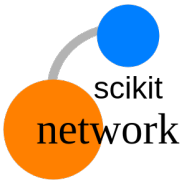
scikit-network Documentation

Release 0.26.0

Bertrand Charpentier

Sep 29, 2022

GETTING STARTED



Python package for the analysis of large graphs:

- Memory-efficient representation as sparse matrices in the CSR format of scipy
- Fast algorithms
- Simple API inspired by scikit-learn

CHAPTER

ONE

RESOURCES

- Free software: BSD license
- GitHub: https://github.com/sknetwork-team/scikit-network
- Documentation: https://scikit-network.readthedocs.io

CHAPTER

TWO

QUICK START

Install scikit-network:

```
$ pip install scikit-network
```

Import scikit-network:

```
import sknetwork
```

See our tutorials; the notebooks are available here.

You can also have a look at some use cases.

CHAPTER

THREE

CITING

If you want to cite scikit-network, please refer to the publication in the Journal of Machine Learning Research:

```
@article{JMLR:v21:20-412,
  author  = {Thomas Bonald and Nathan de Lara and Quentin Lutz and Bertrand Charpentier},
  title   = {Scikit-network: Graph Analysis in Python},
  journal = {Journal of Machine Learning Research},
  year    = {2020},
  volume  = {21},
  number  = {185},
  pages   = {1-6},
  url     = {http://jmlr.org/papers/v21/20-412.html}
}
```

`scikit-network` is an open-source python package for the analysis of large graphs.

Each graph is represented by a sparse `scipy CSR matrix`.

3.1 Install

To install `scikit-network`, run this command in your terminal:

```
$ pip install scikit-network
```

If you don't have pip installed, this Python installation guide can guide you through the process.

Alternately, you can download the sources from the Github repo and run:

```
$ cd <scikit-network folder>
$ python setup.py develop
```

3.2 Import

Import `scikit-network` in Python:

```
import sknetwork as skn
```

3.3 Load

A graph is represented by its *adjacency* matrix (square matrix). When the graph is bipartite, it can be represented by its *biadjacency* matrix (rectangular matrix).

Check our *tutorial* for various ways of loading a graph (from a list of edges, a dataframe or a CSV file, for instance).

3.4 Fit

Each algorithm is represented as an object with a `fit` method.

Here is an example to cluster the Karate club graph with the Louvain algorithm:

```
from sknetwork.data import karate_club
from sknetwork.clustering import Louvain

adjacency = karate_club()
algo = Louvain()
algo.fit(adjacency)
labels = algo.labels_
```

3.5 Data

Tools for loading and saving graphs.

3.5.1 Edge list

sknetwork.data.**from_edge_list**(*edge_list: Union[numpy.ndarray, List[Tuple]]*, *directed: bool = False*, *bipartite: bool = False*, *weighted: bool = True*, *reindex: bool = True*, *sum_duplicates: bool = True*, *matrix_only: Optional[bool] = None*) → Union[sknetwork.utils.Bunch, scipy.sparse.csr.csr_matrix]

Load a graph from an edge list.

Parameters

- **edge_list** (*Union[np.ndarray, List[Tuple]]*) – The edge list to convert, given as a NumPy array of size (n, 2) or (n, 3) or a list of tuples of length 2 or 3.
- **directed** (*bool*) – If `True`, considers the graph as directed.
- **bipartite** (*bool*) – If `True`, returns a biadjacency matrix.
- **weighted** (*bool*) – If `True`, returns a weighted graph.
- **reindex** (*bool*) – If `True`, reindex nodes and returns the original node indices as names. Reindexing is enforced if nodes are not integers.
- **sum_duplicates** (*bool*) – If `True` (default), sums weights of duplicate edges. Otherwise, the weight of each edge is that of the first occurrence of this edge.
- **matrix_only** (*bool*) – If `True`, returns only the adjacency or biadjacency matrix. Otherwise, returns a `Bunch` object with graph attributes (e.g., node names). If not specified (default), selects the most appropriate format.

Returns graph

Return type Bunch (including node names) or sparse matrix

Examples

```
>>> edges = [(0, 1), (1, 2), (2, 0)]
>>> adjacency = from_edge_list(edges)
>>> adjacency.shape
(3, 3)
>>> edges = [('Alice', 'Bob'), ('Bob', 'Carol'), ('Carol', 'Alice')]
>>> graph = from_edge_list(edges)
>>> adjacency = graph.adjacency
>>> adjacency.shape
(3, 3)
>>> print(graph.names)
['Alice' 'Bob' 'Carol']
```

3.5.2 Adjacency list

sknetwork.data.**from_adjacency_list**(*adjacency_list: Union[List[List], Dict[str, List]]*, *directed: bool = False*, *bipartite: bool = False*, *weighted: bool = True*, *reindex: bool = True*, *sum_duplicates: bool = True*, *matrix_only: Optional[bool] = None*) → Union[sknetwork.utils.Bunch, scipy.sparse.csr.csr_matrix]

Load a graph from an adjacency list.

Parameters

- **adjacency_list** (*Union[List[List], Dict[str, List]]*) – Adjacency list (neighbors of each node) or dictionary (node: neighbors).
- **directed** (*bool*) – If True, considers the graph as directed.
- **bipartite** (*bool*) – If True, returns a biadjacency matrix.
- **weighted** (*bool*) – If True, returns a weighted graph.
- **reindex** (*bool*) – If True, reindex nodes and returns the original node indices as names. Reindexing is enforced if nodes are not integers.
- **sum_duplicates** (*bool*) – If True (default), sums weights of duplicate edges. Otherwise, the weight of each edge is that of the first occurrence of this edge.
- **matrix_only** (*bool*) – If True, returns only the adjacency or biadjacency matrix. Otherwise, returns a Bunch object with graph attributes (e.g., node names). If not specified (default), selects the most appropriate format.

Returns graph

Return type Bunch or sparse matrix

Example

```
>>> edges = [(0, 1), (1, 2), (2, 0)]
>>> adjacency = from_edge_list(edges)
>>> adjacency.shape
(3, 3)
```

3.5.3 Files

Check the *tutorial* for importing graphs from dataframes.

sknetwork.data.**from_csv**(*file_path: str*, *delimiter: Optional[str] = None*, *sep: Optional[str] = None*, *comments: tuple = ('#', '%')*, *data_structure: Optional[str] = None*, *directed: bool = False*, *bipartite: bool = False*, *weighted: bool = True*, *reindex: bool = True*, *sum_duplicates: bool = True*, *matrix_only: Optional[bool] = None*) → Union[sknetwork.utils.Bunch, scipy.sparse.csr.csr_matrix]

Load a graph from a CSV or TSV file. The delimiter can be specified (e.g., ' ' for space-separated values).

Parameters

- **file_path** (*str*) – Path to the CSV file.
- **delimiter** (*str*) – Delimiter used in the file. Guessed if not specified.
- **sep** (*str*) – Alias for delimiter.
- **comments** (*str*) – Characters for comment lines.
- **data_structure** (*str*) – If 'edge_list', considers each row of the file as an edge (tuple of size 2 or 3). If 'adjacency_list', considers each row of the file as an adjacency list (list of neighbors). If 'adjacency_dict', considers each row of the file as an adjacency dictionary with key given by the first column (node: list of neighbors). If `None` (default), data_structure is guessed from the first rows of the file.
- **directed** (*bool*) – If `True`, considers the graph as directed.
- **bipartite** (*bool*) – If `True`, returns a biadjacency matrix of shape (n1, n2).
- **weighted** (*bool*) – If `True`, returns a weighted graph (e.g., counts the number of occurrences of each edge).
- **reindex** (*bool*) – If `True`, reindex nodes and returns the original node indices as names. Reindexing is enforced if nodes are not integers.
- **sum_duplicates** (*bool*) – If `True` (default), sums weights of duplicate edges. Otherwise, the weight of each edge is that of the first occurrence of this edge.
- **matrix_only** (*bool*) – If `True`, returns only the adjacency or biadjacency matrix. Otherwise, returns a `Bunch` object with graph attributes (e.g., node names). If not specified (default), selects the most appropriate format.

Returns graph

Return type `Bunch` or sparse matrix

sknetwork.data.**from_graphml**(*file_path: str*, *weight_key: str = 'weight'*, *max_string_size: int = 512*) → sknetwork.utils.Bunch

Load graph from GraphML file.

Hyperedges and nested graphs are not supported.

Parameters

- **file_path** (*str*) – Path to the GraphML file.
- **weight_key** (*str*) – The key to be used as a value for edge weights
- **max_string_size** (*int*) – The maximum size for string features of the data

Returns **data** – The dataset in a bunch with the adjacency as a CSR matrix.

Return type Bunch

3.5.4 Datasets

sknetwork.data.**load_netset**(*name: Optional[str] = None*, *data_home: Optional[Union[str, pathlib.Path]] = None*, *verbose: bool = True*) → sknetwork.utils.Bunch

Load a dataset from the NetSet database.

