuseppe Carenini, and Raymond Ng. 2012. **Using the omega index for evaluating abstractive algorithms detection.** In Proceedings of Workshop on Evaluation Metrics and System Comparison for Automatic Summarization. Association for Computational Linguistics, Stroudsburg, PA, USA, 10-18.

overlapping_normalized_mutual_information_LFK(*clustering*)

Overlapping Normalized Mutual Information between two clusterings.

Extension of the Normalized Mutual Information (NMI) score to cope with overlapping partitions. This is the version proposed by Lancichinetti et al.

Parameters **clustering** – NodeClustering object

Returns onmi score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> leiden_communities = algorithms.leiden(g)
>>> mod = communities.overlapping_normalized_mutual_information_LFK(leiden_
↪communities)
```

Reference

1. Lancichinetti, A., Fortunato, S., & Kertesz, J. (2009). Detecting the overlapping and hierarchical community structure in complex networks. New Journal of Physics, 11(3), 033015.

overlapping_normalized_mutual_information_MGH(*clustering*, *normalization='max'*)

Overlapping Normalized Mutual Information between two clusterings.

Extension of the Normalized Mutual Information (NMI) score to cope with overlapping partitions. This is the version proposed by McDaid et al. using a different normalization than the original LFR one. See ref. for more details.

Parameters

- **clustering** – NodeClustering object
- **normalization** – one of "max" or "LFK". Default "max" (corresponds to the main method described in the article)

Returns onmi score

Example

```
>>> from cdlib import evaluation, algorithms
>>> g = nx.karate_club_graph()
>>> louvain_communities = algorithms.louvain(g)
>>> leiden_communities = algorithms.leiden(g)
>>> evaluation.overlapping_normalized_mutual_information_MGH(louvain_
↪communities,leiden_communities)
:Reference:
```

1. McDaid, A. F., Greene, D., & Hurley, N. (2011). Normalized mutual information to evaluate overlapping community finding algorithms. arXiv preprint arXiv:1110.2515. Chicago

significance()

Significance estimates how likely a partition of dense communities appear in a random graph.

Returns the significance score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.significance()
```

References

Traag, V. A., Aldecoa, R., & Delvenne, J. C. (2015). **Detecting communities using asymptotical surprise.** Physical Review E, 92(2), 022816.

size(***kwargs*)

Size is the number of nodes in the community

Parameters **summary** – (optional, default True) if **True**, an overall summary is returned for the partition (min, max, avg, std); if **False** a list of community-wise score

Returns a FitnessResult object/a list of community-wise score

Example:

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.size()
```

surprise()

Surprise is statistical approach proposes a quality metric assuming that edges between vertices emerge randomly according to a hyper-geometric distribution.

According to the Surprise metric, the higher the score of a partition, the less likely it is resulted from a random realization, the better the quality of the algorithms structure.

Returns the surprise score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.surprise()
```

References

Traag, V. A., Aldecoa, R., & Delvenne, J. C. (2015). **Detecting communities using asymptotical surprise.** Physical Review E, 92(2), 022816.

to_json()

Generate a JSON representation of the algorithms object

Returns a JSON formatted string representing the object

to_node_community_map()

Generate a <node, list(communities)> representation of the current clustering

Returns dict of the form <node, list(communities)>

triangle_participation_ratio(***kwargs*)

Fraction of algorithms nodes that belong to a triad.

$$f(S) = \frac{|\{u : u \in S, \{(v, w) : v, w \in S, (u, v) \in E, (u, w) \in E, (v, w) \in E\} \neq \emptyset\}|}{n_S}$$

where n_S is the set of algorithms nodes.

Parameters **summary** – (optional, default True) if **True**, an overall summary is returned for the partition (min, max, avg, std); if **False** a list of community-wise score

Returns a FitnessResult object/a list of community-wise score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.triangle_participation_ratio()
```

variation_of_information(*clustering*)

Variation of Information among two nodes partitions.

$$ H(p)+H(q)-2MI(p, q) $$

where MI is the mutual information, H the partition entropy and p,q are the algorithms sets

Parameters **clustering** – NodeClustering object

Returns VI score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> leiden_communities = algorithms.leiden(g)
>>> mod = communities.variation_of_information(leiden_communities)
```

Reference

1. Meila, M. (2007). **Comparing clusterings - an information based distance.** Journal of Multivariate Analysis, 98, 873-895. doi:10.1016/j.jmva.2006.11.013

z_modularity()

Z-modularity is another variant of the standard modularity proposed to avoid the resolution limit. The concept of this version is based on an observation that the difference between the fraction of edges inside communities and the expected number of such edges in a null model should not be considered as the only contribution to the final quality of algorithms structure.

Returns the z-modularity score

Example

```
>>> from cdlib.algorithms import louvain
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = communities.z_modularity()
```

References

Miyauchi, Atsushi, and Yasushi Kawase. **Z-score-based modularity for algorithms detection in networks.** PloS one 11.1 (2016): e0147805.

Edge Clustering

Overview

class EdgeClustering(*communities*, *graph*, *method_name*, *method_parameters=None*, *overlap=False*)

Edge Clustering representation.

Parameters

- **communities** – list of communities
- **graph** – a networkx/igraph object
- **method_name** – community discovery algorithm name
- **method_parameters** – configuration for the community discovery algorithm used
- **overlap** – boolean, whether the partition is overlapping or not

get_description(*parameters_to_display=None*, *precision=3*)
Return a description of the clustering, with the name of the method and its numeric parameters.

Parameters

- **parameters_to_display** – parameters to display. By default, all float parameters.
- **precision** – precision used to plot parameters. default: 3

Returns a string description of the method.

to_edge_community_map()
Generate a <edge, list(communities)> representation of the current clustering

Returns dict of the form <edge, list(communities)>

to_json()
Generate a JSON representation of the algorithms object

Returns a JSON formatted string representing the object

Methods

Data transformation and IO

EdgeClustering.to_json()	Generate a JSON representation of the algorithms object
EdgeClustering.to_edge_community_map()	Generate a <edge, list(communities)> representation of the current clustering

1.5.2 Community Discovery algorithms

CDlib collects implementations of several Community Discovery algorithms.

To maintain the library organization as clean and resilient as possible the approaches are grouped following a simple, two level, rationale:

1. The first distinction is made on the object clustered, thus separating **Node Clustering** and **Edge Clustering** algorithms;
2. The second distinction is made on the specific kind of partition each one of them generates: **Crisp**, **Overlapping** or **Fuzzy**.

This documentation follows the same rationale.

Node Clustering

Algorithms falling in this category generate communities composed by nodes. The communities can represent neat, *crisp*, partition as well as *overlapping* or even *fuzzy* ones.

Crisp Communities

A clustering is said to be a *partition* if each node belongs to one and only one community. Methods in this subclass return as result a `NodeClustering` object instance.

agdl(g_original, number_communities, ...)	AGDL is a graph-based agglomerative algorithm, for clustering high-dimensional data.
aslpaw(g_original)	ASLPAw can be used for disjoint and overlapping community detection and works on weighted/unweighted and directed/undirected networks.
async_fluid(g_original, k)	Fluid Communities (FluidC) is based on the simple idea of fluids (i.e., communities) interacting in an environment (i.e., a non-complete graph), expanding and contracting.
cpm(g_original[, initial_membership, ...])	CPM is a model where the quality function to optimize is:
chinesewhispers(g_original[, weighting, ...])	Fuzzy graph clustering that (i) creates an intermediate representation of the input graph, which reflects the "ambiguity" of its nodes, and (ii) uses hard clustering to discover crisp clusters in such "disambiguated" intermediate graph.
der(g_original[, walk_len, threshold, ...])	DER is a Diffusion Entropy Reducer graph clustering algorithm.
edmot(g_original[, component_count, cutoff])	The algorithm first creates the graph of higher order motifs.
eigenvector(g_original)	Newman's leading eigenvector method for detecting community structure based on modularity.
em(g_original, k)	EM is based on based on a mixture model.
gdmp2(g_original[, min_threshold])	Gdmp2 is a method for identifying a set of dense subgraphs of a given sparse graph.
girvan_newman(g_original, level)	The Girvan–Newman algorithm detects communities by progressively removing edges from the original graph.
greedy_modularity(g_original[, weight])	The CNM algorithm uses the modularity to find the communities strcutures.
infomap(g_original)	Infomap is based on ideas of information theory.
label_propagation(g_original)	The Label Propagation algorithm (LPA) detects communities using network structure alone.
leiden(g_original[, initial_membership, weights])	The Leiden algorithm is an improvement of the Louvain algorithm.
louvain(g_original[, weight, resolution, ...])	Louvain maximizes a modularity score for each community.
markov_clustering(g_original[, expansion, ...])	The Markov clustering algorithm (MCL) is based on simulation of (stochastic) flow in graphs.
rber_pots(g_original[, initial_membership, ...])	rber_pots is a model where the quality function to optimize is:
rb_pots(g_original[, initial_membership, ...])	Rb_pots is a model where the quality function to optimize is:
scan(g_original, epsilon, mu)	SCAN (Structural Clustering Algorithm for Networks) is an algorithm which detects clusters, hubs and outliers in networks.
significance_communities(g_original[, ...])	Significance_communities is a model where the quality function to optimize is:

Continued on next page

Table 8 – continued from previous page

spinglass(g_original)	Spinglass relies on an analogy between a very popular statistical mechanic model called Potts spin glass, and the community structure.
surprise_communities(g_original[, ...])	Surprise_communities is a model where the quality function to optimize is:
walktrap(g_original)	walktrap is an approach based on random walks.
sbm_dl(g_original[, B_min, B_max, deg_corr])	Efficient Monte Carlo and greedy heuristic for the inference of stochastic block models.
sbm_dl_nested(g_original[, B_min, B_max, ...])	Efficient Monte Carlo and greedy heuristic for the inference of stochastic block models.

cdlib.algorithms.agdl

agdl(*g_original*, *number_communities*, *number_neighbors*, *kc*, *a*)

AGDL is a graph-based agglomerative algorithm, for clustering high-dimensional data. The algorithm uses the indegree and outdegree to characterize the affinity between two clusters.

Parameters

- **g_original** – a networkx/igraph object
- **number_communities** – number of communities
- **number_neighbors** – Number of neighbors to use for KNN
- **kc** – size of the neighbor set for each cluster
- **a** – range(-infinity;+infinty). From the authors: a=np.arange(-2,2.1,0.5)

Returns

NodeClustering object

Example

```
>>> from cdlib import algorithms
>>> import networkx as nx
>>> G = nx.karate_club_graph()
>>> com = algorithms.agdl(g, number_communities=3, number_neighbors=3, kc=4, a=1)
```

References

Zhang, W., Wang, X., Zhao, D., & Tang, X. (2012, October). Graph degree linkage: Agglomerative clustering on a directed graph. In European Conference on Computer Vision (pp. 428-441). Springer, Berlin, Heidelberg.

Note: Reference implementation: https://github.com/myungjoon/GDL

cdlib.algorithms.aslpaw

aslpaw(*g_original*)

ASLPAw can be used for disjoint and overlapping community detection and works on weighted/unweighted and directed/undirected networks. ASLPAw is adaptive with virtually no configuration parameters.

Parameters **g_original** – a networkx/igraph object

Returns NodeClustering object

Example

```
>>> from cdlib import algorithms
>>> import networkx as nx
>>> G = nx.karate_club_graph()
>>> coms = algorithms.aslpaw(G)
```

References

Xie J, Szymanski B K, Liu X. Slpa: Uncovering Overlapping Communities in Social Networks via a Speaker-Listener Interaction Dynamic Process[C]. IEEE 11th International Conference on Data Mining Workshops (ICDMW). Ancouver, BC: IEEE, 2011: 344–349.

Note: Reference implementation: https://github.com/fsssosei/ASLPAw

cdlib.algorithms.async_fluid

async_fluid(*g_original*, *k*)

Fluid Communities (FluidC) is based on the simple idea of fluids (i.e., communities) interacting in an environment (i.e., a non-complete graph), expanding and contracting. It is propagation-based algorithm and it allows to specify the number of desired communities (k) and it is asynchronous, where each vertex update is computed using the latest partial state of the graph.

Parameters

- **g_original** – a networkx/igraph object
- **k** – Number of communities to search

Returns EdgeClustering object

Example

```
>>> from cdlib import algorithms
>>> import networkx as nx
>>> G = nx.karate_club_graph()
>>> coms = algorithms.async_fluid(G,k=2)
```

References

Ferran Parés, Dario Garcia-Gasulla, Armand Vilalta, Jonatan Moreno, Eduard Ayguadé, Jesús Labarta, Ulises Cortés, Toyotaro Suzumura T. Fluid Communities: A Competitive and Highly Scalable Community Detection Algorithm.

cdlib.algorithms.cpm

cpm(*g_original*, *initial_membership=None*, *weights=None*, *node_sizes=None*, *resolution_parameter=1*)

CPM is a model where the quality function to optimize is:

$$Q = \sum_{ij} (A_{ij} - \gamma)\, \delta(\sigma_i, \sigma_j)$$

where A is the adjacency matrix, σ_i denotes the community of node i, $\delta(\sigma_i, \sigma_j) = 1$ if $\sigma_i = \sigma_j$ and *0* otherwise, and, finally γ is a resolution parameter.

The internal density of communities

$$p_c = \frac{m_c}{\binom{n_c}{2}} \geq \gamma$$

is higher than γ, while the external density

$p_{cd} = \frac{m_{cd}}{n_c n_d} \leq \gamma$ is lower than γ. In other words, choosing a particular γ corresponds to choosing to find communities of a particular density, and as such defines communities. Finally, the definition of a community is in a sense independent of the actual graph, which is not the case for any of the other methods.

Parameters

- **g_original** – a networkx/igraph object
- **initial_membership** – list of int Initial membership for the partition. If `None` then defaults to a singleton partition. Deafault None
- **weights** – list of double, or edge attribute Weights of edges. Can be either an iterable or an edge attribute. Deafault None
- **node_sizes** – list of int, or vertex attribute Sizes of nodes are necessary to know the size of communities in aggregate graphs. Usually this is set to 1 for all nodes, but in specific cases this could be changed. Deafault None
- **resolution_parameter** – double >0 A parameter value controlling the coarseness of the clustering. Higher resolutions lead to more communities, while lower resolutions lead to fewer communities. Deafault 1

Returns NodeClustering object

Example

```
>>> from cdlib import algorithms
>>> import networkx as nx
>>> G = nx.karate_club_graph()
>>> coms = algorithms.cpm(G)
```

References

Traag, V. A., Van Dooren, P., & Nesterov, Y. (2011). Narrow scope for resolution-limit-free community detection. Physical Review E, 84(1), 016114. 10.1103/PhysRevE.84.016114

Note: Reference implementation: https://github.com/vtraag/leidenalg

---

cdlib.algorithms.chinesewhispers

chinesewhispers(*g_original*, *weighting='top'*, *iterations=20*, *seed=None*)

Fuzzy graph clustering that (i) creates an intermediate representation of the input graph, which reflects the "ambiguity" of its nodes, and (ii) uses hard clustering to discover crisp clusters in such "disambiguated" intermediate graph.

Parameters

- **g_original** –

- **weighting** – edge weighing schemas. Available modalities: [‘top’, ‘lin’, ‘log’]
- **iterations** – number of iterations
- **seed** – random seed

Returns NodeClustering object

Example

```
>>> from cdlib import algorithms
>>> import networkx as nx
>>> G = nx.karate_club_graph()
>>> coms = algorithms.chinesewhispers(G)
```

References

Ustalov, D., Panchenko, A., Biemann, C., Ponzetto, S.P.: **‘Watset: Local-Global Graph Clustering with Applications in Sense and Frame Induction.‘_** Computational Linguistics 45(3), 423–479 (2019)

Note: Reference implementation: https://github.com/nlpub/chinese-whispers-python

cdlib.algorithms.der

der (*g_original*, *walk_len=3*, *threshold=1e-05*, *iter_bound=50*)

DER is a Diffusion Entropy Reducer graph clustering algorithm. The algorithm uses random walks to embed the graph in a space of measures, after which a modification of k-means in that space is applied. It creates the walks, creates an initialization, runs the algorithm, and finally extracts the communities.

Parameters

- **g_original** – an undirected networkx graph object
- **walk_len** – length of the random walk, default 3
- **threshold** – threshold for stop criteria; if the likelihood_diff is less than threshold tha algorithm stops, default 0.00001
- **iter_bound** – maximum number of iteration, default 50

Returns NodeClustering object

Example

```
>>> from cdlib import algorithms
>>> import networkx as nx
>>> G = nx.karate_club_graph()
>>> coms = algorithms.der(G, 3, .00001, 50)
```

References

13. Kozdoba and S. Mannor, Community Detection via Measure Space Embedding, NIPS 2015

Note: Reference implementation: https://github.com/komarkdev/der_graph_clustering

cdlib.algorithms.edmot

edmot(*g_original*, *component_count=2*, *cutoff=10*)

The algorithm first creates the graph of higher order motifs. This graph is clustered by the Louvain method.

Parameters

- **g_original** – a networkx/igraph object
- **component_count** – Number of extracted motif hypergraph components. Default is 2.
- **cutoff** – Motif edge cut-off value. Default is 10.

Returns NodeClustering object

Example

```
>>> from cdlib import algorithms
>>> import networkx as nx
>>> G = nx.karate_club_graph()
>>> coms = algorithms.edmot(G, max_loop=1000)
```

References

Li, Pei-Zhen, et al. "EdMot: An Edge Enhancement Approach for Motif-aware Community Detection." Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. 2019.

Note: Reference implementation: https://karateclub.readthedocs.io/

---

cdlib.algorithms.eigenvector

eigenvector(*g_original*)

Newman's leading eigenvector method for detecting community structure based on modularity. This is the proper internal of the recursive, divisive algorithm: each split is done by maximizing the modularity regarding the original network.