Parameters

- **name** (*str*) – Name of the dataset (all low-case). Examples include 'openflights', 'cinema' and 'wikivitals'.
- **data_home** (str or pathlib.Path) – Folder to be used for dataset storage.
- **verbose** (*bool*) – Enable verbosity.

Returns **dataset** – Returned dataset.

Return type Bunch

sknetwork.data.**load_konect**(*name: str*, *data_home: Optional[Union[str, pathlib.Path]] = None*, *auto_numpy_bundle: bool = True*, *verbose: bool = True*) → sknetwork.utils.Bunch

Load a dataset from the Konect database.

Parameters

- **name** (*str*) – Name of the dataset as specified on the Konect website (e.g. for the Zachary Karate club dataset, the corresponding name is 'ucidata-zachary').
- **data_home** (str or pathlib.Path) – Folder to be used for dataset storage.
- **auto_numpy_bundle** (*bool*) – Whether the dataset should be stored in its default format (False) or using Numpy files for faster subsequent access to the dataset (True).
- **verbose** (*bool*) – Enable verbosity.

Returns

dataset –

Object with the following attributes:

- *adjacency* or *biadjacency*: the adjacency/biadjacency matrix for the dataset
- *meta*: a dictionary containing the metadata as specified by Konect
- each attribute specified by Konect (ent.* file)

Return type Bunch

Notes

An attribute *meta* of the *Bunch* class is used to store information about the dataset if present. In any case, *meta* has the attribute *name* which, if not given, is equal to the name of the dataset as passed to this function.

References

Kunegis, J. (2013, May). Konect: the Koblenz network collection. In Proceedings of the 22nd International Conference on World Wide Web (pp. 1343-1350).

You can also find some datasets on NetRep.

3.5.5 Toy graphs

sknetwork.data.**house**(*metadata: bool = False*) → Union[scipy.sparse.csr.csr_matrix, sknetwork.utils.Bunch]
House graph.

- Undirected graph
- 5 nodes, 6 edges

Parameters **metadata** – If True, return a *Bunch* object with metadata.

Returns **adjacency or graph** – Adjacency matrix or graph with metadata (positions).

Return type Union[sparse.csr_matrix, Bunch]

Example

```
>>> from sknetwork.data import house
>>> adjacency = house()
>>> adjacency.shape
(5, 5)
```

sknetwork.data.**bow_tie**(*metadata: bool = False*) → Union[scipy.sparse.csr.csr_matrix, sknetwork.utils.Bunch]
Bow tie graph.

- Undirected graph
- 5 nodes, 6 edges

Parameters **metadata** – If True, return a *Bunch* object with metadata.

Returns **adjacency or graph** – Adjacency matrix or graph with metadata (positions).

Return type Union[sparse.csr_matrix, Bunch]

Example

```
>>> from sknetwork.data import bow_tie
>>> adjacency = bow_tie()
>>> adjacency.shape
(5, 5)
```

sknetwork.data.**karate_club**(*metadata: bool = False*) → Union[scipy.sparse.csr.csr_matrix, sknetwork.utils.Bunch]

Karate club graph.

- Undirected graph
- 34 nodes, 78 edges
- 2 labels

Parameters **metadata** – If True, return a *Bunch* object with metadata.

Returns **adjacency or graph** – Adjacency matrix or graph with metadata (labels, positions).

Return type Union[sparse.csr_matrix, Bunch]

Example

```
>>> from sknetwork.data import karate_club
>>> adjacency = karate_club()
>>> adjacency.shape
(34, 34)
```

References

Zachary's karate club graph https://en.wikipedia.org/wiki/Zachary%27s_karate_club

sknetwork.data.**miserables**(*metadata: bool = False*) → Union[scipy.sparse.csr.csr_matrix, sknetwork.utils.Bunch]

Co-occurrence graph of the characters in the novel Les miserables by Victor Hugo.

- Undirected graph
- 77 nodes, 508 edges
- Names of characters

Parameters **metadata** – If True, return a *Bunch* object with metadata.

Returns **adjacency or graph** – Adjacency matrix or graph with metadata (names, positions).

Return type Union[sparse.csr_matrix, Bunch]

Example

```
>>> from sknetwork.data import miserables
>>> adjacency = miserables()
>>> adjacency.shape
(77, 77)
```

sknetwork.data.**painters**(*metadata: bool = False*) → Union[scipy.sparse.csr.csr_matrix, sknetwork.utils.Bunch]

Graph of links between some famous painters on Wikipedia.

- Directed graph
- 14 nodes, 50 edges
- Names of painters

Parameters **metadata** – If True, return a *Bunch* object with metadata.

Returns **adjacency or graph** – Adjacency matrix or graph with metadata (names, positions).

Return type Union[sparse.csr_matrix, Bunch]

Example

```
>>> from sknetwork.data import painters
>>> adjacency = painters()
>>> adjacency.shape
(14, 14)
```

sknetwork.data.**star_wars**(*metadata: bool = False*) → Union[scipy.sparse.csr.csr_matrix, sknetwork.utils.Bunch]

Bipartite graph connecting some Star Wars villains to the movies in which they appear.

- Bipartite graph
- 7 nodes (4 villains, 3 movies), 8 edges
- Names of villains and movies

Parameters **metadata** – If True, return a *Bunch* object with metadata.

Returns **biadjacency or graph** – Biadjacency matrix or graph with metadata (names).

Return type Union[sparse.csr_matrix, Bunch]

Example

```
>>> from sknetwork.data import star_wars
>>> biadjacency = star_wars()
>>> biadjacency.shape
(4, 3)
```

sknetwork.data.**movie_actor**(*metadata: bool = False*) → Union[scipy.sparse.csr.csr_matrix, sknetwork.utils.Bunch]

Bipartite graph connecting movies to some actors starring in them.

- Bipartite graph
- 31 nodes (15 movies, 16 actors), 42 edges
- 9 labels (rows)
- Names of movies (rows) and actors (columns)
- Names of movies production company (rows)

Parameters **metadata** – If `True`, return a *Bunch* object with metadata.

Returns **biadjacency or graph** – Biadjacency matrix or graph with metadata (names).

Return type Union[sparse.csr_matrix, Bunch]

Example

```
>>> from sknetwork.data import movie_actor
>>> biadjacency = movie_actor()
>>> biadjacency.shape
(15, 16)
```

3.5.6 Models

sknetwork.data.**linear_graph**(*n: int = 3*, *metadata: bool = False*) → Union[scipy.sparse.csr.csr_matrix, sknetwork.utils.Bunch]

Linear graph (undirected).

Parameters

- **n** (*int*) – Number of nodes.
- **metadata** (*bool*) – If `True`, return a *Bunch* object with metadata.

Returns **adjacency or graph** – Adjacency matrix or graph with metadata (positions).

Return type Union[sparse.csr_matrix, Bunch]

Example

```
>>> from sknetwork.data import linear_graph
>>> adjacency = linear_graph(5)
>>> adjacency.shape
(5, 5)
```

sknetwork.data.**linear_digraph**(*n: int = 3*, *metadata: bool = False*) → Union[scipy.sparse.csr.csr_matrix, sknetwork.utils.Bunch]

Linear graph (directed).

Parameters

- **n** (*int*) – Number of nodes.
- **metadata** (*bool*) – If `True`, return a *Bunch* object with metadata.

Returns **adjacency or graph** – Adjacency matrix or graph with metadata (positions).

Return type Union[sparse.csr_matrix, Bunch]

Example

```
>>> from sknetwork.data import linear_digraph
>>> adjacency = linear_digraph(5)
>>> adjacency.shape
(5, 5)
```

sknetwork.data.**cyclic_graph**(*n: int = 3*, *metadata: bool = False*) → Union[scipy.sparse.csr.csr_matrix, sknetwork.utils.Bunch]

Cyclic graph (undirected).

Parameters

- **n** (*int*) – Number of nodes.
- **metadata** (*bool*) – If `True`, return a *Bunch* object with metadata.

Returns **adjacency or graph** – Adjacency matrix or graph with metadata (positions).

Return type Union[sparse.csr_matrix, Bunch]

Example

```
>>> from sknetwork.data import cyclic_graph
>>> adjacency = cyclic_graph(5)
>>> adjacency.shape
(5, 5)
```

sknetwork.data.**cyclic_digraph**(*n: int = 3*, *metadata: bool = False*) → Union[scipy.sparse.csr.csr_matrix, sknetwork.utils.Bunch]

Cyclic graph (directed).

Parameters

- **n** (*int*) – Number of nodes.
- **metadata** (*bool*) – If `True`, return a *Bunch* object with metadata.

Returns **adjacency or graph** – Adjacency matrix or graph with metadata (positions).

Return type Union[sparse.csr_matrix, Bunch]

Example

```
>>> from sknetwork.data import cyclic_digraph
>>> adjacency = cyclic_digraph(5)
>>> adjacency.shape
(5, 5)
```

sknetwork.data.**grid**(*n1: int = 10*, *n2: int = 10*, *metadata: bool = False*) → Union[scipy.sparse.csr.csr_matrix, sknetwork.utils.Bunch]

Grid (undirected).

Parameters

- **n1** (*int*) – Grid dimension.
- **n2** (*int*) – Grid dimension.

- **metadata** (*bool*) – If `True`, return a *Bunch* object with metadata.

Returns **adjacency or graph** – Adjacency matrix or graph with metadata (positions).

Return type Union[sparse.csr_matrix, Bunch]

Example

```
>>> from sknetwork.data import grid
>>> adjacency = grid(10, 5)
>>> adjacency.shape
(50, 50)
```

sknetwork.data.**erdos_renyi**(*n: int = 20*, *p: float = 0.3*, *directed: bool = False*, *self_loops: bool = False*, *seed: Optional[int] = None*) → scipy.sparse.csr.csr_matrix

Erdos-Renyi graph.

Parameters

- **n** – Number of nodes.
- **p** – Probability of connection between nodes.
- **directed** – If `True`, return a directed graph.
- **self_loops** – If `True`, allow self-loops.
- **seed** – Seed of the random generator (optional).

Returns **adjacency** – Adjacency matrix.

Return type sparse.csr_matrix

Example

```
>>> from sknetwork.data import erdos_renyi
>>> adjacency = erdos_renyi(7)
>>> adjacency.shape
(7, 7)
```

References

Erdős, P., Rényi, A. (1959). On Random Graphs. Publicationes Mathematicae.

sknetwork.data.**block_model**(*sizes: Iterable*, *p_in: Union[float, list, numpy.ndarray] = 0.2*, *p_out: float = 0.05*, *directed: bool = False*, *self_loops: bool = False*, *metadata: bool = False*, *seed: Optional[int] = None*) → Union[scipy.sparse.csr.csr_matrix, sknetwork.utils.Bunch]

Stochastic block model.

Parameters

- **sizes** – Block sizes.
- **p_in** – Probability of connection within blocks.
- **p_out** – Probability of connection across blocks.
- **directed** – If `True`, return a directed graph.

- **self_loops** – If True, allow self-loops.
- **metadata** – If True, return a *Bunch* object with labels.
- **seed** – Seed of the random generator (optional).

Returns **adjacency or graph** – Adjacency matrix or graph with metadata (labels).

Return type Union[sparse.csr_matrix, Bunch]

Example

```
>>> from sknetwork.data import block_model
>>> sizes = np.array([4, 5])
>>> adjacency = block_model(sizes)
>>> adjacency.shape
(9, 9)
```

References

Airoldi, E., Blei, D., Feinberg, S., Xing, E. (2007). Mixed membership stochastic blockmodels. Journal of Machine Learning Research.

sknetwork.data.**albert_barabasi**(*n: int = 100*, *degree: int = 3*, *directed: bool = False*, *seed: Optional[int] = None*) → scipy.sparse.csr.csr_matrix

Albert-Barabasi model.

Parameters

- **n** (*int*) – Number of nodes.
- **degree** (*int*) – Degree of incoming nodes (less than **n**).
- **directed** (*bool*) – If True, return a directed graph.
- **seed** – Seed of the random generator (optional).

Returns **adjacency** – Adjacency matrix.

Return type sparse.csr_matrix

Example

```
>>> from sknetwork.data import albert_barabasi
>>> adjacency = albert_barabasi(30, 3)
>>> adjacency.shape
(30, 30)
```

References

Albert, R., Barabási, L. (2002). Statistical mechanics of complex networks Reviews of Modern Physics.

sknetwork.data.**watts_strogatz**(*n: int = 100*, *degree: int = 6*, *prob: float = 0.05*, *seed: Optional[int] = None*, *metadata: bool = False*) → Union[scipy.sparse.csr.csr_matrix, sknetwork.utils.Bunch]

Watts-Strogatz model.

Parameters

- **n** – Number of nodes.
- **degree** – Initial degree of nodes.
- **prob** – Probability of edge modification.
- **seed** – Seed of the random generator (optional).
- **metadata** – If `True`, return a *Bunch* object with metadata.

Returns **adjacency or graph** – Adjacency matrix or graph with metadata (positions).

Return type Union[sparse.csr_matrix, Bunch]

Example

```
>>> from sknetwork.data import watts_strogatz
>>> adjacency = watts_strogatz(30, 4, 0.02)
>>> adjacency.shape
(30, 30)
```

References

Watts, D., Strogatz, S. (1998). Collective dynamics of small-world networks, Nature.

3.5.7 Save

sknetwork.data.**save**(*folder: Union[str, pathlib.Path]*, *data: Union[scipy.sparse.csr.csr_matrix, sknetwork.utils.Bunch]*)

Save a Bunch or a CSR matrix in the current directory to a collection of Numpy and Pickle files for faster subsequent loads. Supported attribute types include sparse matrices, NumPy arrays, strings and Bunch.

Parameters

- **folder** (str or `pathlib.Path`) – Name of the bundle folder.
- **data** (*`Union[sparse.csr_matrix, Bunch]`*) – Data to save.

Example

```
>>> from sknetwork.data import save
>>> my_dataset = Bunch()
>>> my_dataset.adjacency = sparse.csr_matrix(np.random.random((3, 3)) < 0.5)
>>> my_dataset.names = np.array(['a', 'b', 'c'])
>>> save('my_dataset', my_dataset)
>>> 'my_dataset' in listdir('.')
True
```

sknetwork.data.**load**(*folder: Union[str, pathlib.Path]*)

Load a Bunch from a previously created bundle from the current directory (inverse function of `save`).

Parameters **folder** (*str*) – Name of the bundle folder.

Returns **data** – Data.

Return type Bunch

Example

```
>>> from sknetwork.data import save
>>> my_dataset = Bunch()
>>> my_dataset.adjacency = sparse.csr_matrix(np.random.random((3, 3)) < 0.5)
>>> my_dataset.names = np.array(['a', 'b', 'c'])
>>> save('my_dataset', my_dataset)
>>> loaded_graph = load('my_dataset')
>>> loaded_graph.names[0]
'a'
```

3.6 Topology

Algorithms for the analysis of graph topology.

3.6.1 Connectivity

sknetwork.topology.**get_connected_components**(*input_matrix: scipy.sparse.csr.csr_matrix*, *connection: str = 'weak'*, *force_bipartite: bool = False*) → numpy.ndarray

Extract the connected components of a graph.

Parameters

- **input_matrix** – Input matrix (either the adjacency matrix or the biadjacency matrix of the graph).
- **connection** – Must be `'weak'` (default) or `'strong'`. The type of connection to use for directed graphs.
- **force_bipartite** (*bool*) – If `True`, consider the input matrix as the biadjacency matrix of a bipartite graph.

Returns Connected component of each node. For bipartite graphs, rows and columns are concatenated (rows first).

Return type labels

sknetwork.topology.**is_connected**(*input_matrix: scipy.sparse.csr.csr_matrix*, *connection: str = 'weak'*, *force_bipartite: bool = False*) → bool

Check whether the graph is connected.

Parameters

- **input_matrix** – Input matrix (either the adjacency matrix or the biadjacency matrix of the graph).
- **connection** – Must be `'weak'` (default) or `'strong'`. The type of connection to use for directed graphs.
- **force_bipartite** (*bool*) – If `True`, consider the input matrix as the biadjacency matrix of a bipartite graph.

sknetwork.topology.**get_largest_connected_component**(*input_matrix: scipy.sparse.csr.csr_matrix*, *connection: str = 'weak'*, *force_bipartite: bool = False*, *return_index: bool = False*) → Union[scipy.sparse.csr.csr_matrix, Tuple[scipy.sparse.csr.csr_matrix, numpy.ndarray]]

Extract the largest connected component of a graph. Bipartite graphs are treated as undirected.

Parameters

- **input_matrix** – Adjacency matrix or biadjacency matrix of the graph.
- **connection** – Must be `'weak'` (default) or `'strong'`. The type of connection to use for directed graphs.
- **force_bipartite** (*bool*) – If `True`, consider the input matrix as the biadjacency matrix of a bipartite graph.
- **return_index** (*bool*) – Whether to return the index of the nodes of the largest connected component in the original graph.

Returns

- **output_matrix** (*sparse.csr_matrix*) – Adjacency matrix or biadjacency matrix of the largest connected component.
- **index** (*array*) – Indices of the nodes in the original graph. For bipartite graphs, rows and columns are concatenated (rows first).

3.6.2 Structure

sknetwork.topology.**is_bipartite**(*adjacency: scipy.sparse.csr.csr_matrix*, *return_biadjacency: bool = False*) → Union[bool, Tuple[bool, Optional[scipy.sparse.csr.csr_matrix], Optional[numpy.ndarray], Optional[numpy.ndarray]]]

Check whether a graph is bipartite.

Parameters

- **adjacency** – Adjacency matrix of the graph (symmetric).
- **return_biadjacency** – If `True`, return a biadjacency matrix of the graph if bipartite.

Returns

- **is_bipartite** (*bool*) – A boolean denoting if the graph is bipartite.