Parameters **g_original** – a networkx/igraph object

Returns NodeClustering object

Example

```
>>> from cdlib import algorithms
>>> import networkx as nx
>>> G = nx.karate_club_graph()
>>> com = algorithms.eigenvector(G)
```

References

Newman, Mark EJ. Finding community structure in networks using the eigenvectors of matrices. Physical review E 74.3 (2006): 036104.

cdlib.algorithms.em

em(*g_original*, *k*)

EM is based on based on a mixture model. The algorithm uses the expectation–maximization algorithm to detect structure in networks.

Parameters

- **g_original** – a networkx/igraph object
- **k** – the number of desired communities

Returns NodeClustering object

Example

```
>>> from cdlib import algorithms
>>> import networkx as nx
>>> G = nx.karate_club_graph()
>>> com = algorithms.em(G, k=3)
```

References

Newman, Mark EJ, and Elizabeth A. Leicht. Mixture community and exploratory analysis in networks. Proceedings of the National Academy of Sciences 104.23 (2007): 9564-9569.

cdlib.algorithms.gdmp2

gdmp2(*g_original*, *min_threshold=0.75*)

Gdmp2 is a method for identifying a set of dense subgraphs of a given sparse graph. It is inspired by an effective technique designed for a similar problem—matrix blocking, from a different discipline (solving linear systems).

Parameters

- **g_original** – a networkx/igraph object
- **min_threshold** – the minimum density threshold parameter to control the density of the output subgraphs, default 0.75

Returns NodeClustering object

Example

```
>>> from cdlib import algorithms
>>> import networkx as nx
>>> G = nx.karate_club_graph()
>>> com = algorithms.gdmp2(G)
```

References

Chen, Jie, and Yousef Saad. Dense subgraph extraction with application to community detection. IEEE Transactions on Knowledge and Data Engineering 24.7 (2012): 1216-1230.

Note: Reference implementation: https://github.com/imabhishekl/CSC591_Community_Detection

cdlib.algorithms.girvan_newman

girvan_newman(*g_original*, *level*)

The Girvan–Newman algorithm detects communities by progressively removing edges from the original graph. The algorithm removes the "most valuable" edge, traditionally the edge with the highest betweenness centrality, at each step. As the graph breaks down into pieces, the tightly knit community structure is exposed and the result can be depicted as a dendrogram.

Parameters

- **g_original** – a networkx/igraph object
- **level** – the level where to cut the dendrogram

Returns NodeClustering object

Example

```
>>> from cdlib import algorithms
>>> import networkx as nx
>>> G = nx.karate_club_graph()
>>> com = algorithms.girvan_newman(G, level=3)
```

References

Girvan, Michelle, and Mark EJ Newman. Community structure in social and biological networks. Proceedings of the national academy of sciences 99.12 (2002): 7821-7826.

cdlib.algorithms.greedy_modularity

greedy_modularity(*g_original*, *weight=None*)

The CNM algorithm uses the modularity to find the communities strcutures. At every step of the algorithm two communities that contribute maximum positive value to global modularity are merged.

Parameters

- **g_original** – a networkx/igraph object
- **weight** – list of double, or edge attribute Weights of edges. Can be either an iterable or an edge attribute. Deafault None

Returns NodeClustering object

Example

```
>>> from cdlib import algorithms
>>> import networkx as nx
>>> G = nx.karate_club_graph()
>>> coms = algorithms.greedy_modularity(G)
```

References

Clauset, A., Newman, M. E., & Moore, C. Finding community structure in very large networks. Physical Review E 70(6), 2004

cdlib.algorithms.infomap

infomap(*g_original*)

Infomap is based on ideas of information theory. The algorithm uses the probability flow of random walks on a network as a proxy for information flows in the real system and it decomposes the network into modules by compressing a description of the probability flow.

Parameters **g_original** – a networkx/igraph object

Returns NodeClustering object

Example

```
>>> from cdlib import algorithms
>>> import networkx as nx
>>> G = nx.karate_club_graph()
>>> coms = algorithms.infomap(G)
```

References

Rosvall M, Bergstrom CT (2008) Maps of random walks on complex networks reveal community structure. Proc Natl Acad SciUSA 105(4):1118–1123

Note: Reference implementation: https://pypi.org/project/infomap/

cdlib.algorithms.label_propagation

label_propagation(*g_original*)

The Label Propagation algorithm (LPA) detects communities using network structure alone. The algorithm doesn't require a pre-defined objective function or prior information about the communities. It works as follows: -Every node is initialized with a unique label (an identifier) -These labels propagate through the network -At every iteration of propagation, each node updates its label to the one that the maximum numbers of its neighbours belongs to. Ties are broken uniformly and randomly. -LPA reaches convergence when each node has the majority label of its neighbours.

Parameters **g_original** – a networkx/igraph object

Returns EdgeClustering object

Example

```
>>> from cdlib import algorithms
>>> import networkx as nx
>>> G = nx.karate_club_graph()
>>> coms = algorithms.label_propagation(G)
```

References

Raghavan, U. N., Albert, R., & Kumara, S. (2007). Near linear time algorithm to detect community structures in large-scale networks. Physical review E, 76(3), 036106.

cdlib.algorithms.leiden

leiden(*g_original*, *initial_membership=None*, *weights=None*)

The Leiden algorithm is an improvement of the Louvain algorithm. The Leiden algorithm consists of three phases: (1) local moving of nodes, (2) refinement of the partition (3) aggregation of the network based on the refined partition, using the non-refined partition to create an initial partition for the aggregate network.

Parameters

- **g_original** – a networkx/igraph object
- **initial_membership** – list of int Initial membership for the partition. If `None` then defaults to a singleton partition. Deafault None
- **weights** – list of double, or edge attribute Weights of edges. Can be either an iterable or an edge attribute. Deafault None

Returns NodeClustering object

Example

```
>>> from cdlib import algorithms
>>> import networkx as nx
>>> G = nx.karate_club_graph()
>>> coms = algorithms.leiden(G)
```

References

Traag, Vincent, Ludo Waltman, and Nees Jan van Eck. From Louvain to Leiden: guaranteeing well-connected communities. arXiv preprint arXiv:1810.08473 (2018).

Note: Reference implementation: https://github.com/vtraag/leidenalg

---

cdlib.algorithms.louvain

louvain(*g_original*, *weight='weight'*, *resolution=1.0*, *randomize=False*)

Louvain maximizes a modularity score for each community. The algorithm optimises the modularity in two elementary phases: (1) local moving of nodes; (2) aggregation of the network. In the local moving phase, individual nodes are moved to the community that yields the largest increase in the quality function. In the aggregation phase, an aggregate network is created based on the partition obtained in the local moving phase. Each community in this partition becomes a node in the aggregate network. The two phases are repeated until the quality function cannot be increased further.

Parameters

- **g_original** – a networkx/igraph object
- **weight** – str, optional the key in graph to use as weight. Default to 'weight'
- **resolution** – double, optional Will change the size of the communities, default to 1.
- **randomize** – boolean, optional Will randomize the node evaluation order and the community evaluation order to get different partitions at each call, default False

Returns NodeClustering object

Example

```
>>> from cdlib import algorithms
>>> import networkx as nx
>>> G = nx.karate_club_graph()
>>> coms = algorithms.louvain(G, weight='weight', resolution=1., randomize=False)
```

References

Blondel, Vincent D., et al. Fast unfolding of communities in large networks. Journal of statistical mechanics: theory and experiment 2008.10 (2008): P10008.

Note: Reference implementation: https://github.com/taynaud/python-louvain

---

cdlib.algorithms.markov_clustering

markov_clustering(*g_original*, *expansion=2*, *inflation=2*, *loop_value=1*, *iterations=100*, *pruning_threshold=0.001*, *pruning_frequency=1*, *convergence_check_frequency=1*)

The Markov clustering algorithm (MCL) is based on simulation of (stochastic) flow in graphs. The MCL algorithm finds cluster structure in graphs by a mathematical bootstrapping procedure. The process deterministically computes (the probabilities of) random walks through the graph, and uses two operators transforming one set of probabilities into another. It does so using the language of stochastic matrices (also called Markov matrices) which capture the mathematical concept of random walks on a graph. The MCL algorithm simulates random walks within a graph by alternation of two operators called expansion and inflation.

Parameters

- **g_original** – a networkx/igraph object
- **expansion** – The cluster expansion factor
- **inflation** – The cluster inflation factor
- **loop_value** – Initialization value for self-loops
- **iterations** – Maximum number of iterations (actual number of iterations will be less if convergence is reached)
- **pruning_threshold** – Threshold below which matrix elements will be set set to 0
- **pruning_frequency** – Perform pruning every 'pruning_frequency' iterations.
- **convergence_check_frequency** – Perform the check for convergence every convergence_check_frequency iterations

Returns NodeClustering object

Example

```
>>> from cdlib import algorithms
>>> import networkx as nx
>>> G = nx.karate_club_graph()
>>> coms = algorithms.markov_clustering(G)
```

References

Enright, Anton J., Stijn Van Dongen, and Christos A. Ouzounis. An efficient algorithm for large-scale detection of protein families. Nucleic acids research 30.7 (2002): 1575-1584.

Note: Reference implementation: https://github.com/GuyAllard/markov_clustering

cdlib.algorithms.rber_pots

rber_pots(*g_original*, *initial_membership=None*, *weights=None*, *node_sizes=None*, *resolution_parameter=1*)

rber_pots is a model where the quality function to optimize is:

$$Q = \sum_{ij} (A_{ij} - \gamma p)\, \delta(\sigma_i, \sigma_j)$$

where A is the adjacency matrix, $p = \frac{m}{\binom{n}{2}}$ is the overall density of the graph, σ_i denotes the community of node i, $\delta(\sigma_i, \sigma_j) = 1$ if $\sigma_i = \sigma_j$ and *0* otherwise, and, finally γ is a resolution parameter.

Parameters

- **g_original** – a networkx/igraph object
- **initial_membership** – list of int Initial membership for the partition. If `None` then defaults to a singleton partition. Deafault None
- **weights** – list of double, or edge attribute Weights of edges. Can be either an iterable or an edge attribute. Deafault None
- **node_sizes** – list of int, or vertex attribute Sizes of nodes are necessary to know the size of communities in aggregate graphs. Usually this is set to 1 for all nodes, but in specific cases this could be changed. Deafault None
- **resolution_parameter** – double >0 A parameter value controlling the coarseness of the clustering. Higher resolutions lead to more communities, while lower resolutions lead to fewer communities. Deafault 1

Returns NodeClustering object

Example

```
>>> from cdlib import algorithms
>>> import networkx as nx
>>> G = nx.karate_club_graph()
>>> coms = algorithms.rber_pots(G)
```

References

Reichardt, J., & Bornholdt, S. (2006). Statistical mechanics of community detection. Physical Review E, 74(1), 016110. 10.1103/PhysRevE.74.016110

Note: Reference implementation: https://github.com/vtraag/leidenalg

cdlib.algorithms.rb_pots

rb_pots(*g_original*, *initial_membership=None*, *weights=None*, *resolution_parameter=1*)

Rb_pots is a model where the quality function to optimize is:

$$Q = \sum_{ij} \left(A_{ij} - \gamma \frac{k_i k_j}{2m} \right) \delta(\sigma_i, \sigma_j)$$

where A is the adjacency matrix, k_i is the (weighted) degree of node i, m is the total number of edges (or total edge weight), σ_i denotes the community of node i and $\delta(\sigma_i, \sigma_j) = 1$ if $\sigma_i = \sigma_j$ and *0* otherwise. For directed graphs a slightly different formulation is used, as proposed by Leicht and Newman :

$$Q = \sum_{ij} \left(A_{ij} - \gamma \frac{k_i^{\text{out}} k_j^{\text{in}}}{m} \right) \delta(\sigma_i, \sigma_j),$$

where k_i^{out} and k_i^{in} refers to respectively the outdegree and indegree of node i , and A_{ij} refers to an edge from i to j. Note that this is the same of Leiden algorithm when setting $\gamma = 1$ and normalising by $2m$, or m for directed graphs.

Parameters

- **g_original** – a networkx/igraph object
- **initial_membership** – list of int Initial membership for the partition. If `None` then defaults to a singleton partition. Deafault None
- **weights** – list of double, or edge attribute Weights of edges. Can be either an iterable or an edge attribute. Deafault None
- **resolution_parameter** – double >0 A parameter value controlling the coarseness of the clustering. Higher resolutions lead to more communities, while lower resolutions lead to fewer communities. Default 1

Returns NodeClustering object

Example

```
>>> from cdlib import algorithms
>>> import networkx as nx
>>> G = nx.karate_club_graph()
>>> coms = algorithms.rb_pots(G)
```

References

Reichardt, J., & Bornholdt, S. (2006). Statistical mechanics of community detection. Physical Review E, 74(1), 016110. 10.1103/PhysRevE.74.016110

Leicht, E. A., & Newman, M. E. J. (2008). Community Structure in Directed Networks. Physical Review Letters, 100(11), 118703. 10.1103/PhysRevLett.100.118703

cdlib.algorithms.scan

scan(*g_original*, *epsilon*, *mu*)

SCAN (Structural Clustering Algorithm for Networks) is an algorithm which detects clusters, hubs and outliers in networks. It clusters vertices based on a structural similarity measure. The method uses the neighborhood of the vertices as clustering criteria instead of only their direct connections. Vertices are grouped into the clusters by how they share neighbors.

Parameters

- **g_original** – a networkx/igraph object
- **epsilon** – the minimum threshold to assigning cluster membership
- **mu** – minimum number of neineighbors with a structural similarity that exceeds the threshold epsilon

Returns NodeClustering object

Example

```
>>> from cdlib import algorithms
>>> import networkx as nx
>>> G = nx.karate_club_graph()
>>> com = algorithms.scan(G, epsilon=0.7, mu=3)
```

References

Xu, X., Yuruk, N., Feng, Z., & Schweiger, T. A. (2007, August). Scan: a structural clustering algorithm for networks. In Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining (pp. 824-833)

cdlib.algorithms.significance_communities

significance_communities(*g_original*, *initial_membership=None*, *node_sizes=None*)

Significance_communities is a model where the quality function to optimize is:

$$Q = \sum_c \binom{n_c}{2} D(p_c \parallel p)$$

where n_c is the number of nodes in community c, $p_c = \frac{m_c}{\binom{n_c}{2}}$, is the density of community c, $p = \frac{m}{\binom{n}{2}}$ is the overall density of the graph, and finally $D(x \parallel y) = x \ln \frac{x}{y} + (1-x) \ln \frac{1-x}{1-y}$ is the binary Kullback-Leibler divergence. For directed graphs simply multiply the binomials by 2. The expected Significance in Erdos-Renyi graphs behaves roughly as $\frac{1}{2} n \ln n$ for both directed and undirected graphs in this formulation.

Warning: This method is not suitable for weighted graphs.

Parameters

- **g_original** – a networkx/igraph object
- **initial_membership** – list of int Initial membership for the partition. If `None` then defaults to a singleton partition. Deafault None
- **node_sizes** – list of int, or vertex attribute Sizes of nodes are necessary to know the size of communities in aggregate graphs. Usually this is set to 1 for all nodes, but in specific cases this could be changed. Deafault None

Returns NodeClustering object

Example

```
>>> from cdlib import algorithms
>>> import networkx as nx
>>> G = nx.karate_club_graph()
>>> coms = algorithms.significance_communities(G)
```

References

Traag, V. A., Krings, G., & Van Dooren, P. (2013). Significant scales in community structure. Scientific Reports, 3, 2930. *10.1038/srep02930 <http://doi.org/10.1038/srep02930>*

Note: Reference implementation: https://github.com/vtraag/leidenalg

cdlib.algorithms.spinglass

spinglass(*g_original*)

Spinglass relies on an analogy between a very popular statistical mechanic model called Potts spin glass, and the community structure. It applies the simulated annealing optimization technique on this model to optimize the modularity.

Parameters **g_original** – a networkx/igraph object

Returns NodeClustering object

Example

```
>>> from cdlib import algorithms
>>> import networkx as nx
>>> G = nx.karate_club_graph()
>>> com = algorithms.spinglass(G)
```

References

Reichardt, Jörg, and Stefan Bornholdt. Statistical mechanics of community detection. Physical Review E 74.1 (2006): 016110.

cdlib.algorithms.surprise_communities

surprise_communities(*g_original*, *initial_membership=None*, *weights=None*, *node_sizes=None*)

Surprise_communities is a model where the quality function to optimize is:

$$Q = mD(q \parallel \langle q \rangle)$$

where m is the number of edges, $q = \frac{\sum_c m_c}{m}$, is the fraction of internal edges, $\langle q \rangle = \frac{\sum_c \binom{n_c}{2}}{\binom{n}{2}}$ is the expected fraction of internal edges, and finally

$D(x \parallel y) = x \ln \frac{x}{y} + (1-x) \ln \frac{1-x}{1-y}$ is the binary Kullback-Leibler divergence.

For directed graphs we can multiplying the binomials by 2, and this leaves $\langle q \rangle$ unchanged, so that we can simply use the same formulation. For weighted graphs we can simply count the total internal weight instead of the total number of edges for q , while $\langle q \rangle$ remains unchanged.

Parameters

- **g_original** – a networkx/igraph object
- **initial_membership** – list of int Initial membership for the partition. If `None` then defaults to a singleton partition. Deafault None
- **weights** – list of double, or edge attribute Weights of edges. Can be either an iterable or an edge attribute. Deafault None
- **node_sizes** – list of int, or vertex attribute Sizes of nodes are necessary to know the size of communities in aggregate graphs. Usually this is set to 1 for all nodes, but in specific cases this could be changed. Deafault None

Returns NodeClustering object

Example

```
>>> from cdlib import algorithms
>>> import networkx as nx
>>> G = nx.karate_club_graph()
>>> coms = algorithms.surprise_communities(G)
```

References

Traag, V. A., Aldecoa, R., & Delvenne, J.-C. (2015). Detecting communities using asymptotical surprise. Physical Review E, 92(2), 022816. 10.1103/PhysRevE.92.022816

Note: Reference implementation: https://github.com/vtraag/leidenalg

---

cdlib.algorithms.walktrap

walktrap(*g_original*)

walktrap is an approach based on random walks. The general idea is that if you perform random walks on the graph, then the walks are more likely to stay within the same community because there are only a few edges that lead outside a given community. Walktrap runs short random walks and uses the results of these random walks to merge separate communities in a bottom-up manner.