- **biadjacency** (*sparse.csr_matrix*) – A biadjacency matrix of the graph if bipartite (optional).
- **rows** (*np.ndarray*) – Index of rows in the original graph (optional).
- **cols** (*np.ndarray*) – Index of columns in the original graph (optional).

sknetwork.topology.**is_acyclic**(*adjacency: scipy.sparse.csr.csr_matrix*) → bool
Check whether a graph has no cycle.

Parameters **adjacency** – Adjacency matrix of the graph.

Returns **is_acyclic** – A boolean with value True if the graph has no cycle and False otherwise

Return type bool

class sknetwork.topology.**DAG**(*ordering: Optional[str] = None*)
Build a Directed Acyclic Graph from an adjacency.

- Graphs
- DiGraphs

Parameters **ordering** (*str*) – A method to sort the nodes.

- If None, the default order is the index.
- If `'degree'`, the nodes are sorted by ascending degree.

Variables

- **indptr_** (*np.ndarray*) – Pointer index as for CSR format.
- **indices_** (*np.ndarray*) – Indices as for CSR format.

fit(*adjacency: scipy.sparse.csr.csr_matrix*, *sorted_nodes=None*)
Fit algorithm to the data.

Parameters

- **adjacency** – Adjacency matrix of the graph.
- **sorted_nodes** (*np.ndarray*) – An order on the nodes such that the DAG only contains edges (i, j) such that `sorted_nodes[i] < sorted_nodes[j]`.

3.6.3 Core decomposition

class sknetwork.topology.**CoreDecomposition**
K-core decomposition algorithm.

- Graphs

Variables

- **labels_** (*np.ndarray*) – Core value of each node.
- **core_value_** (*int*) – Maximum core value of the graph

Example

```
>>> from sknetwork.topology import CoreDecomposition
>>> from sknetwork.data import karate_club
>>> kcore = CoreDecomposition()
>>> adjacency = karate_club()
>>> kcore.fit(adjacency)
>>> kcore.core_value_
4
```

fit(*adjacency: scipy.sparse.csr.csr_matrix*) → *sknetwork.topology.kcore.CoreDecomposition*
K-core decomposition.

Parameters **adjacency** – Adjacency matrix of the graph.

Returns **self**

Return type *CoreDecomposition*

fit_transform(*adjacency: scipy.sparse.csr.csr_matrix*)
Fit algorithm to the data and return the core value of each node. Same parameters as the `fit` method.

Returns Core value of the nodes.

Return type labels

3.6.4 Cliques

class sknetwork.topology.**Triangles**(*parallelize: bool = False*)
Count the number of triangles in a graph, and evaluate the clustering coefficient.

- Graphs

Parameters **parallelize** – If `True`, use a parallel range while listing the triangles.

Variables

- **n_triangles_** (*int*) – Number of triangles.
- **clustering_coef_** (*float*) – Global clustering coefficient of the graph.

Example

```
>>> from sknetwork.data import karate_club
>>> triangles = Triangles()
>>> adjacency = karate_club()
>>> triangles.fit_transform(adjacency)
45
```

fit(*adjacency: scipy.sparse.csr.csr_matrix*) → *sknetwork.topology.triangles.Triangles*
Count triangles.

Parameters **adjacency** – Adjacency matrix of the graph.

Returns **self**

Return type *Triangles*

fit_transform(*adjacency: scipy.sparse.csr.csr_matrix*) → int
Fit algorithm to the data and return the number of triangles. Same parameters as the `fit` method.

Returns **n_triangles_** – Number of triangles.

Return type int

class sknetwork.topology.**Cliques**(*k: int*)
Clique counting algorithm.

Parameters **k** (*int*) – Clique order (e.g., k = 3 means triangles).

Variables **n_cliques_** (*int*) – Number of cliques.

Example

```
>>> from sknetwork.data import karate_club
>>> cliques = Cliques(k=3)
>>> adjacency = karate_club()
>>> cliques.fit_transform(adjacency)
45
```

References

Danisch, M., Balalau, O., & Sozio, M. (2018, April). Listing k-cliques in sparse real-world graphs. In Proceedings of the 2018 World Wide Web Conference (pp. 589-598).

fit(*adjacency: scipy.sparse.csr.csr_matrix*) → *sknetwork.topology.kcliques.Cliques*
K-cliques count.

Parameters **adjacency** – Adjacency matrix of the graph.

Returns **self**

Return type *Cliques*

fit_transform(*adjacency: scipy.sparse.csr.csr_matrix*) → int
Fit algorithm to the data and return the number of cliques. Same parameters as the `fit` method.

Returns **n_cliques** – Number of k-cliques.

Return type int

3.6.5 Isomorphism

class sknetwork.topology.**WeisfeilerLehman**(*max_iter: int = - 1*)
Weisfeiler-Lehman algorithm for coloring/labeling graphs in order to check similarity.

Parameters **max_iter** (*int*) – Maximum number of iterations. Negative value means until convergence.

Variables **labels_** (*np.ndarray*) – Label of each node.

Example

```
>>> from sknetwork.topology import WeisfeilerLehman
>>> from sknetwork.data import house
>>> weisfeiler_lehman = WeisfeilerLehman()
>>> adjacency = house()
>>> labels = weisfeiler_lehman.fit_transform(adjacency)
>>> labels
array([0, 2, 1, 1, 2], dtype=int32)
```

References

- Douglas, B. L. (2011). The Weisfeiler-Lehman Method and Graph Isomorphism Testing.
- Shervashidze, N., Schweitzer, P., van Leeuwen, E. J., Melhorn, K., Borgwardt, K. M. (2011) Weisfeiler-Lehman graph kernels. Journal of Machine Learning Research 12, 2011.

fit(*adjacency: Union[scipy.sparse.csr.csr_matrix, numpy.ndarray]*) → *sknetwork.topology.weisfeiler_lehman.WeisfeilerLehman*

Fit algorithm to the data.

Parameters **adjacency** (*Union[sparse.csr_matrix, np.ndarray]*) – Adjacency matrix of the graph.

Returns **self**

Return type *WeisfeilerLehman*

fit_transform(*adjacency: Union[scipy.sparse.csr.csr_matrix, numpy.ndarray]*) → numpy.ndarray

Fit algorithm to the data and return the labels. Same parameters as the `fit` method.

Returns **labels** – Labels.

Return type np.ndarray

sknetwork.topology.**are_isomorphic**(*adjacency1: scipy.sparse.csr.csr_matrix*, *adjacency2: scipy.sparse.csr.csr_matrix*, *max_iter: int = - 1*) → bool

Weisfeiler-Lehman isomorphism test. If the test is False, the graphs cannot be isomorphic, otherwise, they might be.

Parameters

- **adjacency1** – First adjacency matrix.
- **adjacency2** – Second adjacency matrix.
- **max_iter** (*int*) – Maximum number of coloring iterations. Negative value means until convergence.

Returns **test_result**

Return type bool

Example

```
>>> from sknetwork.topology import are_isomorphic
>>> from sknetwork.data import house, bow_tie
>>> are_isomorphic(house(), bow_tie())
False
```

References

- Douglas, B. L. (2011). The Weisfeiler-Lehman Method and Graph Isomorphism Testing.
- Shervashidze, N., Schweitzer, P., van Leeuwen, E. J., Melhorn, K., Borgwardt, K. M. (2011) Weisfeiler-Lehman graph kernels. Journal of Machine Learning Research 12, 2011.

3.7 Path

Standard algorithms related to graph traversal.

Most algorithms are adapted from SciPy.

3.7.1 Shortest path

sknetwork.path.**get_distances**(*adjacency: scipy.sparse.csr.csr_matrix*, *sources: Optional[Union[int, Iterable]] = None*, *method: str = 'D'*, *return_predecessors: bool = False*, *unweighted: bool = False*, *n_jobs: Optional[int] = None*)

Compute distances between nodes.

- Graphs
- Digraphs

Based on SciPy (scipy.sparse.csgraph.shortest_path)

Parameters

- **adjacency** – The adjacency matrix of the graph
- **sources** – If specified, only compute the paths for the points at the given indices. Will not work with `method ='FW'`.
- **method** – The method to be used.
  - `'D'` (Dijkstra),
  - `'BF'` (Bellman-Ford),
  - `'J'` (Johnson).
- **return_predecessors** – If `True`, the size predecessor matrix is returned
- **unweighted** – If `True`, the weights of the edges are ignored
- **n_jobs** – If an integer value is given, denotes the number of workers to use (-1 means the maximum number will be used). If `None`, no parallel computations are made.

Returns

- **dist_matrix** (*np.ndarray*) – Matrix of distances between nodes. `dist_matrix[i,j]` gives the shortest distance from the `i`-th source to node `j` in the graph (infinite if no path exists from the `i`-th source to node `j`).
- **predecessors** (*np.ndarray, optional*) – Returned only if `return_predecessors == True`. The matrix of predecessors, which can be used to reconstruct the shortest paths. Row `i` of the predecessor matrix contains information on the shortest paths from the `i`-th source: each entry `predecessors[i, j]` gives the index of the previous node in the path from the `i`-th source to node `j` (-1 if no path exists from the `i`-th source to node `j`).

Examples

```
>>> from sknetwork.data import cyclic_digraph
>>> adjacency = cyclic_digraph(3)
>>> get_distances(adjacency, sources=0)
array([0., 1., 2.])
>>> get_distances(adjacency, sources=0, return_predecessors=True)
(array([0., 1., 2.]), array([-1,  0,  1]))
```

sknetwork.path.**get_shortest_path**(*adjacency: scipy.sparse.csr.csr_matrix*, *sources: Union[int, Iterable]*, *targets: Union[int, Iterable]*, *method: str = 'D'*, *unweighted: bool = False*, *n_jobs: Optional[int] = None*)

Compute the shortest paths in the graph.

Parameters

- **adjacency** – The adjacency matrix of the graph
- **sources** (*int or iterable*) – Sources nodes.
- **targets** (*int or iterable*) – Target nodes.
- **method** – The method to be used.
  - `'D'` (Dijkstra),
  - `'BF'` (Bellman-Ford),
  - `'J'` (Johnson).
- **unweighted** – If `True`, the weights of the edges are ignored
- **n_jobs** – If an integer value is given, denotes the number of workers to use (-1 means the maximum number will be used). If `None`, no parallel computations are made.

Returns **paths** – If single source and single target, return a list containing the nodes on the path from source to target. If multiple sources or multiple targets, return a list of paths as lists. An empty list means that the path does not exist.

Return type list

Examples

```
>>> from sknetwork.data import linear_digraph
>>> adjacency = linear_digraph(3)
>>> get_shortest_path(adjacency, 0, 2)
[0, 1, 2]
>>> get_shortest_path(adjacency, 2, 0)
[]
>>> get_shortest_path(adjacency, 0, [1, 2])
[[0, 1], [0, 1, 2]]
>>> get_shortest_path(adjacency, [0, 1], 2)
[[0, 1, 2], [1, 2]]
```

Summary of the different methods and their worst-case complexity for n nodes and m edges (for the all-pairs problem):

Method	Worst-case time complexity	Remarks
Dijkstra	$O(n^2 \log n + nm)$	For use on graphs with positive weights only
Bellman-Ford	$O(nm)$	For use on graphs without negative-weight cycles only
Johnson	$O(n^2 \log n + nm)$	

3.7.2 Search

sknetwork.path.**breadth_first_search**(*adjacency: scipy.sparse.csr.csr_matrix*, *source: int*, *return_predecessors: bool = True*)

Breadth-first ordering starting with specified node.

- Graphs
- Digraphs

Based on SciPy (scipy.sparse.csgraph.breadth_first_order)

Parameters

- **adjacency** – The adjacency matrix of the graph
- **source** (*int*) – The node from which to start the ordering
- **return_predecessors** (*bool*) – If `True`, the size predecessor matrix is returned

Returns

- **node_array** (*np.ndarray*) – The breadth-first list of nodes, starting with specified node. The length of node_array is the number of nodes reachable from the specified node.
- **predecessors** (*np.ndarray*) – Returned only if `return_predecessors == True`. The list of predecessors of each node in a breadth-first tree. If node `i` is in the tree, then its parent is given by `predecessors[i]`. If node `i` is not in the tree (and for the parent node) then `predecessors[i] = -9999`.

sknetwork.path.**depth_first_search**(*adjacency: scipy.sparse.csr.csr_matrix*, *source: int*, *return_predecessors: bool = True*)

Depth-first ordering starting with specified node.

- Graphs
- Digraphs

Based on SciPy (scipy.sparse.csgraph.depth_first_order)

Parameters

- **adjacency** – The adjacency matrix of the graph
- **source** – The node from which to start the ordering
- **return_predecessors** – If `True`, the size predecessor matrix is returned

Returns

- **node_array** (*np.ndarray*) – The depth-first list of nodes, starting with specified node. The length of node_array is the number of nodes reachable from the specified node.
- **predecessors** (*np.ndarray*) – Returned only if `return_predecessors == True`. The list of predecessors of each node in a depth-first tree. If node `i` is in the tree, then its parent is given by `predecessors[i]`. If node `i` is not in the tree (and for the parent node) then `predecessors[i] = -9999`.

3.7.3 Metrics

`sknetwork.path.`**get_diameter**(*adjacency: Union[scipy.sparse.csr.csr_matrix, numpy.ndarray]*, *n_sources: Optional[Union[int, float]] = None*, *unweighted: bool = False*, *n_jobs: Optional[int] = None*) → int

Lower bound on the diameter of a graph which is the length of the longest shortest path between two nodes.

Parameters

- **adjacency** – Adjacency matrix of the graph.
- **n_sources** – Number of node sources to use for approximation.
  - If None, compute exact diameter.
  - If int, sample n_sample source nodes at random.
  - If float, sample (n_samples * n) source nodes at random.
- **unweighted** – Whether or not the graph is unweighted.
- **n_jobs** – If an integer value is given, denotes the number of workers to use (-1 means the maximum number will be used). If `None`, no parallel computations are made.

Returns **diameter**

Return type int

Examples

```
>>> from sknetwork.data import house
>>> adjacency = house()
>>> d_exact = get_diameter(adjacency)
>>> d_exact
2
>>> d_approx = get_diameter(adjacency, 2)
>>> d_approx <= d_exact
True
>>> d_approx = get_diameter(adjacency, 0.5)
>>> d_approx <= d_exact
True
```

Notes

This is a basic implementation that computes distances between nodes and returns the maximum.

sknetwork.path.**get_radius**(*adjacency: Union[scipy.sparse.csr.csr_matrix, numpy.ndarray]*, *n_sources: Optional[Union[int, float]] = None*, *unweighted: bool = False*, *n_jobs: Optional[int] = None*) → int

Computes the radius of the graph which. The radius of the graph is the minimum eccentricity of the graph.

Parameters

- **adjacency** – Adjacency matrix of the graph.
- **n_sources** – Number of node sources to use for approximation.
  - If None, compute exact diameter.
  - If int, sample n_sample source nodes at random.
  - If float, sample (n_samples * n) source nodes at random.
- **unweighted** – Whether the graph is unweighted.
- **n_jobs** – If an integer value is given, denotes the number of workers to use (-1 means the maximum number will be used). If `None`, no parallel computations are made.

Returns **radius**

Return type int

Notes

This is a basic implementation that computes distances between nodes and returns the maximum.

sknetwork.path.**get_eccentricity**(*adjacency: Union[scipy.sparse.csr.csr_matrix, numpy.ndarray]*, *node: int*, *unweighted: bool = False*, *n_jobs: Optional[int] = None*) → int

Computes the eccentricity of a node. The eccentricity of a node, u, is the maximum length of the shortest paths from u to the other nodes in the graph.

Parameters

- **adjacency** – Adjacency matrix of the graph.
- **node** – The node to compute the eccentricity for.
- **unweighted** – Whether or not the graph is unweighted.
- **n_jobs** – If an integer value is given, denotes the number of workers to use (-1 means the maximum number will be used). If `None`, no parallel computations are made.

Returns **eccentricity**

Return type int

3.8 Clustering

Clustering algorithms.

The attribute `labels_` assigns a label (cluster index) to each node of the graph.

3.8.1 Louvain

Here are the available notions of modularity for the Louvain algorithm:

Modularity	Formula
Newman (`'newman'`)	$Q = \frac{1}{w}\sum_{i,j}\left(A_{ij} - \gamma\frac{d_i d_j}{w}\right)\delta_{c_i,c_j}$
Dugué (`'dugue'`)	$Q = \frac{1}{w}\sum_{i,j}\left(A_{ij} - \gamma\frac{d_i^+ d_j^-}{w}\right)\delta_{c_i,c_j}$
Potts (`'potts'`)	$Q = \sum_{i,j}\left(\frac{A_{ij}}{w} - \gamma\frac{1}{n^2}\right)\delta_{c_i,c_j}$

where

- A is the adjacency matrix,
- c_i is the cluster of node i,
- d_i is the degree of node i,
- d_i^+, d_i^- are the out-degree, in-degree of node i (for directed graphs),
- $w = 1^T A 1$ is the sum of degrees,
- δ is the Kronecker symbol,
- $\gamma \geq 0$ is the resolution parameter.

For bipartite graphs, the considered adjacency matrix is

$$A = \begin{pmatrix} 0 & B \\ B^T & 0 \end{pmatrix}$$

for Newman modularity and Potts modularity (i.e., the graph is considered as undirected), and

$$A = \begin{pmatrix} 0 & B \\ 0 & 0 \end{pmatrix}$$

for Dugué modularity (i.e., the graph is considered as directed). The latter is the default option and corresponds to Barber's modularity:

$$Q = \frac{1}{w}\sum_{i,j}\left(B_{ij} - \gamma\frac{d_i f_j}{w}\right)\delta_{c_i,c_j}$$

where i in the row index, j in the column index, d_i is the degree of row i, f_j is the degree of column j and $w = 1^T B 1$ is the sum of degrees (either rows or columns).