Parameters **g_original** – a networkx/igraph object

Returns NodeClusterint object

Example

```
>>> from cdlib import algorithms
>>> import networkx as nx
>>> G = nx.karate_club_graph()
>>> coms = algorithms.walktrap(G)
```

References

Pons, Pascal, and Matthieu Latapy. Computing communities in large networks using random walks. J. Graph Algorithms Appl. 10.2 (2006): 191-218.

cdlib.algorithms.sbm_dl

sbm_dl(*g_original*, *B_min=None*, *B_max=None*, *deg_corr=True*, ***kwargs*)
Efficient Monte Carlo and greedy heuristic for the inference of stochastic block models.

Fit a non-overlapping stochastic block model (SBM) by minimizing its description length using an agglomerative heuristic. If no parameter is given, the number of blocks will be discovered automatically. Bounds for the number of communities can be provided using B_min, B_max.

Parameters

- **g_original** – network/igraph object
- **B_min** – minimum number of communities that can be found
- **B_max** – maximum number of communities that can be found
- **deg_corr** – if true, use the degree corrected version of the SBM

Returns NodeClustering object

Example

```
>>> from cdlib import algorithms
>>> import networkx as nx
>>> G = nx.karate_club_graph()
>>> coms = sbm_dl(G)
```

References

Tiago P. Peixoto, "Efficient Monte Carlo and greedy heuristic for the inference of stochastic block models", Phys. Rev. E 89, 012804 (2014), DOI: 10.1103/PhysRevE.89.012804 [sci-hub, @tor], arXiv: 1310.4378. .. note:: Use implementation from graph-tool library, please report to https://graph-tool.skewed.de for details

cdlib.algorithms.sbm_dl_nested

sbm_dl_nested(*g_original*, *B_min=None*, *B_max=None*, *deg_corr=True*, ***kwargs*)
Efficient Monte Carlo and greedy heuristic for the inference of stochastic block models. (nested)

Fit a nested non-overlapping stochastic block model (SBM) by minimizing its description length using an agglomerative heuristic. Return the lowest level found. Currently cdlib do not support hierarchical clustering. If no parameter is given, the number of blocks will be discovered automatically. Bounds for the number of communities can be provided using B_min, B_max.

Parameters

- **g_original** – igraph/networkx object
- **B_min** – minimum number of communities that can be found
- **B_max** – maximum number of communities that can be found
- **deg_corr** – if true, use the degree corrected version of the SBM

Returns NodeClustering object

Example

```
>>> from cdlib import algorithms
>>> import networkx as nx
>>> G = nx.karate_club_graph()
>>> coms = sbm_dl(G)
```

References

Tiago P. Peixoto, “Hierarchical block structures and high-resolution model selection in large networks”, Physical Review X 4.1 (2014): 011047 .. note:: Use implementation from graph-tool library, please report to https://graph-tool.skewed.de for details

Overlapping Communities

A clustering is said to be *overlapping* if any generic node can be assigned to more than one community. Methods in this subclass return as result a `NodeClustering` object instance.

angel(g_original, threshold[, ...])	Angel is a node-centric bottom-up community discovery algorithm.
big_clam(g_original[, dimensions, ...])	BigClam is an overlapping community detection method that scales to large networks.
conga(g_original, number_communities)	CONGA (Cluster-Overlap Newman Girvan Algorithm) is an algorithm for discovering overlapping communities.
congo(g_original, number_communities[, height])	CONGO (CONGA Optimized) is an optimization of the CONGA algortithm.
danmf(g_original[, layers, pre_iterations, ...])	The procedure uses telescopic non-negative matrix factorization in order to learn a cluster memmbership distribution over nodes.
demon(g_original, epsilon[, min_com_size])	Demon is a node-centric bottom-up overlapping community discovery algorithm.
ego_networks(g_original[, level])	Ego-networks returns overlapping communities centered at each nodes within a given radius.
egonet_splitter(g_original[, resolution])	The method first creates the egonets of nodes.
kclique(g_original, k)	Find k-clique communities in graph using the percolation method.
lais2(g_original)	LAIS2 is an overlapping community discovery algorithm based on the density function.
lemon(g_original, seeds[, min_com_size, ...])	Lemon is a large scale overlapping community detection method based on local expansion via minimum one norm.
lfm(g_original, alpha)	LFM is based on the local optimization of a fitness function.
multicom(g_original, seed_node)	MULTICOM is an algorithm for detecting multiple local communities, possibly overlapping, by expanding the initial seed set.
nmnf(g_original[, dimensions, clusters, ...])	The procedure uses joint non-negative matrix factorization with modularity based regul;arization in order to learn a cluster memmbership distribution over nodes.

Continued on next page

Table 9 – continued from previous page

nnsed(g_original[, dimensions, iterations, seed])	The procedure uses non-negative matrix factorization in order to learn an unnormalized cluster membership distribution over nodes.
node_perception(g_original, threshold, ...)	Node perception is based on the idea of joining together small sets of nodes.
overlapping_seed_set_expansion(g_original, seeds)	OSSE is an overlapping community detection algorithm optimizing the conductance community score The algorithm uses a seed set expansion approach; the key idea is to find good seeds, and then expand these seed sets using the personalized PageRank clustering procedure.
percomvc(g_original)	The PercoMVC approach composes of two steps.
slpa(g_original[, t, r])	SLPA is an overlapping community discovery that extends tha LPA.
wCommunity(g_original[, min_bel_degree, ...])	Algorithm to identify overlapping communities in weighted graphs

cdlib.algorithms.angel

angel(*g_original*, *threshold*, *min_community_size=3*)

Angel is a node-centric bottom-up community discovery algorithm. It leverages ego-network structures and overlapping label propagation to identify micro-scale communities that are subsequently merged in mesoscale ones. Angel is the, faster, successor of Demon.

Parameters

- **g_original** – a networkx/igraph object
- **threshold** – merging threshold in [0,1].
- **min_community_size** – minimum community size, default 3.

Returns NodeClustering object

Example

```
>>> from cdlib import algorithms
>>> import networkx as nx
>>> G = nx.karate_club_graph()
>>> coms = algorithms.angel(G, min_com_size=3, threshold=0.25)
```

References

1. Rossetti, Giulio. "Exorcising the Demon: Angel, Efficient Node-Centric Community Discovery." International Conference on Complex Networks and Their Applications. Springer, Cham, 2019.

Note: Reference implementation: https://github.com/GiulioRossetti/ANGEL

cdlib.algorithms.big_clam

big_clam(*g_original*, *dimensions=8*, *iterations=50*, *learning_rate=0.005*)

BigClam is an overlapping community detection method that scales to large networks. The procedure uses gradient ascent to create an embedding which is used for deciding the node-cluster affiliations.

Parameters

- **g_original** – a networkx/igraph object
- **dimensions** – Number of embedding dimensions. Default 8.
- **iterations** – Number of training iterations. Default 50.
- **learning_rate** – Gradient ascent learning rate. Default is 0.005.

Returns NodeClustering object

Example

```
>>> from cdlib import algorithms
>>> import networkx as nx
>>> G = nx.karate_club_graph()
>>> coms = algorithms.big_clam(G)
```

References

Yang, Jaewon, and Jure Leskovec. "Overlapping community detection at scale: a nonnegative matrix factorization approach." Proceedings of the sixth ACM international conference on Web search and data mining. 2013.

Note: Reference implementation: https://karateclub.readthedocs.io/

cdlib.algorithms.conga

conga(*g_original*, *number_communities*)

CONGA (Cluster-Overlap Newman Girvan Algorithm) is an algorithm for discovering overlapping communities. It extends the Girvan and Newman's algorithm with a specific method of deciding when and how to split vertices. The algorithm is as follows:

1. Calculate edge betweenness of all edges in network.
2. Calculate vertex betweenness of vertices, from edge betweennesses.
3. Find candidate set of vertices: those whose vertex betweenness is greater than the maximum edge betweenness.
4. If candidate set is non-empty, calculate pair betweennesses of candidate vertices, and then calculate split betweenness of candidate vertices.
5. Remove edge with maximum edge betweenness or split vertex with maximum split betweenness (if greater).
6. Recalculate edge betweenness for all remaining edges in same component(s) as removed edge or split vertex.
7. Repeat from step 2 until no edges remain.

Parameters

- **g_original** – a networkx/igraph object
- **number_communities** – the number of communities desired

Returns NodeClustering object

Example

```
>>> from cdlib import algorithms
>>> import networkx as nx
>>> G = nx.karate_club_graph()
>>> com = algorithms.conga(G, number_communities=3)
```

References

Gregory, Steve. An algorithm to find overlapping community structure in networks. European Conference on Principles of Data Mining and Knowledge Discovery. Springer, Berlin, Heidelberg, 2007.

Note: Reference implementation: https://github.com/Lab41/Circulo/tree/master/circulo/algorithms

---

cdlib.algorithms.congo

congo(*g_original*, *number_communities*, *height=2*)

CONGO (CONGA Optimized) is an optimization of the CONGA algortithm. The CONGO algorithm is the same as CONGA but using local betweenness. The complete CONGO algorithm is as follows:

1. Calculate edge betweenness of edges and split betweenness of vertices.
2. Find edge with maximum edge betweenness or vertex with maximum split betweenness, if greater.
3. **Recalculate edge betweenness and split betweenness:**
   (a) Subtract betweenness of h-region centred on the removed edge or split vertex.
   (b) Remove the edge or split the vertex.
   (c) Add betweenness for the same region.
4. Repeat from step 2 until no edges remain.

Parameters

- **g_original** – a networkx/igraph object
- **number_communities** – the number of communities desired
- **height** – The lengh of the longest shortest paths that CONGO considers, default 2

Returns NodeClustering object

Example

```
>>> from cdlib import algorithms
>>> import networkx as nx
>>> G = nx.karate_club_graph()
>>> com = algorithms.congo(G, number_communities=3, height=2)
```

References

Gregory, Steve. A fast algorithm to find overlapping communities in networks. Joint European Conference on Machine Learning and Knowledge Discovery in Databases. Springer, Berlin, Heidelberg, 2008.

Note: Reference implementation: https://github.com/Lab41/Circulo/tree/master/circulo/algorithms

---

cdlib.algorithms.danmf

danmf(*g_original*, *layers=(32, 8)*, *pre_iterations=100*, *iterations=100*, *seed=42*, *lamb=0.01*)

The procedure uses telescopic non-negative matrix factorization in order to learn a cluster memmbership distribution over nodes. The method can be used in an overlapping and non-overlapping way.

Parameters

- **g_original** – a networkx/igraph object
- **layers** – Autoencoder layer sizes in a list of integers. Default [32, 8].
- **pre_iterations** – Number of pre-training epochs. Default 100.
- **iterations** – Number of training epochs. Default 100.
- **seed** – Random seed for weight initializations. Default 42.
- **lamb** – Regularization parameter. Default 0.01.

Returns NodeClustering object

Example

```
>>> from cdlib import algorithms
>>> import networkx as nx
>>> G = nx.karate_club_graph()
>>> coms = algorithms.danmf(G)
```

References

Ye, Fanghua, Chuan Chen, and Zibin Zheng. "Deep autoencoder-like nonnegative matrix factorization for community detection." Proceedings of the 27th ACM International Conference on Information and Knowledge Management. 2018.

Note: Reference implementation: https://karateclub.readthedocs.io/

cdlib.algorithms.demon

demon(*g_original*, *epsilon*, *min_com_size=3*)

Demon is a node-centric bottom-up overlapping community discovery algorithm. It leverages ego-network structures and overlapping label propagation to identify micro-scale communities that are subsequently merged in mesoscale ones.

Parameters

- **g_original** – a networkx/igraph object
- **epsilon** – merging threshold in [0,1], default 0.25.
- **min_com_size** – minimum community size, default 3.

Returns NodeClustering object

Example

```
>>> from cdlib import algorithms
>>> import networkx as nx
>>> G = nx.karate_club_graph()
>>> coms = algorithms.demon(G, min_com_size=3, epsilon=0.25)
```

References

1. Coscia, M., Rossetti, G., Giannotti, F., & Pedreschi, D. (2012, August). Demon: a local-first discovery method for overlapping communities. In Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining (pp. 615-623). ACM.
2. Coscia, M., Rossetti, G., Giannotti, F., & Pedreschi, D. (2014). Uncovering hierarchical and overlapping communities with a local-first approach. ACM Transactions on Knowledge Discovery from Data (TKDD), 9(1), 6.

Note: Reference implementation: https://github.com/GiulioRossetti/DEMON

cdlib.algorithms.ego_networks

ego_networks(*g_original*, *level=1*)

Ego-networks returns overlapping communities centered at each nodes within a given radius.

Parameters

- **g_original** – a networkx/igraph object
- **level** – extrac communities with all neighbors of distance<=level from a node. Deafault 1

Returns NodeClustering object

Example

```
>>> from cdlib import algorithms
>>> import networkx as nx
>>> G = nx.karate_club_graph()
>>> coms = algorithms.ego_networks(G)
```

cdlib.algorithms.egonet_splitter

egonet_splitter(*g_original*, *resolution=1.0*)

The method first creates the egonets of nodes. A persona-graph is created which is clustered by the Louvain method.

Parameters

- **g_original** – a networkx/igraph object
- **resolution** – Resolution parameter of Python Louvain. Default 1.0.

Returns NodeClustering object

Example

```
>>> from cdlib import algorithms
>>> import networkx as nx
>>> G = nx.karate_club_graph()
>>> coms = algorithms.egonet_splitter(G)
```

References

Epasto, Alessandro, Silvio Lattanzi, and Renato Paes Leme. "Ego-splitting framework: From non-overlapping to overlapping clusters." Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. 2017.

Note: Reference implementation: https://karateclub.readthedocs.io/

---

cdlib.algorithms.kclique

kclique(*g_original*, *k*)

Find k-clique communities in graph using the percolation method. A k-clique community is the union of all cliques of size k that can be reached through adjacent (sharing k-1 nodes) k-cliques.

Parameters

- **g_original** – a networkx/igraph object
- **k** – Size of smallest clique

Returns NodeClustering object

Example

```
>>> from cdlib import algorithms
>>> import networkx as nx
>>> G = nx.karate_club_graph()
>>> com = algorithms.kclique(G, k=3)
```

References

Gergely Palla, Imre Derényi, Illés Farkas1, and Tamás Vicsek, Uncovering the overlapping community structure of complex networks in nature and society Nature 435, 814-818, 2005, doi:10.1038/nature03607

cdlib.algorithms.lais2

lais2(*g_original*)

LAIS2 is an overlapping community discovery algorithm based on the density function. In the algorithm considers the density of a group is defined as the average density of the communication exchanges between the actors of the group. LAIS2 IS composed of two procedures LA (Link Aggregate Algorithm) and IS2 (Iterative Scan Algorithm).

Parameters **g_original** – a networkx/igraph object

Returns NodeClustering object

Example

```
>>> from cdlib import algorithms
>>> import networkx as nx
>>> G = nx.karate_club_graph()
>>> com = algorithms.lais2(G)
```

References

Baumes, Jeffrey, Mark Goldberg, and Malik Magdon-Ismail. Efficient identification of overlapping communities. International Conference on Intelligence and Security Informatics. Springer, Berlin, Heidelberg, 2005.

Note: Reference implementation: https://github.com/kritishrivastava/CommunityDetection-Project2GDM

cdlib.algorithms.lemon

lemon(*g_original*, *seeds*, *min_com_size=20*, *max_com_size=50*, *expand_step=6*, *subspace_dim=3*, *walk_steps=3*, *biased=False*)

Lemon is a large scale overlapping community detection method based on local expansion via minimum one norm.

The algorithm adopts a local expansion method in order to identify the community members from a few exemplary seed members. The algorithm finds the community by seeking a sparse vector in the span of the local spectra such that the seeds are in its support. LEMON can achieve the highest detection accuracy among state-of-the-art proposals. The running time depends on the size of the community rather than that of the entire graph.

Parameters

- **g_original** – a networkx/igraph object
- **seeds** – Node list
- **min_com_size** – the minimum size of a single community in the network, default 20
- **max_com_size** – the maximum size of a single community in the network, default 50
- **expand_step** – the step of seed set increasement during expansion process, default 6
- **subspace_dim** – dimension of the subspace; choosing a large dimension is undesirable because it would increase the computation cost of generating local spectra default 3
- **walk_steps** – the number of step for the random walk, default 3
- **biased** – boolean; set if the random walk starting from seed nodes, default False

Returns NodeClustering object

Example

```
>>> from cdlib import algorithms
>>> import networkx as nx
>>> G = nx.karate_club_graph()
>>> seeds = ["$0$", "$2$", "$3$"]
>>> coms = algorithms.lemon(G, seeds, min_com_size=2, max_com_size=5)
```

References

Yixuan Li, Kun He, David Bindel, John Hopcroft Uncovering the small community structure in large networks: A local spectral approach. Proceedings of the 24th international conference on world wide web. International World Wide Web Conferences Steering Committee, 2015.

Note: Reference implementation: https://github.com/YixuanLi/LEMON

cdlib.algorithms.lfm

lfm(*g_original*, *alpha*)

LFM is based on the local optimization of a fitness function. It finds both overlapping communities and the hierarchical structure.

Parameters

- **g_original** – a networkx/igraph object
- **alpha** – parameter to controll the size of the communities: Large values of alpha yield very small communities, small values instead deliver large modules. If alpha is small enough, all nodes end up in the same cluster, the network itself. In most cases, for alpha < 0.5 there is only one community, for alpha > 2 one recovers the smallest communities. A natural choise is alpha =1.

Returns NodeClustering object

Example

```
>>> from cdlib import algorithms
>>> import networkx as nx
>>> G = nx.karate_club_graph()
>>> com = algorithms.lfm(G, alpha=0.8)
```

References

Lancichinetti, Andrea, Santo Fortunato, and János Kertész. Detecting the overlapping and hierarchical community structure in complex networks New Journal of Physics 11.3 (2009): 033015.

cdlib.algorithms.multicom

multicom(*g_original*, *seed_node*)

MULTICOM is an algorithm for detecting multiple local communities, possibly overlapping, by expanding the initial seed set. This algorithm uses local scoring metrics to define an embedding of the graph around the seed set. Based on this embedding, it picks new seeds in the neighborhood of the original seed set, and uses these new seeds to recover multiple communities.

Parameters

- **g_original** – a networkx/igraph object
- **seed_node** – Id of the seed node around which we want to detect communities.

Returns EdgeClustering object

Example

```
>>> from cdlib import algorithms
>>> import networkx as nx
>>> G = nx.karate_club_graph()
>>> coms = algorithms.multicom(G, seed_node=0)
```

References

Hollocou, Alexandre, Thomas Bonald, and Marc Lelarge. Multiple Local Community Detection. ACM SIGMETRICS Performance Evaluation Review 45.2 (2018): 76-83.

Note: Reference implementation: https://github.com/ahollocou/multicom

cdlib.algorithms.nmnf

nmnf(*g_original*, *dimensions=128*, *clusters=10*, *lambd=0.2*, *alpha=0.05*, *beta=0.05*, *iterations=200*, *lower_control=1e-15*, *eta=5.0*)

The procedure uses joint non-negative matrix factorization with modularity based regul;arization in order to learn a cluster memmbership distribution over nodes. The method can be used in an overlapping and non-overlapping way.

Parameters

- **g_original** – a networkx/igraph object
- **dimensions** – Number of dimensions. Default is 128.
- **clusters** – Number of clusters. Default is 10.
- **lambd** – KKT penalty. Default is 0.2
- **alpha** – Clustering penalty. Default is 0.05.
- **beta** – Modularity regularization penalty. Default is 0.05.
- **iterations** – Number of power iterations. Default is 200.
- **lower_control** – Floating point overflow control. Default is 10**-15.
- **eta** – Similarity mixing parameter. Default is 5.0.

Returns NodeClustering object

Example

```
>>> from cdlib import algorithms
>>> import networkx as nx
>>> G = nx.karate_club_graph()
>>> coms = algorithms.nmnf(G)
```

References

Wang, Xiao, et al. “Community preserving network embedding.” Thirty-first AAAI conference on artificial intelligence. 2017.

Note: Reference implementation: https://karateclub.readthedocs.io/

cdlib.algorithms.nnsed

nnsed(*g_original*, *dimensions=32*, *iterations=10*, *seed=42*)

The procedure uses non-negative matrix factorization in order to learn an unnormalized cluster membership distribution over nodes. The method can be used in an overlapping and non-overlapping way.

Parameters

- **g_original** – a networkx/igraph object
- **dimensions** – Embedding layer size. Default is 32.
- **iterations** – Number of training epochs. Default 10.
- **seed** – Random seed for weight initializations. Default 42.

Returns NodeClustering object

Example

```
>>> from cdlib import algorithms
>>> import networkx as nx
>>> G = nx.karate_club_graph()
>>> coms = algorithms.nnsed(G)
```

References

Sun, Bing-Jie, et al. "A non-negative symmetric encoder-decoder approach for community detection." Proceedings of the 2017 ACM on Conference on Information and Knowledge Management. 2017.

Note: Reference implementation: https://karateclub.readthedocs.io/

cdlib.algorithms.node_perception

node_perception(*g_original*, *threshold*, *overlap_threshold*, *min_comm_size=3*)

Node perception is based on the idea of joining together small sets of nodes. The algorithm first identifies sub-communities corresponding to each node's perception of the network around it. To perform this step, it considers each node individually, and partition that node's neighbors into communities using some existing community detection method. Next, it creates a new network in which every node corresponds to a sub-community, and two nodes are linked if their associated sub-communities overlap by at least some threshold amount. Finally, the algorithm identifies overlapping communities in this new network, and for every such community, merge together the associated sub-communities to identify communities in the original network.

Parameters

- **g_original** – a networkx/igraph object
- **threshold** – the tolerance required in order to merge communities
- **overlap_threshold** – the overlap tolerance
- **min_comm_size** – minimum community size default 3

Returns NodeClustering object

Example

```
>>> from cdlib import algorithms
>>> import networkx as nx
>>> G = nx.karate_club_graph()
>>> coms = algorithms.node_perception(G, threshold=0.25, overlap_threshold=0.25)
```

References

Sucheta Soundarajan and John E. Hopcroft. 2015. Use of Local Group Information to Identify Communities in Networks. ACM Trans. Knowl. Discov. Data 9, 3, Article 21 (April 2015), 27 pages. DOI=http://dx.doi.org/10.1145/2700404

cdlib.algorithms.overlapping_seed_set_expansion

overlapping_seed_set_expansion(*g_original*, *seeds*, *ninf=False*, *expansion='ppr'*, *stopping='cond'*, *nworkers=1*, *nruns=13*, *alpha=0.99*, *maxexpand=inf*, *delta=0.2*)

OSSE is an overlapping community detection algorithm optimizing the conductance community score The algorithm uses a seed set expansion approach; the key idea is to find good seeds, and then expand these seed sets using the personalized PageRank clustering procedure.

Parameters

- **g_original** – a networkx/igraph object
- **seeds** – Node list
- **ninf** – Neighbourhood Inflation parameter (boolean)
- **expansion** – Seed expansion: ppr or vppr
- **stopping** – Stopping criteria: cond
- **nworkers** – Number of Workers: default 1
- **nruns** – Number of runs: default 13
- **alpha** – alpha value for Personalized PageRank expansion: default 0.99
- **maxexpand** – Maximum expansion allowed for approximate ppr: default INF
- **delta** – Minimum distance parameter for near duplicate communities: default 0.2

Returns NodeClustering object

Example

```
>>> from cdlib import algorithms
>>> import networkx as nx
>>> G = nx.karate_club_graph()
>>> com = algorithms.overlapping_seed_set_expansion(G)
```

References

1.Whang, J. J., Gleich, D. F., & Dhillon, I. S. (2013, October). Overlapping community detection using seed set expansion. In Proceedings of the 22nd ACM international conference on Conference on information & knowledge management (pp. 2099-2108). ACM.

Note: Reference implementation: https://github.com/pratham16/algorithms-detection-by-seed-expansion

cdlib.algorithms.percomvc

percomvc(*g_original*)

The PercoMVC approach composes of two steps. In the first step, the algorithm attempts to determine all communities that the clique percolation algorithm may find. In the second step, the algorithm computes the Eigenvector Centrality method on the output of the first step to measure the influence of network nodes and reduce the rate of the unclassified nodes

Parameters **g_original** – a networkx/igraph object

Returns NodeClustering object

Example

```
>>> from cdlib import algorithms
>>> import networkx as nx
>>> G = nx.karate_club_graph()
>>> coms = algorithms.percomvc(G)
```

References

Kasoro, Nathanaël, et al. "PercoMCV: A hybrid approach of community detection in social networks." Procedia Computer Science 151 (2019): 45-52.

Note: Reference implementation: https://github.com/sedjokas/PercoMCV-Code-source

cdlib.algorithms.slpa

slpa(*g_original*, *t=21*, *r=0.1*)

SLPA is an overlapping community discovery that extends tha LPA. SLPA consists of the following three stages: 1) the initialization 2) the evolution 3) the post-processing

Parameters

- **g_original** – a networkx/igraph object
- **t** – maximum number of iterations, default 20
- **r** – threshold [0, 1]. It is used in the post-processing stage: if the probability of seeing a particular label during the whole process is less than r, this label is deleted from a node's memory. Default 0.1

Returns EdgeClustering object

Example

```
>>> from cdlib import algorithms
>>> import networkx as nx
>>> G = nx.karate_club_graph()
>>> coms = algorithms.slpa(G,  t=21, r=0.1)
```

References

Xie Jierui, Boleslaw K. Szymanski, and Xiaoming Liu. Slpa: Uncovering overlapping communities in social networks via a speaker-listener interaction dynamic process. Data Mining Workshops (ICDMW), 2011 IEEE 11th International Conference on. IEEE, 2011.

Note: Reference implementation: https://github.com/kbalasu/SLPA

cdlib.algorithms.wCommunity

wCommunity(*g_original*, *min_bel_degree=0.7*, *threshold_bel_degree=0.7*, *weightName='weight'*)

Algorithm to identify overlapping communities in weighted graphs

Parameters

- **g_original** – a networkx/igraph object
- **min_bel_degree** – the tolerance, in terms of beloging degree, required in order to add a node in a community
- **threshold_bel_degree** – the tolerance, in terms of beloging degree, required in order to add a node in a 'NLU' community
- **weightName** – name of the edge attribute containing the weights

Returns NodeClustering object

Example

```
>>> from cdlib import algorithms
>>> import networkx as nx
>>> G = nx.karate_club_graph()
>>> nx.set_edge_attributes(G, values=1, name='weight')
>>> coms = algorithms.wCommunity(G, min_bel_degree=0.6, threshold_bel_degree=0.6)
```

References

Chen, D., Shang, M., Lv, Z., & Fu, Y. (2010). Detecting overlapping communities of weighted networks via a local algorithm. Physica A: Statistical Mechanics and its Applications, 389(19), 4177-4187.

Note: Implementation provided by Marco Cardia <cardiamc@gmail.com> and Francesco Sabiu <fsabiu@gmail.com> (Computer Science Dept., University of Pisa, Italy)

Fuzzy Communities

A clustering is said to be a *fuzzy* if each node can belongs (with a different degree of likelihood) to more than one community. Methods in this subclass return as result a `FuzzyNodeClustering` object instance.

frc_fgsn(g_original, theta, eps, r)	Fuzzy-Rough Community Detection on Fuzzy Granular model of Social Network.

cdlib.algorithms.frc_fgsn

frc_fgsn(*g_original*, *theta*, *eps*, *r*)

Fuzzy-Rough Community Detection on Fuzzy Granular model of Social Network.

FRC-FGSN assigns nodes to communities specifying the probability of each association. The flattened partition ensure that each node is associated to the community that maximize such association probability. FRC-FGSN may generate orphan nodes (i.e., nodes not assigned to any community).

Parameters

- **g_original** – networkx/igraph object
- **theta** – community density coefficient
- **eps** – coupling coefficient of the community. Ranges in [0, 1], small values ensure that only strongly connected node granules are merged togheter.
- **r** – radius of the granule (int)

Returns FuzzyNodeClustering object

Example

```
>>> from cdlib import algorithms
>>> import networkx as nx
>>> G = nx.karate_club_graph()
>>> coms = frc_fgsn(G, theta=1, eps=0.5, r=3)
```

References

Kundu, S., & Pal, S. K. (2015). Fuzzy-rough community in social networks. Pattern Recognition Letters, 67, 145-152.

Note: Reference implementation: https://github.com/nidhisridhar/Fuzzy-Community-Detection

Node Attribute

Methods in this subclass return as result a `AttrNodeClustering` object instance.

eva(g_original, labels[, weight, ...])	The Eva algorithm extends the Louvain approach in order to deal with the attributes of the nodes (aka Louvain Extended to Vertex Attributes).
ilouvain(g_original, labels, id)	The I-Louvain algorithm extends the Louvain approach in order to deal only with the scalar attributes of the nodes.

cdlib.algorithms.eva

eva(*g_original*, *labels*, *weight='weight'*, *resolution=1.0*, *randomize=False*, *alpha=0.5*)

The Eva algorithm extends the Louvain approach in order to deal with the attributes of the nodes (aka Louvain Extended to Vertex Attributes). It optimizes - combining them linearly - two quality functions, a structural and a clustering one, namely Newman's modularity and purity, estimated as the product of the frequencies of the most frequent labels carried by the nodes within the communities. A parameter alpha tunes the importance of the two functions: an high value of alpha favors the clustering criterion instead of the structural one.

param g_original a networkx/igraph object

param labels dictionary specifying for each node (key) a dict (value) specifying the name attribute (key) and its value (value)

param weight str, optional the key in graph to use as weight. Default to 'weight'

param resolution double, optional Will change the size of the communities, default to 1.

param randomize boolean, optional Will randomize the node evaluation order and the community evaluation order to get different partitions at each call, default False

param alpha float, assumed in [0,1], optional Will tune the importance of modularity and purity criteria, default to 0.5

return AttrNodeClustering object

Example

```
>>> from cdlib.algorithms import eva
>>> import networkx as nx
>>> import random
>>> l1 = ['A', 'B', 'C', 'D']
>>> l2 = ["E", "F", "G"]
>>> g_attr = nx.barabasi_albert_graph(100, 5)
>>> labels=dict()
>>> for node in g_attr.nodes():
>>>    labels[node]={"l1":random.choice(l1), "l2":random.
↪choice(l2)}
>>> communities = eva(g_attr, labels, alpha=0.8)
```

References

1. Citraro, S., & Rossetti, G. (2019, December). Eva: Attribute-Aware Network Segmentation. In International Conference on Complex Networks and Their Applications (pp. 141-151). Springer, Cham.

Note: Reference implementation: https://github.com/GiulioRossetti/Eva/tree/master/Eva

cdlib.algorithms.ilouvain

ilouvain(*g_original*, *labels*, *id*)

The I-Louvain algorithm extends the Louvain approach in order to deal only with the scalar attributes of the nodes. It optimizes Newman's modularity combined with an entropy measure.

param g_original a networkx/igraph object

param labels dictionary specifying for each node (key) a dict (value) specifying the name attribute (key) and its value (value)

param id a dict specifying the node id

return AttrNodeClustering object

Example

```
>>> from cdlib.algorithms import ilouvain
>>> import networkx as nx
>>> import random
>>> l1 = [0.1, 0.4, 0.5]
>>> l2 = [34, 3, 112]
>>> g_attr = nx.barabasi_albert_graph(100, 5)
>>> labels=dict()
>>> for node in g_attr.nodes():
>>>    labels[node]={"l1":random.choice(l1), "l2":random.
↪choice(l2)}
>>> id = dict()
>>> for n in g.nodes():
>>>     id[n] = n
>>> communities = ilouvain(g_attr, labels, id)
```

References

1. Combe D., Largeron C., Géry M., Egyed-Zsigmond E. "I-Louvain: An Attributed Graph Clustering Method". <https://link.springer.com/chapter/10.1007/978-3-319-24465-5_16> In: Fromont E., De Bie T., van Leeuwen M. (eds) Advances in Intelligent Data Analysis XIV. IDA (2015). Lecture Notes in Computer Science, vol 9385. Springer, Cham

Bipartite Graph Communities

Methods in this subclass return as result a `BiNodeClustering` object instance.

bimlpa(g_original[, theta, lambd])	BiMLPA is designed to detect the many-to-many correspondence community in bipartite networks using multi-label propagation algorithm.

cdlib.algorithms.bimlpa

bimlpa (*g_original*, *theta=0.3*, *lambd=7*)

BiMLPA is designed to detect the many-to-many correspondence community in bipartite networks using multi-label propagation algorithm.

Parameters

- **g_original** – a networkx/igraph object
- **theta** – Label weights threshold. Default 0.3.
- **lambd** – The max number of labels. Default 7.

Returns BiNodeClustering object

Example

```
>>> from cdlib import algorithms
>>> import networkx as nx
>>> G = nx.algorithms.bipartite.generators.random_graph(100, 20, 0.1)
>>> coms = algorithms.bimlpa(G)
```

References

Taguchi, Hibiki, Tsuyoshi Murata, and Xin Liu. "BiMLPA: Community Detection in Bipartite Networks by Multi-Label Propagation." International Conference on Network Science. Springer, Cham, 2020.

Note: Reference implementation: https://github.com/hbkt/BiMLPA

Antichain Communities

Methods in this subclass are designed to extract communities from Directed Acyclic Graphs (DAG) and return as result a `NodeClustering` object instance.

siblinarity_antichain(g_original[, ...])	The algorithm extract communities from a DAG that (i) respects its intrinsic order and (ii) are composed of similar nodes.

cdlib.algorithms.siblinarity_antichain

siblinarity_antichain(*g_original*, *forwards_backwards_on=True*, *backwards_forwards_on=False*, *Lambda=1*, *with_replacement=False*, *space_label=None*, *time_label=None*)

The algorithm extract communities from a DAG that (i) respects its intrinsic order and (ii) are composed of similar nodes. The approach takes inspiration from classic similarity measures of bibliometrics, used to assess how similar two publications are, based on their relative citation patterns.

Parameters

- **g_original** – a networkx/igraph object representing a DAG (directed acyclic graph)
- **forwards_backwards_on** – checks successors' similarity. Boolean, default True
- **backwards_forwards_on** – checks predecessors' similarity. Boolean, default True
- **Lambda** – desired resolution of the partition. Default 1
- **with_replacement** – If True he similarity of a node to itself is equal to the number of its neighbours based on which the similarity is defined. Boolean, default True.

Returns NodeClustering object

Example

```
>>> from cdlib import algorithms
>>> import networkx as nx
>>> G = nx.karate_club_graph()
>>> coms = algorithms.siblinarity_antichain(G, Lambda=1)
```

References

Vasiliauskaite, V., Evans, T.S. Making communities show respect for order. Appl Netw Sci 5, 15 (2020). https://doi.org/10.1007/s41109-020-00255-5

Note: Reference implementation: https://github.com/vv2246/siblinarity_antichains

Edge Clustering

Algorithms falling in this category generates communities composed by edges. They return as result a `EdgeClustering` object instance.

hierarchical_link_community(g_original)	HLC (hierarchical link clustering) is a method to classify links into topologically related groups.

cdlib.algorithms.hierarchical_link_community

hierarchical_link_community(*g_original*)

HLC (hierarchical link clustering) is a method to classify links into topologically related groups. The algorithm uses a similarity between links to build a dendrogram where each leaf is a link from the original network and branches represent link communities. At each level of the link dendrogram is calculated the partition density function, based on link density inside communities, to pick the best level to cut.