When the graph is weighted, the degree of a node is replaced by its weight (sum of edge weights).

class sknetwork.clustering.**Louvain**(*resolution: float = 1*, *modularity: str = 'dugue'*, *tol_optimization: float = 0.001*, *tol_aggregation: float = 0.001*, *n_aggregations: int = - 1*, *shuffle_nodes: bool = False*, *sort_clusters: bool = True*, *return_membership: bool = True*, *return_aggregate: bool = True*, *random_state: Optional[Union[numpy.random.mtrand.RandomState, int]] = None*, *verbose: bool = False*)

Louvain algorithm for clustering graphs by maximization of modularity.

For bipartite graphs, the algorithm maximizes Barber's modularity by default.

Parameters

- **resolution** – Resolution parameter.
- **modularity** (*str*) – Which objective function to maximize. Can be `'dugue'`, `'newman'` or `'potts'` (default = `'dugue'`).
- **tol_optimization** – Minimum increase in the objective function to enter a new optimization pass.
- **tol_aggregation** – Minimum increase in the objective function to enter a new aggregation pass.
- **n_aggregations** – Maximum number of aggregations. A negative value is interpreted as no limit.
- **shuffle_nodes** – Enables node shuffling before optimization.
- **sort_clusters** – If `True`, sort labels in decreasing order of cluster size.
- **return_membership** – If `True`, return the membership matrix of nodes to each cluster (soft clustering).
- **return_aggregate** – If `True`, return the adjacency matrix of the graph between clusters.
- **random_state** – Random number generator or random seed. If None, numpy.random is used.
- **verbose** – Verbose mode.

Variables

- **labels_** (*np.ndarray*) – Labels of the nodes.
- **labels_row_** (*np.ndarray*) – Labels of the rows (for bipartite graphs).
- **labels_col_** (*np.ndarray*) – Labels of the columns (for bipartite graphs).
- **membership_** (*sparse.csr_matrix*) – Membership matrix of the nodes, shape (n_nodes, n_clusters).
- **membership_row_** (*sparse.csr_matrix*) – Membership matrix of the rows (for bipartite graphs).
- **membership_col_** (*sparse.csr_matrix*) – Membership matrix of the columns (for bipartite graphs).
- **aggregate_** (*sparse.csr_matrix*) – Aggregate adjacency matrix or biadjacency matrix between clusters.

Example

```
>>> from sknetwork.clustering import Louvain
>>> from sknetwork.data import karate_club
>>> louvain = Louvain()
>>> adjacency = karate_club()
>>> labels = louvain.fit_transform(adjacency)
>>> len(set(labels))
4
```

References

- Blondel, V. D., Guillaume, J. L., Lambiotte, R., & Lefebvre, E. (2008). Fast unfolding of communities in large networks. Journal of statistical mechanics: theory and experiment, 2008.
- Dugué, N., & Perez, A. (2015). Directed Louvain: maximizing modularity in directed networks (Doctoral dissertation, Université d'Orléans).
- Barber, M. J. (2007). Modularity and community detection in bipartite networks Physical Review E, 76(6).

fit(*input_matrix: Union[scipy.sparse.csr.csr_matrix, numpy.ndarray]*, *force_bipartite: bool = False*) → *sknetwork.clustering.louvain.Louvain*
Fit algorithm to data.

Parameters

- **input_matrix** – Adjacency matrix or biadjacency matrix of the graph.
- **force_bipartite** – If `True`, force the input matrix to be considered as a biadjacency matrix even if square.

Returns **self**

Return type *Louvain*

fit_transform(**args*, ***kwargs*) → numpy.ndarray
Fit algorithm to the data and return the labels. Same parameters as the `fit` method.

Returns **labels** – Labels.

Return type np.ndarray

3.8.2 Propagation

class sknetwork.clustering.**PropagationClustering**(*n_iter: int = 5*, *node_order: str = 'decreasing'*, *weighted: bool = True*, *sort_clusters: bool = True*, *return_membership: bool = True*, *return_aggregate: bool = True*)

Clustering by label propagation.

Parameters

- **n_iter** (*int*) – Maximum number of iterations (-1 for infinity).
- **node_order** (*str*) –
  - *'random'*: node labels are updated in random order.
  - *'increasing'*: node labels are updated by increasing order of weight.
  - *'decreasing'*: node labels are updated by decreasing order of weight.
  - Otherwise, node labels are updated by index order.
- **weighted** (*bool*) – If `True`, the vote of each neighbor is proportional to the edge weight. Otherwise, all votes have weight 1.
- **sort_clusters** – If `True`, sort labels in decreasing order of cluster size.
- **return_membership** – If `True`, return the membership matrix of nodes to each cluster (soft clustering).

- **return_aggregate** – If True, return the aggregate adjacency matrix or biadjacency matrix between clusters.

Variables

- **labels_** (*np.ndarray*) – Labels of the nodes.
- **labels_row_** (*np.ndarray*) – Labels of the rows (for bipartite graphs).
- **labels_col_** (*np.ndarray*) – Labels of the columns (for bipartite graphs).
- **membership_** (*sparse.csr_matrix*) – Membership matrix of the nodes, shape (n_nodes, n_clusters).
- **membership_row_** (*sparse.csr_matrix*) – Membership matrix of the rows (for bipartite graphs).
- **membership_col_** (*sparse.csr_matrix*) – Membership matrix of the columns (for bipartite graphs).
- **aggregate_** (*sparse.csr_matrix*) – Aggregate adjacency matrix or biadjacency matrix between clusters.

Example

```
>>> from sknetwork.clustering import PropagationClustering
>>> from sknetwork.data import karate_club
>>> propagation = PropagationClustering()
>>> graph = karate_club(metadata=True)
>>> adjacency = graph.adjacency
>>> labels = propagation.fit_transform(adjacency)
>>> len(set(labels))
2
```

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.

fit(*input_matrix: Union[scipy.sparse.csr.csr_matrix, numpy.ndarray]*) → *sknetwork.clustering.propagation_clustering.PropagationClustering*

Clustering by label propagation.

Parameters **input_matrix** – Adjacency matrix or biadjacency matrix of the graph.

Returns **self**

Return type *PropagationClustering*

fit_transform(**args*, ***kwargs*) → numpy.ndarray

Fit algorithm to the data and return the labels. Same parameters as the fit method.

Returns **labels** – Labels.

Return type np.ndarray

score(*label: int*)

Classification scores for a given label.

Parameters **label** (*int*) – The label index of the class.

Returns **scores** – Classification scores of shape (number of nodes,).

Return type np.ndarray

3.8.3 K-Means

class sknetwork.clustering.**KMeans**(*n_clusters: int = 2*, *embedding_method: sknetwork.embedding.base.BaseEmbedding = Spectral(n_components=10, decomposition='rw', regularization=- 1, normalized=True)*, *co_cluster: bool = False*, *sort_clusters: bool = True*, *return_membership: bool = True*, *return_aggregate: bool = True*)

K-means clustering applied in the embedding space.

Parameters

- **n_clusters** – Number of desired clusters (default = 2).
- **embedding_method** – Embedding method (default = Spectral embedding in dimension 10).
- **co_cluster** – If `True`, co-cluster rows and columns, considered as different nodes (default = `False`).
- **sort_clusters** – If `True`, sort labels in decreasing order of cluster size.
- **return_membership** – If `True`, return the membership matrix of nodes to each cluster (soft clustering).
- **return_aggregate** – If `True`, return the adjacency matrix of the graph between clusters.

Variables

- **labels_** (*np.ndarray*) – Labels of the nodes.
- **labels_row_** (*np.ndarray*) – Labels of the rows (for bipartite graphs).
- **labels_col_** (*np.ndarray*) – Labels of the columns (for bipartite graphs).
- **membership_** (*sparse.csr_matrix*) – Membership matrix of the nodes, shape (n_nodes, n_clusters).
- **membership_row_** (*sparse.csr_matrix*) – Membership matrix of the rows (for bipartite graphs).
- **membership_col_** (*sparse.csr_matrix*) – Membership matrix of the columns (for bipartite graphs).
- **aggregate_** (*sparse.csr_matrix*) – Aggregate adjacency matrix or biadjacency matrix between clusters.

Example

```
>>> from sknetwork.clustering import KMeans
>>> from sknetwork.data import karate_club
>>> kmeans = KMeans(n_clusters=3)
>>> adjacency = karate_club()
>>> labels = kmeans.fit_transform(adjacency)
>>> len(set(labels))
3
```

fit(*input_matrix: Union[scipy.sparse.csr.csr_matrix, numpy.ndarray]*) → *sknetwork.clustering.kmeans.KMeans*

Apply embedding method followed by K-means.

Parameters **input_matrix** – Adjacency matrix or biadjacency matrix of the graph.