Parameters **g_original** – a networkx/igraph object

Returns EdgeClustering object

Example

```
>>> from cdlib import algorithms
>>> import networkx as nx
>>> G = nx.karate_club_graph()
>>> com = algorithms.hierarchical_link_community(G)
```

References

Ahn, Yong-Yeol, James P. Bagrow, and Sune Lehmann. Link communities reveal multiscale complexity in networks. nature 466.7307 (2010): 761.

1.5.3 Ensemble Methods

Methods to automate the execution of multiple instances of community detection algorithm(s).

Configuration Objects

Ranges can be specified to automate the execution of a same method while varying (part of) its inputs.

`Parameter` allows to specify ranges for numeric parameters, while `BoolParamter` for boolean ones.

Parameter(name, start, end, step)
BoolParameter(name, value)

cdlib.ensemble.Parameter

class Parameter(*name*, *start*, *end*, *step*)

__init__()
Initialize self. See help(type(self)) for accurate signature.

Methods

count	Return number of occurrences of value.
index	Return first index of value.

Attributes

end	Alias for field number 2
name	Alias for field number 0
start	Alias for field number 1
step	Alias for field number 3

cdlib.ensemble.BoolParameter

class BoolParameter(*name*, *value*)

__init__()
Initialize self. See help(type(self)) for accurate signature.

Methods

count	Return number of occurrences of value.
index	Return first index of value.

Attributes

name	Alias for field number 0
value	Alias for field number 1

Multiple Instantiation

Two scenarios often arise when applying community discovery algorithms to a graph: 1. the need to compare the results obtained by a give algorithm while varying its parameters 2. the need to compare the multiple algorithms

`cdlib` allows to do so by leveraging, respectively, `grid_execution` and `pool`.

grid_execution(graph, method, parameters)	Instantiate the specified community discovery method performing a grid search on the parameter set.
pool(graph, methods, configurations)	Execute on a pool of community discovery internal on the input graph.

cdlib.ensemble.grid_execution

grid_execution (*graph*, *method*, *parameters*)

Instantiate the specified community discovery method performing a grid search on the parameter set.

Parameters

- **method** – community discovery method (from nclib.community)
- **graph** – networkx/igraph object
- **parameters** – list of Parameter and BoolParameter objects

Returns at each call the generator yields a tuple composed by the current configuration and the obtained communities

Example

```
>>> import networkx as nx
>>> from cdlib import algorithms, ensemble
>>> g = nx.karate_club_graph()
>>> resolution = ensemble.Parameter(name="resolution", start=0.1, end=1, step=0.1)
>>> for communities in ensemble.grid_execution(graph=g, method=algorithms.louvain,
↪ parameters=[resolution]):
>>>     print(communities)
```

cdlib.ensemble.pool

pool (*graph*, *methods*, *configurations*)

Execute on a pool of community discovery internal on the input graph.

Parameters

- **methods** – list community discovery methods (from nclib.community)
- **graph** – networkx/igraph object
- **configurations** – list of lists (one for each method) of Parameter and BoolParameter objects

Returns at each call the generator yields a tuple composed by: the actual method, its current configuration and the obtained communities

Raises **ValueError** – if the number of methods is different from the number of configurations specified

Example

```
>>> import networkx as nx
>>> from cdlib import algorithms, ensemble
>>> g = nx.karate_club_graph()
>>> # Louvain
```

(continues on next page)

(continued from previous page)

```
>>> resolution = ensemble.Parameter(name="resolution", start=0.1, end=1, step=0.1)
>>> randomize = ensemble.BoolParameter(name="randomize")
>>> louvain_conf = [resolution, randomize]
>>>
>>> # Angel
>>> threshold = ensemble.Parameter(name="threshold", start=0.1, end=1, step=0.1)
>>> angel_conf = [threshold]
>>>
>>> methods = [algorithms.louvain, algorithms.angel]
>>>
>>> for communities in ensemble.pool(g, methods, [louvain_conf, angel_conf]):
>>>     print(communities)
```

Optimal Configuration Search

In some scenarios it could be helpful delegate to the library the selection of the method parameters to obtain a partition that optimize a given quality function. `cdlib` allows to do so using the methods `grid_search` and `random_search`. Finally, `pool_grid_filter` generalizes such approach allowing to obtain the optimal partitions from a pool of different algorithms.

grid_search(graph, method, parameters, ...)	Returns the optimal partition of the specified graph w.r.t.
random_search(graph, method, parameters, ...)	Returns the optimal partition of the specified graph w.r.t.
pool_grid_filter(graph, methods, ... [, ...])	Execute a pool of community discovery internal on the input graph.

cdlib.ensemble.grid_search

grid_search (*graph*, *method*, *parameters*, *quality_score*, *aggregate=<built-in function max>*)

Returns the optimal partition of the specified graph w.r.t. the selected algorithm and quality score.

Parameters

- **method** – community discovery method (from nclib.community)
- **graph** – networkx/igraph object
- **parameters** – list of Parameter and BoolParameter objects
- **quality_score** – a fitness function to evaluate the obtained partition (from nclib.evaluation)
- **aggregate** – function to select the best fitness value. Possible values: min/max

Returns at each call the generator yields a tuple composed by: the optimal configuration for the given algorithm, input paramters and fitness function; the obtained communities; the fitness score

Example

```
>>> import networkx as nx
>>> from cdlib import algorithms, ensemble
>>> g = nx.karate_club_graph()
>>> resolution = ensemble.Parameter(name="resolution", start=0.1, end=1, step=0.1)
>>> randomize = ensemble.BoolParameter(name="randomize")
>>> communities, scoring = ensemble.grid_search(graph=g, method=algorithms.
↪louvain,
```

(continues on next page)

(continued from previous page)

```
>>>                                                    parameters=[resolution,␣
↪randomize],
>>>                                                    quality_score=evaluation.
↪erdos_renyi_modularity,
>>>                                                    aggregate=max)
>>> print(communities, scoring)
```

cdlib.ensemble.random_search

random_search(*graph*, *method*, *parameters*, *quality_score*, *instances=10*, *aggregate=<built-in function max>*)

Returns the optimal partition of the specified graph w.r.t. the selected algorithm and quality score over a randomized sample of the input parameters.

Parameters

- **method** – community discovery method (from nclib.community)
- **graph** – networkx/igraph object
- **parameters** – list of Parameter and BoolParameter objects
- **quality_score** – a fitness function to evaluate the obtained partition (from nclib.evaluation)
- **instances** – number of randomly selected parameters configurations
- **aggregate** – function to select the best fitness value. Possible values: min/max

Returns at each call the generator yields a tuple composed by: the optimal configuration for the given algorithm, input paramters and fitness function; the obtained communities; the fitness score

Example

```
>>> import networkx as nx
>>> from cdlib import algorithms, ensemble
>>> g = nx.karate_club_graph()
>>> resolution = ensemble.Parameter(name="resolution", start=0.1, end=1, step=0.1)
>>> randomize = ensemble.BoolParameter(name="randomize")
>>> communities, scoring = ensemble.random_search(graph=g, method=algorithms.
↪louvain,
>>>                                                    parameters=[resolution,␣
↪randomize],
>>>                                                    quality_
↪score=evaluation.erdos_renyi_modularity,
>>>                                                    instances=5,␣
↪aggregate=max)
>>> print(communities, scoring)
```

cdlib.ensemble.pool_grid_filter

pool_grid_filter(*graph*, *methods*, *configurations*, *quality_score*, *aggregate=<built-in function max>*)

Execute a pool of community discovery internal on the input graph. Returns the optimal partition for each algorithm given the specified quality function.

Parameters

- **methods** – list community discovery methods (from nclib.community)
- **graph** – networkx/igraph object
- **configurations** – list of lists (one for each method) of Parameter and BoolParameter objects
- **quality_score** – a fitness function to evaluate the obtained partition (from nclib.evaluation)
- **aggregate** – function to select the best fitness value. Possible values: min/max

Returns at each call the generator yields a tuple composed by: the actual method, its optimal configuration; the obtained communities; the fitness score.

Raises **ValueError** – if the number of methods is different from the number of configurations specified

Example

```
>>> import networkx as nx
>>> from cdlib import algorithms, ensemble
>>> g = nx.karate_club_graph()
>>> # Louvain
>>> resolution = ensemble.Parameter(name="resolution", start=0.1, end=1, step=0.1)
>>> randomize = ensemble.BoolParameter(name="randomize")
>>> louvain_conf = [resolution, randomize]
>>>
>>> # Angel
>>> threshold = ensemble.Parameter(name="threshold", start=0.1, end=1, step=0.1)
>>> angel_conf = [threshold]
>>>
>>> methods = [algorithms.louvain, algorithms.angel]
>>>
>>> for communities, scoring in ensemble.pool_grid_filter(g, methods, [louvain_
↪conf, angel_conf], quality_score=evaluation.erdos_renyi_modularity,␣
↪aggregate=max):
>>>     print(communities, scoring)
```

1.5.4 Evaluation

The evaluation of Community Discovery algorithms is not an easy task. CDlib implements two families of evaluation strategies:

- Internal evaluation through quality scores
- External evaluation through partitions comparison

Fitness Functions

Fitness functions allows to summarize the characteristics of a computed set of communities. CDlib implements the following quality scores:

avg_distance(graph, communities, **kwargs)	Average distance.
avg_embeddedness(graph, communities, **kwargs)	Average embeddedness of nodes within the community.

Continued on next page

Table 22 – continued from previous page

average_internal_degree(graph, community, ...)	The average internal degree of the community set.
avg_transitivity(graph, communities, **kwargs)	Average transitivity.
conductance(graph, community, **kwargs)	Fraction of total edge volume that points outside the community.
cut_ratio(graph, community, **kwargs)	Fraction of existing edges (out of all possible edges) leaving the community.
edges_inside(graph, community, **kwargs)	Number of edges internal to the community.
expansion(graph, community, **kwargs)	Number of edges per community node that point outside the cluster.
fraction_over_median_degree(graph, ...)	Fraction of community nodes of having internal degree higher than the median degree value.
hub_dominance(graph, communities, **kwargs)	Hub dominance.
internal_edge_density(graph, community, **kwargs)	The internal density of the community set.
normalized_cut(graph, community, **kwargs)	Normalized variant of the Cut-Ratio
max_odf(graph, community, **kwargs)	Maximum fraction of edges of a node of a community that point outside the community itself.
avg_odf(graph, community, **kwargs)	Average fraction of edges of a node of a community that point outside the community itself.
flake_odf(graph, community, **kwargs)	Fraction of nodes in S that have fewer edges pointing inside than to the outside of the community.
scaled_density(graph, communities, **kwargs)	Scaled density.
significance(graph, communities, **kwargs)	Significance estimates how likely a partition of dense communities appear in a random graph.
size(graph, communities, **kwargs)	Size is the number of nodes in the community
surprise(graph, communities, **kwargs)	Surprise is statistical approach proposes a quality metric assuming that edges between vertices emerge randomly according to a hyper-geometric distribution.
triangle_participation_ratio(graph, ...)	Fraction of community nodes that belong to a triad.
purity(communities)	Purity is the product of the frequencies of the most frequent labels carried by the nodes within the communities

cdlib.evaluation.avg_distance

avg_distance(*graph*, *communities*, ***kwargs*)

Average distance.

The average distance of a community is defined average path length across all possible pair of nodes composing it.

Parameters

- **graph** – a networkx/igraph object
- **communities** – NodeClustering object
- **summary** – boolean. If **True** it is returned an aggregated score for the partition is returned, otherwise individual-community ones. Default **True**.

Returns If **summary==True** a FitnessResult object, otherwise a list of floats.

Example:

```
>>> from cdlib.algorithms import louvain
>>> from cdlib import evaluation
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> scd = evaluation.avg_distance(g,communities)
```

cdlib.evaluation.avg_embeddedness

avg_embeddedness(*graph*, *communities*, ***kwargs*)

Average embeddedness of nodes within the community.

The embeddedness of a node n w.r.t. a community C is the ratio of its degree within the community and its overall degree.

$$emb(n, C) = \frac{k_n^C}{k_n}$$

The average embeddedness of a community C is:

$$avg_e mbd(c) = \frac{1}{|C|} \sum_{i \in C} \frac{k_n^C}{k_n}$$

Parameters

- **graph** – a networkx/igraph object
- **communities** – NodeClustering object
- **summary** – boolean. If **True** it is returned an aggregated score for the partition is returned, otherwise individual-community ones. Default **True**.

Returns If **summary==True** a FitnessResult object, otherwise a list of floats.

Example:

```
>>> from cdlib.algorithms import louvain
>>> from cdlib import evaluation
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> ave = evaluation.avg_embeddedness(g,communities)
```

References

cdlib.evaluation.average_internal_degree

average_internal_degree(*graph*, *community*, ***kwargs*)

The average internal degree of the community set.

$$f(S) = \frac{2m_S}{n_S}$$

$where : math : `m_S` is the number of community internal edges and : math : `n_S` is the number of community nodes.$

Parameters

- **graph** – a networkx/igraph object
- **community** – NodeClustering object

- **summary** – boolean. If **True** it is returned an aggregated score for the partition is returned, otherwise individual-community ones. Default **True**.

Returns If **summary==True** a FitnessResult object, otherwise a list of floats.

Example:

```
>>> from cdlib.algorithms import louvain
>>> from cdlib import evaluation
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = evaluation.average_internal_degree(g,communities)
```

References

1. Radicchi, F., Castellano, C., Cecconi, F., Loreto, V., & Parisi, D. (2004). Defining and identifying communities in networks. Proceedings of the National Academy of Sciences, 101(9), 2658-2663.

cdlib.evaluation.avg_transitivity

avg_transitivity(*graph*, *communities*, ***kwargs*)

Average transitivity.

The average transitivity of a community is defined the as the average clustering coefficient of its nodes w.r.t. their connection within the community itself.

Parameters

- **graph** – a networkx/igraph object
- **communities** – NodeClustering object
- **summary** – boolean. If **True** it is returned an aggregated score for the partition is returned, otherwise individual-community ones. Default **True**.

Returns If **summary==True** a FitnessResult object, otherwise a list of floats.

Example:

```
>>> from cdlib.algorithms import louvain
>>> from cdlib import evaluation
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> scd = evaluation.avg_transitivity(g,communities)
```

cdlib.evaluation.conductance

conductance(*graph*, *community*, ***kwargs*)

Fraction of total edge volume that points outside the community.

$$f(S) = \frac{c_S}{2m_S + c_S}$$

where c_S is the number of community nodes and, m_S is the number of community edges

Parameters

- **graph** – a networkx/igraph object

- **community** – NodeClustering object
- **summary** – boolean. If **True** it is returned an aggregated score for the partition is returned, otherwise individual-community ones. Default **True**.

Returns If **summary==True** a FitnessResult object, otherwise a list of floats.

Example:

```
>>> from cdlib.algorithms import louvain
>>> from cdlib import evaluation
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = evaluation.conductance(g,communities)
```

References

1.Shi, J., Malik, J.: Normalized cuts and image segmentation. Departmental Papers (CIS), 107 (2000)

cdlib.evaluation.cut_ratio

cut_ratio (*graph*, *community*, ***kwargs*)

Fraction of existing edges (out of all possible edges) leaving the community.

..math:: f(S) = frac{c_S}{n_S (n n_S)}

where c_S is the number of community nodes and, n_S is the number of edges on the community boundary

Parameters

- **graph** – a networkx/igraph object
- **community** – NodeClustering object
- **summary** – boolean. If **True** it is returned an aggregated score for the partition is returned, otherwise individual-community ones. Default **True**.

Returns If **summary==True** a FitnessResult object, otherwise a list of floats.

Example:

```
>>> from cdlib.algorithms import louvain
>>> from cdlib import evaluation
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = evaluation.cut_ratio(g,communities)
```

References

1. Fortunato, S.: Community detection in graphs. Physics reports 486(3-5), 75–174 (2010)

cdlib.evaluation.edges_inside

edges_inside (*graph*, *community*, ***kwargs*)

Number of edges internal to the community.

Parameters

- **graph** – a networkx/igraph object

- **community** – NodeClustering object
- **summary** – boolean. If **True** it is returned an aggregated score for the partition is returned, otherwise individual-community ones. Default **True**.

Returns If **summary==True** a FitnessResult object, otherwise a list of floats.

Example:

```
>>> from cdlib.algorithms import louvain
>>> from cdlib import evaluation
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = evaluation.edges_inside(g,communities)
```

References

1. Radicchi, F., Castellano, C., Cecconi, F., Loreto, V., & Parisi, D. (2004). Defining and identifying communities in networks. Proceedings of the National Academy of Sciences, 101(9), 2658-2663.

cdlib.evaluation.expansion

expansion(*graph*, *community*, ***kwargs*)

Number of edges per community node that point outside the cluster.

$$f(S) = \frac{c_S}{n_S}$$

where n_S is the number of edges on the community boundary, c_S is the number of community nodes.

Parameters

- **graph** – a networkx/igraph object
- **community** – NodeClustering object
- **summary** – boolean. If **True** it is returned an aggregated score for the partition is returned, otherwise individual-community ones. Default **True**.

Returns If **summary==True** a FitnessResult object, otherwise a list of floats.

Example:

```
>>> from cdlib.algorithms import louvain
>>> from cdlib import evaluation
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = evaluation.expansion(g,communities)
```

References

1. Radicchi, F., Castellano, C., Cecconi, F., Loreto, V., & Parisi, D. (2004). Defining and identifying communities in networks. Proceedings of the National Academy of Sciences, 101(9), 2658-2663.

cdlib.evaluation.fraction_over_median_degree

fraction_over_median_degree (*graph*, *community*, ***kwargs*)

Fraction of community nodes of having internal degree higher than the median degree value.

$$f(S) = \frac{|\{u : u \in S, |\{(u, v) : v \in S\}| > d_m\}|}{n_S}$$

where d_m is the internal degree median value

Parameters

- **graph** – a networkx/igraph object
- **community** – NodeClustering object
- **summary** – boolean. If **True** it is returned an aggregated score for the partition is returned, otherwise individual-community ones. Default **True**.

Returns If **summary==True** a FitnessResult object, otherwise a list of floats.

Example:

```
>>> from cdlib.algorithms import louvain
>>> from cdlib import evaluation
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = evaluation.fraction_over_median_degree(g,communities)
```

References

1. Yang, J., Leskovec, J.: Defining and evaluating network communities based on ground-truth. Knowledge and Information Systems 42(1), 181–213 (2015)

cdlib.evaluation.hub_dominance

hub_dominance (*graph*, *communities*, ***kwargs*)

Hub dominance.

The hub dominance of a community is defined as the ratio of the degree of its most connected node w.r.t. the theoretically maximal degree within the community.

Parameters

- **graph** – a networkx/igraph object
- **communities** – NodeClustering object
- **summary** – boolean. If **True** it is returned an aggregated score for the partition is returned, otherwise individual-community ones. Default **True**.

Returns If **summary==True** a FitnessResult object, otherwise a list of floats.

Example:

```
>>> from cdlib.algorithms import louvain
>>> from cdlib import evaluation
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> scd = evaluation.hub_dominance(g,communities)
```

cdlib.evaluation.internal_edge_density

internal_edge_density(*graph*, *community*, ***kwargs*)

The internal density of the community set.

$$f(S) = \frac{m_S}{n_S(n_S 1)/2}$$

where m_S is the number of community internal edges and n_S is the number of community nodes.

Parameters

- **graph** – a networkx/igraph object
- **community** – NodeClustering object
- **summary** – boolean. If **True** it is returned an aggregated score for the partition is returned, otherwise individual-community ones. Default **True**.

Returns If **summary==True** a FitnessResult object, otherwise a list of floats.

Example:

```
>>> from cdlib.algorithms import louvain
>>> from cdlib import evaluation
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = evaluation.internal_edge_density(g,communities)
```

References

1. Radicchi, F., Castellano, C., Cecconi, F., Loreto, V., & Parisi, D. (2004). Defining and identifying communities in networks. Proceedings of the National Academy of Sciences, 101(9), 2658-2663.

cdlib.evaluation.normalized_cut

normalized_cut(*graph*, *community*, ***kwargs*)

Normalized variant of the Cut-Ratio

$$f(S) = \frac{c_S}{2m_S + c_S} + \frac{c_S}{2(mm_S) + c_S}$$

where m is the number of graph edges, m_S is the number of community internal edges and c_S is the number of community nodes.

Parameters

- **graph** – a networkx/igraph object
- **community** – NodeClustering object
- **summary** – boolean. If **True** it is returned an aggregated score for the partition is returned, otherwise individual-community ones. Default **True**.

Returns If **summary==True** a FitnessResult object, otherwise a list of floats.

Example:

```
>>> from cdlib.algorithms import louvain
>>> from cdlib import evaluation
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = evaluation.normalized_cut(g,communities)
```

References

1.Shi, J., Malik, J.: Normalized cuts and image segmentation. Departmental Papers (CIS), 107 (2000)

cdlib.evaluation.max_odf

max_odf (*graph*, *community*, ***kwargs*)

Maximum fraction of edges of a node of a community that point outside the community itself.

$$max_{u \in S} \frac{|\{(u,v) \in E : v \notin S\}|}{d(u)}$$

where E is the graph edge set, v is a node in S and $d(u)$ is the degree of u

Parameters

- **graph** – a networkx/igraph object
- **community** – NodeClustering object
- **summary** – boolean. If **True** it is returned an aggregated score for the partition is returned, otherwise individual-community ones. Default **True**.

Returns If **summary==True** a FitnessResult object, otherwise a list of floats.

Example:

```
>>> from cdlib.algorithms import louvain
>>> from cdlib import evaluation
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = evaluation.max_odf(g,communities)
```

References

1. Flake, G.W., Lawrence, S., Giles, C.L., et al.: Efficient identification of web communities. In: KDD, vol. 2000, pp. 150–160 (2000)

cdlib.evaluation.avg_odf

avg_odf (*graph*, *community*, ***kwargs*)

Average fraction of edges of a node of a community that point outside the community itself.

$$\frac{1}{n_S} \sum_{u \in S} \frac{|\{(u,v) \in E : v \notin S\}|}{d(u)}$$

where E is the graph edge set, v is a node in S, $d(u)$ is the degree of u and n_S is the set of community nodes.

Parameters

- **graph** – a networkx/igraph object
- **community** – NodeClustering object
- **summary** – boolean. If **True** it is returned an aggregated score for the partition is returned, otherwise individual-community ones. Default **True**.

Returns If **summary==True** a FitnessResult object, otherwise a list of floats.

Example:

```
>>> from cdlib.algorithms import louvain
>>> from cdlib import evaluation
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = evaluation.avg_odf(g,communities)
```

References

1. Flake, G.W., Lawrence, S., Giles, C.L., et al.: Efficient identification of web communities. In: KDD, vol. 2000, pp. 150–160 (2000)

cdlib.evaluation.flake_odf

flake_odf(*graph*, *community*, ***kwargs*)

Fraction of nodes in S that have fewer edges pointing inside than to the outside of the community.

$$f(S) = \frac{|\{u : u \in S, |\{(u,v) \in E : v \in S\}| < d(u)/2\}|}{n_S}$$

where E is the graph edge set, v is a node in S, $d(u)$ is the degree of u and n_S is the set of community nodes.

Parameters

- **graph** – a networkx/igraph object
- **community** – NodeClustering object
- **summary** – boolean. If **True** it is returned an aggregated score for the partition is returned, otherwise individual-community ones. Default **True**.

Returns If **summary==True** a FitnessResult object, otherwise a list of floats.

Example:

```
>>> from cdlib.algorithms import louvain
>>> from cdlib import evaluation
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = evaluation.flake_odf(g,communities)
```

References

1. Flake, G.W., Lawrence, S., Giles, C.L., et al.: Efficient identification of web communities. In: KDD, vol. 2000, pp. 150–160 (2000)

cdlib.evaluation.scaled_density

scaled_density(*graph*, *communities*, ***kwargs*)

Scaled density.

The scaled density of a community is defined as the ratio of the community density w.r.t. the complete graph density.

Parameters

- **graph** – a networkx/igraph object
- **communities** – NodeClustering object
- **summary** – boolean. If **True** it is returned an aggregated score for the partition is returned, otherwise individual-community ones. Default **True**.

Returns If **summary==True** a FitnessResult object, otherwise a list of floats.

Example:

```
>>> from cdlib.algorithms import louvain
>>> from cdlib import evaluation
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> scd = evaluation.scaled_density(g,communities)
```

cdlib.evaluation.significance

significance(*graph*, *communities*, ***kwargs*)

Significance estimates how likely a partition of dense communities appear in a random graph.

Parameters

- **graph** – a networkx/igraph object
- **communities** – NodeClustering object

Returns FitnessResult object

Example:

```
>>> from cdlib.algorithms import louvain
>>> from cdlib import evaluation
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = evaluation.significance(g,communities)
```

References

1. Traag, V. A., Aldecoa, R., & Delvenne, J. C. (2015). Detecting communities using asymptotical surprise. Physical Review E, 92(2), 022816.

cdlib.evaluation.size

size(*graph*, *communities*, ***kwargs*)

Size is the number of nodes in the community

Parameters

- **graph** – a networkx/igraph object
- **communities** – NodeClustering object
- **summary** – boolean. If **True** it is returned an aggregated score for the partition is returned, otherwise individual-community ones. Default **True**.

Returns If **summary==True** a FitnessResult object, otherwise a list of floats.

Example:

```
>>> from cdlib.algorithms import louvain
>>> from cdlib import evaluation
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> sz = evaluation.size(g,communities)
```

cdlib.evaluation.surprise

surprise(*graph*, *communities*, ***kwargs*)

Surprise is statistical approach proposes a quality metric assuming that edges between vertices emerge randomly according to a hyper-geometric distribution.

According to the Surprise metric, the higher the score of a partition, the less likely it is resulted from a random realization, the better the quality of the community structure.

Parameters

- **graph** – a networkx/igraph object
- **communities** – NodeClustering object

Returns FitnessResult object

Example:

```
>>> from cdlib.algorithms import louvain
>>> from cdlib import evaluation
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = evaluation.surprise(g,communities)
```

References

1. Traag, V. A., Aldecoa, R., & Delvenne, J. C. (2015). Detecting communities using asymptotical surprise. Physical Review E, 92(2), 022816.

cdlib.evaluation.triangle_participation_ratio

triangle_participation_ratio(*graph*, *community*, ***kwargs*)

Fraction of community nodes that belong to a triad.

$$f(S) = \frac{|\{u : u \in S, \{(v, w) : v, w \in S, (u, v) \in E, (u, w) \in E, (v, w) \in E\} \neq \emptyset\}|}{n_S}$$

where n_S is the set of community nodes.

Parameters

- **graph** – a networkx/igraph object
- **community** – NodeClustering object
- **summary** – boolean. If **True** it is returned an aggregated score for the partition is returned, otherwise individual-community ones. Default **True**.

Returns If **summary==True** a FitnessResult object, otherwise a list of floats.

Example:

```
>>> from cdlib.algorithms import louvain
>>> from cdlib import evaluation
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = evaluation.triangle_participation_ratio(g,communities)
```

References

1. Yang, J., Leskovec, J.: Defining and evaluating network communities based on ground-truth. Knowledge and Information Systems 42(1), 181–213 (2015)

cdlib.evaluation.purity

purity(*communities*)

Purity is the product of the frequencies of the most frequent labels carried by the nodes within the communities

Parameters **communities** – AttrNodeClustering object

Returns FitnessResult object

Example:

```
>>> from cdlib.algorithms import eva
>>> from cdlib import evaluation
>>> import random
>>> l1 = ['A', 'B', 'C', 'D']
>>> l2 = ["E", "F", "G"]
>>> g = nx.barabasi_albert_graph(100, 5)
>>> labels=dict()
>>> for node in g.nodes():
>>>    labels[node]={"l1":random.choice(l1), "l2":random.choice(l2)}
>>> communities = eva(g_attr, labels, alpha=0.5)
>>> pur = evaluation.purity(communities)
```

References

1. Citraro, Salvatore, and Giulio Rossetti. “Eva: Attribute-Aware Network Segmentation.” International Conference on Complex Networks and Their Applications. Springer, Cham, 2019.

Among the fitness function a well-defined family of measures is the Modularity-based one:

erdos_renyi_modularity(graph, communities, ...)	Erdos-Renyi modularity is a variation of the Newman-Girvan one.

Continued on next page

Table 23 – continued from previous page

link_modularity(graph, communities, **kwargs)	Quality function designed for directed graphs with overlapping communities.
modularity_density(graph, communities, **kwargs)	The modularity density is one of several propositions that envisioned to palliate the resolution limit issue of modularity based measures.
newman_girvan_modularity(graph, communities, ...)	Difference the fraction of intra community edges of a partition with the expected number of such edges if distributed according to a null model.
z_modularity(graph, communities, **kwargs)	Z-modularity is another variant of the standard modularity proposed to avoid the resolution limit.

cdlib.evaluation.erdos_renyi_modularity

erdos_renyi_modularity (*graph*, *communities*, ***kwargs*)

Erdos-Renyi modularity is a variation of the Newman-Girvan one. It assumes that vertices in a network are connected randomly with a constant probability p.

$$Q(S) = \frac{1}{m}\sum_{c \in S}(m_S \frac{mn_S(n_S 1)}{n(n1)})$$

where m is the number of graph edges, m_S is the number of community edges, l_S is the number of edges from nodes in S to nodes outside S.

Parameters

- **graph** – a networkx/igraph object
- **communities** – NodeClustering object

Returns FitnessResult object

Example:

```
>>> from cdlib.algorithms import louvain
>>> from cdlib import evaluation
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = evaluation.erdos_renyi_modularity(g,communities)
```

References

1. Erdos, P., & Renyi, A. (1959). On random graphs I. Publ. Math. Debrecen, 6, 290-297.

cdlib.evaluation.link_modularity

link_modularity (*graph*, *communities*, ***kwargs*)

Quality function designed for directed graphs with overlapping communities.

Parameters

- **graph** – a networkx/igraph object
- **communities** – NodeClustering object

Returns FitnessResult object

Example:

```
>>> from cdlib.algorithms import louvain
>>> from cdlib import evaluation
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = evaluation.link_modularity(g,communities)
```

References

1. Nicosia, V., Mangioni, G., Carchiolo, V., Malgeri, M.: Extending the definition of modularity to directed graphs with overlapping communities. Journal of Statistical Mechanics: Theory and Experiment 2009(03), 03024 (2009)

cdlib.evaluation.modularity_density

modularity_density(*graph*, *communities*, ***kwargs*)

The modularity density is one of several propositions that envisioned to palliate the resolution limit issue of modularity based measures. The idea of this metric is to include the information about community size into the expected density of community to avoid the negligence of small and dense communities. For each community C in partition S, it uses the average modularity degree calculated by $d(C) = d^{int(C)}d^{ext(C)}$ where $d^{int(C)}$ and $d^{ext(C)}$ are the average internal and external degrees of C respectively to evaluate the fitness of C in its network. Finally, the modularity density can be calculated as follows:

$$Q(S) = \sum_{C \in S} \frac{1}{n_C} (\sum_{i \in C} k_{iC}^{int} - \sum_{i \in C} k_{iC}^{out})$$

where n_C is the number of nodes in C, k_{iC}^{int} is the degree of node i within C and k_{iC}^{out} is the deree of node i outside C.

Parameters

- **graph** – a networkx/igraph object
- **communities** – NodeClustering object

Returns FitnessResult object

Example:

```
>>> from cdlib.algorithms import louvain
>>> from cdlib import evaluation
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = evaluation.modularity_density(g,communities)
```

References

1. Li, Z., Zhang, S., Wang, R. S., Zhang, X. S., & Chen, L. (2008). Quantitative function for community detection. Physical review E, 77(3), 036109.

cdlib.evaluation.newman_girvan_modularity

newman_girvan_modularity(*graph*, *communities*, ***kwargs*)

Difference the fraction of intra community edges of a partition with the expected number of such edges if distributed according to a null model.

In the standard version of modularity, the null model preserves the expected degree sequence of the graph under consideration. In other words, the modularity compares the real network structure with a corresponding one where nodes are connected without any preference about their neighbors.

$$Q(S) = \frac{1}{m}\sum_{c \in S}(m_S - \frac{(2m_S + l_S)^2}{4m})$$

where m is the number of graph edges, m_S is the number of community edges, l_S is the number of edges from nodes in S to nodes outside S.

Parameters

- **graph** – a networkx/igraph object
- **communities** – NodeClustering object

Returns FitnessResult object

Example:

```
>>> from cdlib.algorithms import louvain
>>> from cdlib import evaluation
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = evaluation.newman_girvan_modularity(g,communities)
```

References

1. Newman, M.E.J. & Girvan, M. Finding and evaluating community structure in networks. Physical Review E 69, 26113(2004).

cdlib.evaluation.z_modularity

z_modularity(*graph*, *communities*, ***kwargs*)

Z-modularity is another variant of the standard modularity proposed to avoid the resolution limit. The concept of this version is based on an observation that the difference between the fraction of edges inside communities and the expected number of such edges in a null model should not be considered as the only contribution to the final quality of community structure.

Parameters

- **graph** – a networkx/igraph object
- **communities** – NodeClustering object

Returns FitnessResult object

Example:

```
>>> from cdlib.algorithms import louvain
>>> from cdlib import evaluation
>>> g = nx.karate_club_graph()
>>> communities = louvain(g)
>>> mod = evaluation.z_modularity(g,communities)
```

References

1. Miyauchi, Atsushi, and Yasushi Kawase. Z-score-based modularity for community detection in networks. PloS one 11.1 (2016): e0147805.

Some measures will return an instance of `FitnessResult` that takes together min/max/mean/std values of the computed index.

FitnessResult(min, max, score, std)	

cdlib.evaluation.FitnessResult

class FitnessResult(*min*, *max*, *score*, *std*)

__init__()
Initialize self. See help(type(self)) for accurate signature.

Methods

count	Return number of occurrences of value.
index	Return first index of value.

Attributes

max	Alias for field number 1
min	Alias for field number 0
score	Alias for field number 2
std	Alias for field number 3

Partition Comparisons

It is often useful to compare different graph partition to assess their resemblance (i.e., to perform ground truth testing). `CDlib` implements the following partition comparisons scores:

adjusted_mutual_information(first_partition, ...)	Adjusted Mutual Information between two clusterings.
adjusted_rand_index(first_partition, ...)	Rand index adjusted for chance.
f1(first_partition, second_partition)	Compute the average F1 score of the optimal algorithms matches among the partitions in input.
nf1(first_partition, second_partition)	Compute the Normalized F1 score of the optimal algorithms matches among the partitions in input.
normalized_mutual_information(...)	Normalized Mutual Information between two clusterings.
omega(first_partition, second_partition)	Index of resemblance for overlapping, complete coverage, network clusterings.
overlapping_normalized_mutual_information_LFK(...)	Overlapping Normalized Mutual Information between two clusterings.
overlapping_normalized_mutual_information_MGH(...)	Overlapping Normalized Mutual Information between two clusterings.
variation_of_information(first_partition, ...)	Variation of Information among two nodes partitions.

cdlib.evaluation.adjusted_mutual_information

adjusted_mutual_information (*first_partition*, *second_partition*)

Adjusted Mutual Information between two clusterings.