Returns **self**

Return type *KMeans*

fit_transform(**args*, ***kwargs*) → numpy.ndarray

Fit algorithm to the data and return the labels. Same parameters as the `fit` method.

Returns **labels** – Labels.

Return type np.ndarray

3.8.4 Metrics

sknetwork.clustering.**modularity**(*adjacency: Union[scipy.sparse.csr.csr_matrix, numpy.ndarray]*, *labels: numpy.ndarray*, *weights: Union[str, numpy.ndarray] = 'degree'*, *weights_in: Union[str, numpy.ndarray] = 'degree'*, *resolution: float = 1*, *return_all: bool = False*) → Union[float, Tuple[float, float, float]]

Modularity of a clustering.

The modularity of a clustering is

$Q = \frac{1}{w}\sum_{i,j}\left(A_{ij} - \gamma\frac{d_i d_j}{w}\right)\delta_{c_i,c_j}$ for graphs,

$Q = \frac{1}{w}\sum_{i,j}\left(A_{ij} - \gamma\frac{d_i^+ d_j^-}{w}\right)\delta_{c_i,c_j}$ for directed graphs,

where

- c_i is the cluster of node i,
- d_i is the weight of node i,
- d_i^+, d_i^- are the out-weight, in-weight of node i (for digraphs),
- $w = 1^T A 1$ is the total weight,
- δ is the Kronecker symbol,
- $\gamma \geq 0$ is the resolution parameter.

Parameters

- **adjacency** – Adjacency matrix of the graph.
- **labels** – Labels of nodes, vector of size n .
- **weights** – Weights of nodes. `'degree'` (default), `'uniform'` or custom weights.
- **weights_in** – In-weights of nodes. `None` (default), `'degree'`, `'uniform'` or custom weights. If `None`, taken equal to weights.
- **resolution** – Resolution parameter (default = 1).
- **return_all** – If `True`, return modularity, fit, diversity.

Returns

- **modularity** (*float*)

- **fit** (*float, optional*)
- **diversity** (*float, optional*)

Example

```
>>> from sknetwork.clustering import modularity
>>> from sknetwork.data import house
>>> adjacency = house()
>>> labels = np.array([0, 0, 1, 1, 0])
>>> np.round(modularity(adjacency, labels), 2)
0.11
```

sknetwork.clustering.**bimodularity**(*biadjacency: Union[scipy.sparse.csr.csr_matrix, numpy.ndarray]*, *labels: numpy.ndarray*, *labels_col: numpy.ndarray*, *weights: Union[str, numpy.ndarray] = 'degree'*, *weights_col: Union[str, numpy.ndarray] = 'degree'*, *resolution: float = 1*, *return_all: bool = False*) → Union[float, Tuple[float, float, float]]

Bimodularity of the clustering (for bipartite graphs).

The bimodularity of a clustering is

$$Q = \sum_i \sum_j \left(\frac{B_{ij}}{w} - \gamma \frac{d_{1,i} d_{2,j}}{w^2} \right) \delta_{c_{1,i}, c_{2,j}}$$

where

- $c_{1,i}, c_{2,j}$ are the clusters of nodes i (row) and j (column),
- $d_{1,i}, d_{2,j}$ are the weights of nodes i (row) and j (column),
- $w = 1^T B 1$ is the total weight,
- δ is the Kronecker symbol,
- $\gamma \geq 0$ is the resolution parameter.

Parameters

- **biadjacency** – Biadjacency matrix of the graph (shape $n_1 \times n_2$).
- **labels** – Labels of rows, vector of size n_1.
- **labels_col** – Labels of columns, vector of size n_2.
- **weights** – Weights of nodes. `'degree'` (default), `'uniform'` or custom weights.
- **weights_col** – Weights of columns. `'degree'` (default), `'uniform'` or custom weights.
- **resolution** – Resolution parameter (default = 1).
- **return_all** – If `True`, return modularity, fit, diversity.

Returns

- **modularity** (*float*)
- **fit** (*float, optional*)
- **diversity** (*float, optional*)

Example

```
>>> from sknetwork.clustering import bimodularity
>>> from sknetwork.data import star_wars
>>> biadjacency = star_wars()
>>> labels = np.array([1, 1, 0, 0])
>>> labels_col = np.array([1, 0, 0])
>>> np.round(bimodularity(biadjacency, labels, labels_col), 2)
0.22
```

sknetwork.clustering.**comodularity**(*adjacency: Union[scipy.sparse.csr.csr_matrix, numpy.ndarray]*, *labels: numpy.ndarray*, *resolution: float = 1*, *return_all: bool = False*) → Union[float, Tuple[float, float, float]]

Modularity of a clustering in the normalized co-neighborhood graph.

Quality metric of a clustering given by:

$$Q = \frac{1}{w} \sum_{i,j} \left((AD_2^{-1}A^T)_{ij} - \gamma \frac{d_i d_j}{w} \right) \delta_{c_i,c_j}$$

where

- c_i is the cluster of node i,
- D_2 is the diagonal matrix of the weights of columns,
- δ is the Kronecker symbol,
- $\gamma \geq 0$ is the resolution parameter.

Parameters

- **adjacency** – Adjacency matrix or biadjacency matrix of the graph.
- **labels** – Labels of the nodes.
- **resolution** – Resolution parameter (default = 1).
- **return_all** – If True, return modularity, fit, diversity.

Returns

- **modularity** (*float*)
- **fit** (*float, optional*)
- **diversity** (*float, optional*)

Example

```
>>> from sknetwork.clustering import comodularity
>>> from sknetwork.data import house
>>> adjacency = house()
>>> labels = np.array([0, 0, 1, 1, 0])
>>> np.round(comodularity(adjacency, labels), 2)
0.06
```

Notes

Does not require the computation of the adjacency matrix of the normalized co-neighborhood graph.

sknetwork.clustering.**normalized_std**(*labels: numpy.ndarray*) → float

Normalized standard deviation of cluster sizes.

A score of 1 means perfectly balanced clustering.

Parameters **labels** – Labels of nodes.

Returns

Return type float

Example

```
>>> from sknetwork.clustering import normalized_std
>>> labels = np.array([0, 0, 1, 1])
>>> normalized_std(labels)
1.0
```

3.8.5 Post-processing

sknetwork.clustering.postprocess.**reindex_labels**(*labels: numpy.ndarray*, *consecutive: bool = True*) → numpy.ndarray

Reindex clusters in decreasing order of size.

Parameters

- **labels** – label of each node.
- **consecutive** – If `True`, the set of labels must be from 0 to $k-1$, where k is the number of labels. Lead to faster computation.

Returns **new_labels** – New label of each node.

Return type np.ndarray

Example

```
>>> from sknetwork.clustering import reindex_labels
>>> labels = np.array([0, 1, 1])
>>> reindex_labels(labels)
array([1, 0, 0])
```

3.9 Hierarchy

Hierarchical clustering algorithms.

The attribute `dendrogram_` gives the dendrogram.

A dendrogram is an array of size $(n-1) \times 4$ representing the successive merges of nodes. Each row gives the two merged nodes, their distance and the size of the resulting cluster. Any new node resulting from a merge takes the first available index (e.g., the first merge corresponds to node n).

3.9.1 Paris

class sknetwork.hierarchy.**Paris**(*weights: unicode = 'degree', reorder: bool = True*)

Agglomerative clustering algorithm that performs greedy merge of nodes based on their similarity.

The similarity between nodes i, j is $\dfrac{A_{ij}}{w_i w_j}$ where

- A_{ij} is the weight of edge i, j,
- w_i, w_j are the weights of nodes i, j

If the input matrix B is a biadjacency matrix (i.e., rectangular), the algorithm is applied to the corresponding adjacency matrix $A = \begin{bmatrix} 0 & B \\ B^T & 0 \end{bmatrix}$

Parameters

- **weights** – Weights of nodes. `'degree'` (default) or `'uniform'`.
- **reorder** – If `True` (default), reorder the dendrogram in non-decreasing order of height.

Variables

- **dendrogram_** – Dendrogram of the graph.
- **dendrogram_row_** – Dendrogram for the rows, for bipartite graphs.
- **dendrogram_col_** – Dendrogram for the columns, for bipartite graphs.
- **dendrogram_full_** – Dendrogram for both rows and columns, indexed in this order, for bipartite graphs.

Examples

```
>>> from sknetwork.hierarchy import Paris
>>> from sknetwork.data import house
>>> paris = Paris()
>>> adjacency = house()
>>> dendrogram = paris.fit_transform(adjacency)
>>> np.round(dendrogram, 2)
array([[3.  , 2.  , 0.17, 2.  ],
       [1.  , 0.  , 0.25, 2.  ],
       [6.  , 4.  , 0.31, 3.  ],
       [7.  , 5.  , 0.67, 5.  ]])
```

Notes

Each row of the dendrogram = i, j, distance, size of cluster $i + j$.