Adjusted Mutual Information (AMI) is an adjustment of the Mutual Information (MI) score to account for chance. It accounts for the fact that the MI is generally higher for two clusterings with a larger number of clusters, regardless of whether there is actually more information shared. For two clusterings U and V, the AMI is given as:

```
AMI(U, V) = [MI(U, V) - E(MI(U, V))] / [max(H(U), H(V)) - E(MI(U, V))]
```

This metric is independent of the absolute values of the labels: a permutation of the class or cluster label values won't change the score value in any way.

This metric is furthermore symmetric: switching `label_true` with `label_pred` will return the same score value. This can be useful to measure the agreement of two independent label assignments strategies on the same dataset when the real ground truth is not known.

Be mindful that this function is an order of magnitude slower than other metrics, such as the Adjusted Rand Index.

Parameters

- **first_partition** – NodeClustering object
- **second_partition** – NodeClustering object

Returns MatchingResult object

Example

```
>>> from cdlib import evaluation, algorithms
>>> g = nx.karate_club_graph()
>>> louvain_communities = algorithms.louvain(g)
>>> leiden_communities = algorithms.leiden(g)
>>> evaluation.adjusted_mutual_information(louvain_communities,leiden_communities)
```

Reference

1. Vinh, N. X., Epps, J., & Bailey, J. (2010). Information theoretic measures for clusterings comparison: Variants, properties, normalization and correction for chance. Journal of Machine Learning Research, 11(Oct), 2837-2854.

cdlib.evaluation.adjusted_rand_index

adjusted_rand_index (*first_partition*, *second_partition*)

Rand index adjusted for chance.

The Rand Index computes a similarity measure between two clusterings by considering all pairs of samples and counting pairs that are assigned in the same or different clusters in the predicted and true clusterings.

The raw RI score is then "adjusted for chance" into the ARI score using the following scheme:

```
ARI = (RI - Expected_RI) / (max(RI) - Expected_RI)
```

The adjusted Rand index is thus ensured to have a value close to 0.0 for random labeling independently of the number of clusters and samples and exactly 1.0 when the clusterings are identical (up to a permutation).

ARI is a symmetric measure:

```
adjusted_rand_index(a, b) == adjusted_rand_index(b, a)
```

Parameters

- **first_partition** – NodeClustering object
- **second_partition** – NodeClustering object

Returns MatchingResult object

Example

```
>>> from cdlib import evaluation, algorithms
>>> g = nx.karate_club_graph()
>>> louvain_communities = algorithms.louvain(g)
>>> leiden_communities = algorithms.leiden(g)
>>> evaluation.adjusted_rand_index(louvain_communities,leiden_communities)
```

Reference

1. Hubert, L., & Arabie, P. (1985). Comparing partitions. Journal of classification, 2(1), 193-218.

cdlib.evaluation.f1

f1 (*first_partition*, *second_partition*)

Compute the average F1 score of the optimal algorithms matches among the partitions in input. Works on overlapping/non-overlapping complete/partial coverage partitions.

Parameters

- **first_partition** – NodeClustering object
- **second_partition** – NodeClustering object

Returns MatchingResult object

Example

```
>>> from cdlib import evaluation, algorithms
>>> g = nx.karate_club_graph()
>>> louvain_communities = algorithms.louvain(g)
>>> leiden_communities = algorithms.leiden(g)
>>> evaluation.f1(louvain_communities,leiden_communities)
```

Reference

1. Rossetti, G., Pappalardo, L., & Rinzivillo, S. (2016). A novel approach to evaluate algorithms detection internal on ground truth. In Complex Networks VII (pp. 133-144). Springer, Cham.

cdlib.evaluation.nf1

nf1 (*first_partition*, *second_partition*)

Compute the Normalized F1 score of the optimal algorithms matches among the partitions in input. Works on overlapping/non-overlapping complete/partial coverage partitions.

Parameters

- **first_partition** – NodeClustering object
- **second_partition** – NodeClustering object

Returns MatchingResult object

Example

```
>>> from cdlib import evaluation, algorithms
>>> g = nx.karate_club_graph()
>>> louvain_communities = algorithms.louvain(g)
>>> leiden_communities = algorithms.leiden(g)
>>> evaluation.nf1(louvain_communities,leiden_communities)
```

Reference

1. Rossetti, G., Pappalardo, L., & Rinzivillo, S. (2016). A novel approach to evaluate algorithms detection internal on ground truth.
2. Rossetti, G. (2017). : RDyn: graph benchmark handling algorithms dynamics. Journal of Complex Networks. 5(6), 893-912.

cdlib.evaluation.normalized_mutual_information

normalized_mutual_information(*first_partition*, *second_partition*)

Normalized Mutual Information between two clusterings.

Normalized Mutual Information (NMI) is an normalization of the Mutual Information (MI) score to scale the results between 0 (no mutual information) and 1 (perfect correlation). In this function, mutual information is normalized by `sqrt(H(labels_true) * H(labels_pred))`

Parameters

- **first_partition** – NodeClustering object
- **second_partition** – NodeClustering object

Returns MatchingResult object

Example

```
>>> from cdlib import evaluation, algorithms
>>> g = nx.karate_club_graph()
>>> louvain_communities = algorithms.louvain(g)
>>> leiden_communities = algorithms.leiden(g)
>>> evaluation.normalized_mutual_information(louvain_communities,
↪leiden_communities)
```

cdlib.evaluation.omega

omega(*first_partition*, *second_partition*)

Index of resemblance for overlapping, complete coverage, network clusterings.

Parameters

- **first_partition** – NodeClustering object

- **second_partition** – NodeClustering object

Returns MatchingResult object

Example

```
>>> from cdlib import evaluation, algorithms
>>> g = nx.karate_club_graph()
>>> louvain_communities = algorithms.louvain(g)
>>> leiden_communities = algorithms.leiden(g)
>>> evaluation.omega(louvain_communities,leiden_communities)
:Reference:
```

1. Gabriel Murray, Giuseppe Carenini, and Raymond Ng. 2012. Using the omega index for evaluating abstractive algorithms detection. In Proceedings of Workshop on Evaluation Metrics and System Comparison for Automatic Summarization. Association for Computational Linguistics, Stroudsburg, PA, USA, 10-18.

cdlib.evaluation.overlapping_normalized_mutual_information_LFK

overlapping_normalized_mutual_information_LFK(*first_partition*, *second_partition*)

Overlapping Normalized Mutual Information between two clusterings.

Extension of the Normalized Mutual Information (NMI) score to cope with overlapping partitions. This is the version proposed by Lancichinetti et al. (1)

Parameters

- **first_partition** – NodeClustering object
- **second_partition** – NodeClustering object

Returns MatchingResult object

Example

```
>>> from cdlib import evaluation, algorithms
>>> g = nx.karate_club_graph()
>>> louvain_communities = algorithms.louvain(g)
>>> leiden_communities = algorithms.leiden(g)
>>> evaluation.overlapping_normalized_mutual_information_LFK(louvain_communities,
↪leiden_communities)
:Reference:
```

1. Lancichinetti, A., Fortunato, S., & Kertesz, J. (2009). Detecting the overlapping and hierarchical community structure in complex networks. New Journal of Physics, 11(3), 033015.

cdlib.evaluation.overlapping_normalized_mutual_information_MGH

overlapping_normalized_mutual_information_MGH(*first_partition*, *second_partition*, *normalization='max'*)

Overlapping Normalized Mutual Information between two clusterings.

Extension of the Normalized Mutual Information (NMI) score to cope with overlapping partitions. This is the version proposed by McDaid et al. using a different normalization than the original LFR one. See ref. for more details.

Parameters

- **first_partition** – NodeClustering object
- **second_partition** – NodeClustering object
- **normalization** – one of "max" or "LFK". Default "max" (corresponds to the main method described in the article)

Returns MatchingResult object

Example

```
>>> from cdlib import evaluation, algorithms
>>> g = nx.karate_club_graph()
>>> louvain_communities = algorithms.louvain(g)
>>> leiden_communities = algorithms.leiden(g)
>>> evaluation.overlapping_normalized_mutual_information_MGH(louvain_communities,
↪leiden_communities)
:Reference:
```

1. McDaid, A. F., Greene, D., & Hurley, N. (2011). Normalized mutual information to evaluate overlapping community finding algorithms. arXiv preprint arXiv:1110.2515. Chicago

cdlib.evaluation.variation_of_information

variation_of_information (*first_partition*, *second_partition*)

Variation of Information among two nodes partitions.

$$ H(p)+H(q)-2MI(p, q) $$

where MI is the mutual information, H the partition entropy and p,q are the algorithms sets

Parameters

- **first_partition** – NodeClustering object
- **second_partition** – NodeClustering object

Returns MatchingResult object

Example

```
>>> from cdlib import evaluation, algorithms
>>> g = nx.karate_club_graph()
>>> louvain_communities = algorithms.louvain(g)
>>> leiden_communities = algorithms.leiden(g)
>>> evaluation.variation_of_information(louvain_communities,leiden_communities)
```

Reference

1. Meila, M. (2007). Comparing clusterings - an information based distance. Journal of Multivariate Analysis, 98, 873-895. doi:10.1016/j.jmva.2006.11.013

Some measures will return an instance of `MatchingResult` that takes together mean and standard deviation values of the computed index.

MatchingResult(score, std)

cdlib.evaluation.MatchingResult

class MatchingResult(*score*, *std*)

__init__()
Initialize self. See help(type(self)) for accurate signature.

Methods

count	Return number of occurrences of value.
index	Return first index of value.

Attributes

score	Alias for field number 0
std	Alias for field number 1

1.5.5 Input-Output

Functions to save/load CDlib communities to/from file.

CSV format

The easiest way to save the result of a community discovery algorithm is to organize it in a .csv file. The following methods allows to read/write communities to/from csv.

read_community_csv(path[, delimiter, nodetype])	Read community list from comma separated value (csv) file.
write_community_csv(communities, path[, ...])	Save community structure to comma separated value (csv) file.

cdlib.readwrite.read_community_csv

read_community_csv(*path*, *delimiter=', '*, *nodetype=<class 'str'>*)
Read community list from comma separated value (csv) file.

Parameters

- **path** – input filename
- **delimiter** – column delimiter
- **nodetype** – specify the type of node labels, default str

Returns NodeClustering object

Example

```
>>> import networkx as nx
>>> from cdlib import algorithms, readwrite
```

(continues on next page)

(continued from previous page)

```
>>> g = nx.karate_club_graph()
>>> coms = algorithms.louvain(g)
>>> readwrite.write_community_csv(coms, "communities.csv", ",")
>>> coms = readwrite.read_community_csv(coms, "communities.csv", ",", str)
```

cdlib.readwrite.write_community_csv

write_community_csv(*communities*, *path*, *delimiter=', '*)

Save community structure to comma separated value (csv) file.

Parameters

- **communities** – a NodeClustering object
- **path** – output filename
- **delimiter** – column delimiter

Example

```
>>> import networkx as nx
>>> from cdlib import algorithms, readwrite
>>> g = nx.karate_club_graph()
>>> coms = algorithms.louvain(g)
>>> readwrite.write_community_csv(coms, "communities.csv", ",")
```

Note: CSV formatting allows only to save/retrieve NodeClustering object loosing most of the metadata present in the CD computation result - e.g., algorithm name, parameters, coverage...

JSON format

JSON format allows to store/load community discovery algorithm results in a more comprehensive way.

read_community_json(path)	Read community list from JSON file.
write_community_json(communities, path)	Generate a JSON representation of the clustering object

cdlib.readwrite.read_community_json

read_community_json(*path*)

Read community list from JSON file.

Parameters **path** – input filename

Returns a Clustering object

Example

```
>>> import networkx as nx
>>> from cdlib import algorithms, readwrite
>>> g = nx.karate_club_graph()
>>> coms = algorithms.louvain(g)
```

(continues on next page)

(continued from previous page)

```
>>> readwrite.write_community_json(coms, "communities.json")
>>> readwrite.read_community_json(coms, "communities.json")
```

cdlib.readwrite.write_community_json

write_community_json(*communities*, *path*)

Generate a JSON representation of the clustering object

Parameters

- **communities** – a cdlib clustering object
- **path** – output filename

Returns a JSON formatted string representing the object

Example

```
>>> import networkx as nx
>>> from cdlib import algorithms, readwrite
>>> g = nx.karate_club_graph()
>>> coms = algorithms.louvain(g)
>>> readwrite.write_community_json(coms, "communities.json")
```

Note: JSON formatting allows only to save/retrieve all kind of Clustering object maintaining all their metadata - except for the graph object instance.

1.5.6 Visual Analytics

At the end of the analytical process is it often useful to visualize the obtained results. `CDlib` provides a few built-in facilities to ease such task.

Network Visualization

Visualizing a graph is always a good idea (if its size is reasonable).

plot_network_clusters(graph, partition[, ...])	Plot a graph with node color coding for communities.
plot_community_graph(graph, partition[, ...])	Plot a algorithms-graph with node color coding for communities.

cdlib.viz.plot_network_clusters

plot_network_clusters(*graph*, *partition*, *position=None*, *figsize=(8, 8)*, *node_size=200*, *plot_overlaps=False*, *plot_labels=False*, *cmap=None*, *top_k=None*, *min_size=None*)

Plot a graph with node color coding for communities.

Parameters

- **graph** – NetworkX/igraph graph

- **partition** – NodeClustering object
- **position** – A dictionary with nodes as keys and positions as values. Example: networkx.fruchterman_reingold_layout(G). By default, uses nx.spring_layout(g)
- **figsize** – the figure size; it is a pair of float, default (8, 8)
- **node_size** – int, default 200
- **plot_overlaps** – bool, default False. Flag to control if multiple algorithms memberships are plotted.
- **plot_labels** – bool, default False. Flag to control if node labels are plotted.
- **cmap** – str or Matplotlib colormap, Colormap(Matplotlib colormap) for mapping intensities of nodes. If set to None, original colormap is used.
- **top_k** – int, Show the top K influential communities. If set to zero or negative value indicates all.
- **min_size** – int, Exclude communities below the specified minimum size.

Example:

```
>>> from cdlib import algorithms, viz
>>> import networkx as nx
>>> g = nx.karate_club_graph()
>>> coms = algorithms.louvain(g)
>>> pos = nx.spring_layout(g)
>>> viz.plot_network_clusters(g, coms, pos)
```

cdlib.viz.plot_community_graph

plot_community_graph(*graph*, *partition*, *figsize=(8, 8)*, *node_size=200*, *plot_overlaps=False*, *plot_labels=False*, *cmap=None*, *top_k=None*, *min_size=None*)

Plot a algorithms-graph with node color coding for communities.

Parameters

- **graph** – NetworkX/igraph graph
- **partition** – NodeClustering object
- **figsize** – the figure size; it is a pair of float, default (8, 8)
- **node_size** – int, default 200
- **plot_overlaps** – bool, default False. Flag to control if multiple algorithms memberships are plotted.
- **plot_labels** – bool, default False. Flag to control if node labels are plotted.
- **cmap** – str or Matplotlib colormap, Colormap(Matplotlib colormap) for mapping intensities of nodes. If set to None, original colormap is used..
- **top_k** – int, Show the top K influential communities. If set to zero or negative value indicates all.
- **min_size** – int, Exclude communities below the specified minimum size.

Example:

```
>>> from cdlib import algorithms, viz
>>> import networkx as nx
>>> g = nx.karate_club_graph()
>>> coms = algorithms.louvain(g)
>>> viz.plot_community_graph(g, coms)
```

Analytics plots

Community evaluation outputs can be easily used to generate a visual representation of the main partition characteristics.

plot_sim_matrix(clusterings, scoring)	Plot a similarity matrix between a list of clusterings, using the provided scoring function.
plot_com_stat(com_clusters, com_fitness)	Plot the distribution of a property among all communities for a clustering, or a list of clusterings (violin-plots)
plot_com_properties_relation(com_clusters, ...)	Plot the relation between two properties/fitness function of a clustering
plot_scoring(graphs, ref_partitions, ... [, ...])	Plot the scores obtained by a list of methods on a list of graphs.

cdlib.viz.plot_sim_matrix

plot_sim_matrix (*clusterings*, *scoring*)

Plot a similarity matrix between a list of clusterings, using the provided scoring function.

Parameters

- **clusterings** – list of clusterings to compare
- **scoring** – the scoring function to use

Returns a ClusterGrid instance

Example:

```
>>> from cdlib import algorithms, viz, evaluation
>>> import networkx as nx
>>> g = nx.karate_club_graph()
>>> coms = algorithms.louvain(g)
>>> coms2 = algorithms.walktrap(g)
>>> clustermap = viz.plot_sim_matrix([coms,coms2],evaluation.adjusted_mutual_
↪information)
```

cdlib.viz.plot_com_stat

plot_com_stat (*com_clusters*, *com_fitness*)

Plot the distribution of a property among all communities for a clustering, or a list of clusterings (violin-plots)

Parameters

- **com_clusters** – list of clusterings to compare, or a single clustering
- **com_fitness** – the fitness/community property to use

Returns the violin-plots

Example:

```
>>> from cdlib import algorithms, viz, evaluation
>>> import networkx as nx
>>> g = nx.karate_club_graph()
>>> coms = algorithms.louvain(g)
>>> coms2 = algorithms.walktrap(g)
>>> violinplot = viz.plot_com_stat([coms,coms2],evaluation.size)
```

cdlib.viz.plot_com_properties_relation

plot_com_properties_relation(*com_clusters*, *com_fitness_x*, *com_fitness_y*, ***kwargs*)

Plot the relation between two properties/fitness function of a clustering

Parameters

- **com_clusters** – clustering(s) to analyze (cluster or cluster list)
- **com_fitness_x** – first fitness/community property
- **com_fitness_y** – first fitness/community property
- **kwargs** – parameters for the seaborn lmplot

Returns a seaborn lmplot

Example:

```
>>> from cdlib import algorithms, viz, evaluation
>>> import networkx as nx
>>> g = nx.karate_club_graph()
>>> coms = algorithms.louvain(g)
>>> coms2 = algorithms.walktrap(g)
>>> lmplot = viz.plot_com_properties_relation([coms,coms2],evaluation.size,
↪evaluation.internal_edge_density)
```

cdlib.viz.plot_scoring

plot_scoring(*graphs*, *ref_partitions*, *graph_names*, *methods*, *scoring=<function adjusted_mutual_information>*, *nbRuns=5*)

Plot the scores obtained by a list of methods on a list of graphs.

Parameters

- **graphs** – list of graphs on which to make computations
- **ref_partitions** – list of reference clusterings corresponding to graphs
- **graph_names** – list of the names of the graphs to display
- **methods** – list of functions that take a graph as input and return a Clustering as output
- **scoring** – the scoring function to use, default anmi
- **nbRuns** – number of runs to do for each method on each graph

Returns a seaborn lineplot

Example:

```
>>> from cdlib import algorithms, viz, evaluation
>>> import networkx as nx
>>> g1 = nx.algorithms.community.LFR_benchmark_graph(1000, 3, 1.5, 0.5, min_
↪community=20, average_degree=5)
>>> g2 = nx.algorithms.community.LFR_benchmark_graph(1000, 3, 1.5, 0.7, min_
↪community=20, average_degree=5)
>>> names = ["g1", "g2"]
>>> graphs = [g1, g2]
>>> for g in graphs:
>>>     references.append(NodeClustering(communities={frozenset(g.nodes[v][
↪'community']) for v in g}, graph=g, method_name="reference"))
>>> algos = [algorithms.crisp_partition.louvain, algorithms.crisp_partition.label_
↪propagation]
>>> viz.plot_scoring(graphs, references, names, algos, nbRuns=2)
```

1.5.7 Utilities

CDlib exposes a few utilities to manipulate graph objects generated with igraph and networkx.

Graph Transformation

Transform igraph to/from networkx objects.

convert_graph_formats(graph, desired_format)	Converts from/to networkx/igraph

cdlib.utils.convert_graph_formats

convert_graph_formats(*graph*, *desired_format*, *directed=None*)

Converts from/to networkx/igraph

Parameters

- **graph** – original graph object
- **desired_format** – desired final type. Either nx.Graph or ig.Graph
- **directed** – boolean, default **False**

Returns the converted graph

Raises **TypeError** – if input graph is neither an instance of nx.Graph nor ig.Graph

Identifier mapping

Remapping of graph nodes. It is often a good idea - to limit the memory usage - to use progressive integers as node labels. CDlib automatically - and transparently - makes the conversion for the user, however, this step can be costly: for such reason the library also exposes facilities to directly pre/post process the network/community data.

nx_node_integer_mapping(graph)	Maps node labels from strings to integers.
remap_node_communities(communities, node_map)	Apply a map to the obtained communities to retreive the original node labels

cdlib.utils.nx_node_integer_mapping

nx_node_integer_mapping(*graph*)
Maps node labels from strings to integers.

Parameters **graph** – networkx graph

Returns if the node labels are string: networkx graph, dictionary <numeric_id, original_node_label>, false otherwise

cdlib.utils.remap_node_communities

remap_node_communities(*communities*, *node_map*)
Apply a map to the obtained communities to retreive the original node labels

Parameters

- **communities** – NodeClustering object
- **node_map** – dictionary <numeric_id, node_label>

Returns remapped communities

1.6 Developer Guide

1.7 Bibliography

CDlib was developed for research purposes.

Reference algorithms:

- **Crisp Partition:**
  - Girvan-Newman: Girvan, Michelle, and Mark EJ Newman. Community structure in social and biological networks. Proceedings of the national academy of sciences 99.12 (2002): 7821-7826.
  - EM: Newman, Mark EJ, and Elizabeth A. Leicht. Mixture community and exploratory analysis in networks. Proceedings of the National Academy of Sciences 104.23 (2007): 9564-9569.
  - SCAN: Xu, X., Yuruk, N., Feng, Z., & Schweiger, T. A. (2007, August). Scan: a structural clustering algorithm for networks. In Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining (pp. 824-833)
  - GDMP2: Chen, Jie, and Yousef Saad. Dense subgraph extraction with application to community detection. IEEE Transactions on Knowledge and Data Engineering 24.7 (2012): 1216-1230.
  - Spinglass: Reichardt, Jörg, and Stefan Bornholdt. Statistical mechanics of community detection. Physical Review E 74.1 (2006): 016110.
  - Eigenvector: Newman, Mark EJ. Finding community structure in networks using the eigenvectors of matrices. Physical review E 74.3 (2006): 036104.
  - AGDL: Zhang, W., Wang, X., Zhao, D., & Tang, X. (2012, October). Graph degree linkage: Agglomerative clustering on a directed graph. In European Conference on Computer Vision (pp. 428-441). Springer, Berlin, Heidelberg.
  - Louvain: Blondel, Vincent D., et al. Fast unfolding of communities in large networks. Journal of statistical mechanics: theory and experiment 2008.10 (2008): P10008.

  - Leiden: Traag, Vincent, Ludo Waltman, and Nees Jan van Eck. From Louvain to Leiden: guaranteeing well-connected communities. arXiv preprint arXiv:1810.08473 (2018).
  - **Rb_pots:**
    1. Reichardt, J., & Bornholdt, S. (2006). Statistical mechanics of community detection. Physical Review E, 74(1), 016110. 10.1103/PhysRevE.74.016110
    2. Leicht, E. A., & Newman, M. E. J. (2008). Community Structure in Directed Networks. Physical Review Letters, 100(11), 118703. 10.1103/PhysRevLett.100.118703
  - Rber_pots: Reichardt, J., & Bornholdt, S. (2006). Statistical mechanics of community detection. Physical Review E, 74(1), 016110. 10.1103/PhysRevE.74.016110
  - CPM: Traag, V. A., Van Dooren, P., & Nesterov, Y. (2011). Narrow scope for resolution-limit-free community detection. Physical Review E, 84(1), 016114. 10.1103/PhysRevE.84.016114
  - Significance_communities: Traag, V. A., Krings, G., & Van Dooren, P. (2013). Significant scales in community structure. Scientific Reports, 3, 2930. *10.1038/srep02930 <http://doi.org/10.1038/srep02930>*
  - Surprise_communities: Traag, V. A., Aldecoa, R., & Delvenne, J.-C. (2015). Detecting communities using asymptotical surprise. Physical Review E, 92(2), 022816. 10.1103/PhysRevE.92.022816
  - Greedy_modularity: Clauset, A., Newman, M. E., & Moore, C. Finding community structure in very large networks. Physical Review E 70(6), 2004
  - Infomap: Rosvall M, Bergstrom CT (2008) Maps of random walks on complex networks reveal community structure. Proc Natl Acad SciUSA 105(4):1118–1123
  - Walktrap: Pons, Pascal, and Matthieu Latapy. Computing communities in large networks using random walks. J. Graph Algorithms Appl. 10.2 (2006): 191-218.
  - Label_propagation: Raghavan, U. N., Albert, R., & Kumara, S. (2007). Near linear time algorithm to detect community structures in large-scale networks. Physical review E, 76(3), 036106.
  - Async_fluid: Ferran Parés, Dario Garcia-Gasulla, Armand Vilalta, Jonatan Moreno, Eduard Ayguadé, Jesús Labarta, Ulises Cortés, Toyotaro Suzumura T. Fluid Communities: A Competitive and Highly Scalable Community Detection Algorithm.
  - DER: M. Kozdoba and S. Mannor, Community Detection via Measure Space Embedding, NIPS 2015
  - FRC_FGSN: Kundu, S., & Pal, S. K. (2015). Fuzzy-rough community in social networks. Pattern Recognition Letters, 67, 145-152.
  - SBM_dl: Tiago P. Peixoto, Efficient Monte Carlo and greedy heuristic for the inference of stochastic block models , Phys. Rev. E 89, 012804 (2014), DOI: 10.1103/PhysRevE.89.012804 [sci-hub, @tor], arXiv: 1310.4378.
  - SBM_dl_nested: Tiago P. Peixoto, Hierarchical block structures and high-resolution model selection in large networks ,Physical Review X 4.1 (2014): 011047

- **Edge clustering:**
  - hierarchical_link_community: Ahn, Yong-Yeol, James P. Bagrow, and Sune Lehmann. Link communities reveal multiscale complexity in networks. nature 466.7307 (2010): 761.
  - Markov_clustering: Enright, Anton J., Stijn Van Dongen, and Christos A. Ouzounis. An efficient algorithm for large-scale detection of protein families. Nucleic acids research 30.7 (2002): 1575-1584.
- **Overlapping partition:**

- **Demon:**
  1. Coscia, M., Rossetti, G., Giannotti, F., & Pedreschi, D. (2012, August). Demon: a local-first discovery method for overlapping communities. In Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining (pp. 615-623). ACM.
  2. Coscia, M., Rossetti, G., Giannotti, F., & Pedreschi, D. (2014). Uncovering hierarchical and overlapping communities with a local-first approach. ACM Transactions on Knowledge Discovery from Data (TKDD), 9(1), 6.
- Angel: Rossetti, G. (2019) Exorcising the Demon: Angel, Efficient Node-Centric Community Discovery. International Conference on Complex Networks and Their Applications. Springer, Cham.
- Node_perception: Sucheta Soundarajan and John E. Hopcroft. 2015. Use of Local Group Information to Identify Communities in Networks. ACM Trans. Knowl. Discov. Data 9, 3, Article 21 (April 2015), 27 pages. DOI=http://dx.doi.org/10.1145/2700404
- Overlapping_seed_set_expansion: Whang, J. J., Gleich, D. F., & Dhillon, I. S. (2013, October). Overlapping community detection using seed set expansion. In Proceedings of the 22nd ACM international conference on Conference on information & knowledge management (pp. 2099-2108). ACM.
- Kclique: Gergely Palla, Imre Derényi, Illés Farkas1, and Tamás Vicsek, Uncovering the overlapping community structure of complex networks in nature and society Nature 435, 814-818, 2005, doi:10.1038/nature03607
- LFM: Lancichinetti, Andrea, Santo Fortunato, and János Kertész. Detecting the overlapping and hierarchical community structure in complex networks New Journal of Physics 11.3 (2009): 033015.
- Lais2: Baumes, Jeffrey, Mark Goldberg, and Malik Magdon-Ismail. Efficient identification of overlapping communities. International Conference on Intelligence and Security Informatics. Springer, Berlin, Heidelberg, 2005.
- Congo: Gregory, Steve. A fast algorithm to find overlapping communities in networks. Joint European Conference on Machine Learning and Knowledge Discovery in Databases. Springer, Berlin, Heidelberg, 2008.
- Conga: Gregory, Steve. An algorithm to find overlapping community structure in networks. European Conference on Principles of Data Mining and Knowledge Discovery. Springer, Berlin, Heidelberg, 2007.
- Lemon: Yixuan Li, Kun He, David Bindel, John Hopcroft Uncovering the small community structure in large networks: A local spectral approach. Proceedings of the 24th international conference on world wide web. International World Wide Web Conferences Steering Committee, 2015.
- SLPA: Xie Jierui, Boleslaw K. Szymanski, and Xiaoming Liu. Slpa: Uncovering overlapping communities in social networks via a speaker-listener interaction dynamic process. Data Mining Workshops (ICDMW), 2011 IEEE 11th International Conference on. IEEE, 2011.
- Multicom: Hollocou, Alexandre, Thomas Bonald, and Marc Lelarge. Multiple Local Community Detection. ACM SIGMETRICS Performance Evaluation Review 45.2 (2018): 76-83.
- Big_clam: Yang, J., & Leskovec, J. (2013, February). Overlapping community detection at scale: a nonnegative matrix factorization approach. In Proceedings of the sixth ACM international conference on Web search and data mining (pp. 587-596). ACM.

Reference evaluation:

- **Comparison:**

- Omega: Gabriel Murray, Giuseppe Carenini, and Raymond Ng. 2012. Using the omega index for evaluating abstractive algorithms detection. In Proceedings of Workshop on Evaluation Metrics and System Comparison for Automatic Summarization. Association for Computational Linguistics, Stroudsburg, PA, USA, 10-18.
- f1: Rossetti, G., Pappalardo, L., & Rinzivillo, S. (2016). A novel approach to evaluate algorithms detection internal on ground truth. In Complex Networks VII (pp. 133-144). Springer, Cham.
- **nf1:**
  1. Rossetti, G., Pappalardo, L., & Rinzivillo, S. (2016). A novel approach to evaluate algorithms detection internal on ground truth.
  2. Rossetti, G. (2017). : RDyn: graph benchmark handling algorithms dynamics. Journal of Complex Networks. 5(6), 893-912.
- Adjusted_rand_index: Hubert, L., & Arabie, P. (1985). Comparing partitions. Journal of classification, 2(1), 193-218.
- Adjusted_mutual_information: Vinh, N. X., Epps, J., & Bailey, J. (2010). Information theoretic measures for clusterings comparison: Variants, properties, normalization and correction for chance. Journal of Machine Learning Research, 11(Oct), 2837-2854.
- Variation_of_information: Meila, M. (2007). Comparing clusterings - an information based distance. Journal of Multivariate Analysis, 98, 873-895. doi:10.1016/j.jmva.2006.11.013
- Overlapping_normalized_mutual_information_MGH: McDaid, A. F., Greene, D., & Hurley, N. (2011). Normalized mutual information to evaluate overlapping community finding algorithms.. arXiv preprint arXiv:1110.2515. Chicago
- Overlapping_normalized_mutual_information_LFK: Lancichinetti, A., Fortunato, S., & Kertesz, J. (2009). Detecting the overlapping and hierarchical community structure in complex networks. New Journal of Physics, 11(3), 033015.

- **Fitness:**
  - Newman_girvan_modularity: Newman, M.E.J. & Girvan, M. Finding and evaluating algorithms structure in networks. Physical Review E 69, 26113(2004).
  - Erdos_renyi_modularity: Erdos, P., & Renyi, A. (1959). On random graphs I. Publ. Math. Debrecen, 6, 290-297.
  - Modularity_density: Li, Z., Zhang, S., Wang, R. S., Zhang, X. S., & Chen, L. (2008). Quantitative function for algorithms detection. Physical review E, 77(3), 036109.
  - Z_modularity: Miyauchi, Atsushi, and Yasushi Kawase. Z-score-based modularity for algorithms detection in networks. PloS one 11.1 (2016): e0147805.
  - Surprise & Significance: Traag, V. A., Aldecoa, R., & Delvenne, J. C. (2015). Detecting communities using asymptotical surprise .. Physical Review E, 92(2), 022816.
  - average_internal_degree: Radicchi, F., Castellano, C., Cecconi, F., Loreto, V., & Parisi, D. (2004). Defining and identifying communities in networks. Proceedings of the National Academy of Sciences, 101(9), 2658-2663.
  - conductance: Shi, J., Malik, J.: Normalized cuts and image segmentation. Departmental Papers (CIS), 107 (2000)
  - cut_ratio: Fortunato, S.: Community detection in graphs. Physics reports 486(3-5), 75–174 (2010)
  - edges_inside: Radicchi, F., Castellano, C., Cecconi, F., Loreto, V., & Parisi, D. (2004). Defining and identifying communities in networks. Proceedings of the National Academy of Sciences, 101(9), 2658-2663.

- expansion: Radicchi, F., Castellano, C., Cecconi, F., Loreto, V., & Parisi, D. (2004). Defining and identifying communities in networks. Proceedings of the National Academy of Sciences, 101(9), 2658-2663.
- internal_edge_density: Radicchi, F., Castellano, C., Cecconi, F., Loreto, V., & Parisi, D. (2004). Defining and identifying communities in networks. Proceedings of the National Academy of Sciences, 101(9), 2658-2663.
- normalized_cut: Shi, J., Malik, J.: Normalized cuts and image segmentation. Departmental Papers (CIS), 107 (2000)
- fraction_over_median_degree: Yang, J and Leskovec, J.: Defining and evaluating network communities based on ground-truth. Knowledge and Information Systems 42(1), 181–213 (2015)
- max_odf: Flake, G.W., Lawrence, S., Giles, C.L., et al.: Efficient identification of web communities. In: KDD, vol. 2000, pp. 150–160 (2000)
- avg_odf: Flake, G.W., Lawrence, S., Giles, C.L., et al.: Efficient identification of web communities. In: KDD, vol. 2000, pp. 150–160 (2000)
- flake_odf: Flake, G.W., Lawrence, S., Giles, C.L., et al.: Efficient identification of web communities. In: KDD, vol. 2000, pp. 150–160 (2000)
- triangle_participation_ratio: Yang, J and Leskovec, J.: Defining and evaluating network communities based on ground-truth. Knowledge and Information Systems 42(1), 181–213 (2015)
- link_modularity: Nicosia, V., Mangioni, G., Carchiolo, V., Malgeri, M.: Extending the definition of modularity to directed graphs with overlapping communities. Journal of Statistical Mechanics: Theory and Experiment 2009(03), 03024 (2009)

So far it has been used as support to the following publications:

- Hubert, M. Master Thesis. (2020) Crawling and Analysing code review networks on industry and open source data
- Pister, A., Buono, P., Fekete, J. D., Plaisant, C., & Valdivia, P. (2020). Integrating Prior Knowledge in Mixed Initiative Social Network Clustering. arXiv preprint arXiv:2005.02972.
- Mohammadmosaferi, K. K., & Naderi, H. (2020). Evolution of communities in dynamic social networks: An efficient map-based approach. Expert Systems with Applications, 147, 113221.

Python Module Index

C

Index

Symbols

A

B

C

D

N

O

P

R

S

T

V

W

Z