See also:

`scipy.cluster.hierarchy.linkage`

References

T. Bonald, B. Charpentier, A. Galland, A. Hollocou (2018). Hierarchical Graph Clustering using Node Pair Sampling. Workshop on Mining and Learning with Graphs.

fit(*input_matrix: Union[scipy.sparse.csr.csr_matrix, numpy.ndarray]*) → *sknetwork.hierarchy.paris.Paris*
Agglomerative clustering using the nearest neighbor chain.

Parameters **input_matrix** – Adjacency matrix or biadjacency matrix of the graph.

Returns **self**

Return type *Paris*

fit_transform(**args*, ***kwargs*) → numpy.ndarray
Fit algorithm to data and return the dendrogram. Same parameters as the `fit` method.

Returns **dendrogram** – Dendrogram.

Return type np.ndarray

3.9.2 Louvain

class sknetwork.hierarchy.**LouvainHierarchy**(*depth: int = 3*, *resolution: float = 1*, *tol_optimization: float = 0.001*, *tol_aggregation: float = 0.001*, *n_aggregations: int = - 1*, *shuffle_nodes: bool = False*, *random_state: Optional[Union[numpy.random.mtrand.RandomState, int]] = None*, *verbose: bool = False*)
Hierarchical clustering by successive instances of Louvain (top-down).

Parameters

- **depth** – Depth of the tree. A negative value is interpreted as no limit (return a tree of maximum depth).
- **resolution** – Resolution parameter.
- **tol_optimization** – Minimum increase in the objective function to enter a new optimization pass.
- **tol_aggregation** – Minimum increase in the objective function to enter a new aggregation pass.
- **n_aggregations** – Maximum number of aggregations. A negative value is interpreted as no limit.
- **shuffle_nodes** – Enables node shuffling before optimization.
- **random_state** – Random number generator or random seed. If `None`, numpy.random is used.
- **verbose** – Verbose mode.

Variables

- **dendrogram_** – Dendrogram of the graph.
- **dendrogram_row_** – Dendrogram for the rows, for bipartite graphs.
- **dendrogram_col_** – Dendrogram for the columns, for bipartite graphs.
- **dendrogram_full_** – Dendrogram for both rows and columns, indexed in this order, for bipartite graphs.

Example

```
>>> from sknetwork.hierarchy import LouvainHierarchy
>>> from sknetwork.data import house
>>> louvain = LouvainHierarchy()
>>> adjacency = house()
>>> louvain.fit_transform(adjacency)
array([[3., 2., 0., 2.],
       [4., 1., 0., 2.],
       [6., 0., 0., 3.],
       [5., 7., 1., 5.]])
```

Notes

Each row of the dendrogram = merge nodes, distance, size of cluster.

See also:

`scipy.cluster.hierarchy.dendrogram`

fit(*input_matrix: Union[scipy.sparse.csr.csr_matrix, numpy.ndarray]*) → *sknetwork.hierarchy.louvain_hierarchy.LouvainHierarchy*

Fit algorithm to data.

Parameters **input_matrix** – Adjacency matrix or biadjacency matrix of the graph.

Returns **self**

Return type *LouvainHierarchy*

fit_transform(**args*, ***kwargs*) → numpy.ndarray

Fit algorithm to data and return the dendrogram. Same parameters as the `fit` method.

Returns **dendrogram** – Dendrogram.

Return type np.ndarray

3.9.3 Ward

class sknetwork.hierarchy.**Ward**(*embedding_method: sknetwork.embedding.base.BaseEmbedding = Spectral(n_components=10, decomposition='rw', regularization=- 1, normalized=True)*, *co_cluster: bool = False*)

Hierarchical clustering by the Ward method.

Parameters

- **embedding_method** – Embedding method (default = Spectral embedding in dimension 10).

- **co_cluster** – If `True`, co-cluster rows and columns, considered as different nodes (default = `False`).

Variables

- **dendrogram_** – Dendrogram of the graph.
- **dendrogram_row_** – Dendrogram for the rows, for bipartite graphs.
- **dendrogram_col_** – Dendrogram for the columns, for bipartite graphs.
- **dendrogram_full_** – Dendrogram for both rows and columns, indexed in this order, for bipartite graphs.

Examples

```
>>> from sknetwork.hierarchy import Ward
>>> from sknetwork.data import karate_club
>>> ward = Ward()
>>> adjacency = karate_club()
>>> dendrogram = ward.fit_transform(adjacency)
>>> dendrogram.shape
(33, 4)
```

References

- Ward, J. H., Jr. (1963). Hierarchical grouping to optimize an objective function. Journal of the American Statistical Association.
- Murtagh, F., & Contreras, P. (2012). Algorithms for hierarchical clustering: an overview. Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery.

fit(*input_matrix: Union[scipy.sparse.csr.csr_matrix, numpy.ndarray]*) → *sknetwork.hierarchy.ward.Ward*

Applies embedding method followed by the Ward algorithm.

Parameters **input_matrix** – Adjacency matrix or biadjacency matrix of the graph.

Returns **self**

Return type *Ward*

fit_transform(**args*, ***kwargs*) → numpy.ndarray

Fit algorithm to data and return the dendrogram. Same parameters as the `fit` method.

Returns **dendrogram** – Dendrogram.

Return type np.ndarray

3.9.4 Metrics

sknetwork.hierarchy.**dasgupta_cost**(*adjacency: scipy.sparse.csr.csr_matrix*, *dendrogram: numpy.ndarray*, *weights: str = 'uniform'*, *normalized: bool = False*) → float

Dasgupta's cost of a hierarchy.

Expected size (weights = `'uniform'`) or expected volume (weights = `'degree'`) of the cluster induced by random edge sampling (closest ancestor of the two nodes in the hierarchy).

Parameters

- **adjacency** – Adjacency matrix of the graph.
- **dendrogram** – Dendrogram.
- **weights** – Weights of nodes. `'degree'` or `'uniform'` (default).
- **normalized** – If `True`, normalized cost (between 0 and 1).

Returns **cost** – Cost.

Return type float

Example

```
>>> from sknetwork.hierarchy import dasgupta_score, Paris
>>> from sknetwork.data import house
>>> paris = Paris()
>>> adjacency = house()
>>> dendrogram = paris.fit_transform(adjacency)
>>> cost = dasgupta_cost(adjacency, dendrogram)
>>> np.round(cost, 2)
3.33
```

References

Dasgupta, S. (2016). A cost function for similarity-based hierarchical clustering. Proceedings of ACM symposium on Theory of Computing.

sknetwork.hierarchy.**dasgupta_score**(*adjacency: scipy.sparse.csr.csr_matrix*, *dendrogram: numpy.ndarray*, *weights: str = 'uniform'*) → float

Dasgupta's score of a hierarchy (quality metric, between 0 and 1).

Defined as 1 - normalized Dasgupta's cost.

Parameters

- **adjacency** – Adjacency matrix of the graph.
- **dendrogram** – Dendrogram.
- **weights** – Weights of nodes. `'degree'` or `'uniform'` (default).

Returns **score** – Score.

Return type float

Example

```
>>> from sknetwork.hierarchy import dasgupta_score, Paris
>>> from sknetwork.data import house
>>> paris = Paris()
>>> adjacency = house()
>>> dendrogram = paris.fit_transform(adjacency)
>>> score = dasgupta_score(adjacency, dendrogram)
>>> np.round(score, 2)
0.33
```

References

Dasgupta, S. (2016). A cost function for similarity-based hierarchical clustering. Proceedings of ACM symposium on Theory of Computing.

sknetwork.hierarchy.**tree_sampling_divergence**(*adjacency: scipy.sparse.csr.csr_matrix*, *dendrogram: numpy.ndarray*, *weights: str = 'degree'*, *normalized: bool = True*) → float

Tree sampling divergence of a hierarchy (quality metric).

Parameters

- **adjacency** – Adjacency matrix of the graph.
- **dendrogram** – Dendrogram.
- **weights** – Weights of nodes. `'degree'` (default) or `'uniform'`.
- **normalized** – If `True`, normalized score (between 0 and 1).

Returns **score** – Score.

Return type float

Example

```
>>> from sknetwork.hierarchy import tree_sampling_divergence, Paris
>>> from sknetwork.data import house
>>> paris = Paris()
>>> adjacency = house()
>>> dendrogram = paris.fit_transform(adjacency)
>>> score = tree_sampling_divergence(adjacency, dendrogram)
>>> np.round(score, 2)
0.05
```

References

Charpentier, B. & Bonald, T. (2019). Tree Sampling Divergence: An Information-Theoretic Metric for Hierarchical Graph Clustering. Proceedings of IJCAI.

3.9.5 Cuts

sknetwork.hierarchy.**cut_straight**(*dendrogram: numpy.ndarray*, *n_clusters: Optional[int] = None*, *threshold: Optional[float] = None*, *sort_clusters: bool = True*, *return_dendrogram: bool = False*) → Union[numpy.ndarray, Tuple[numpy.ndarray, numpy.ndarray]]

Cut a dendrogram and return the corresponding clustering.

Parameters

- **dendrogram** – Dendrogram.
- **n_clusters** – Number of clusters (optional). The number of clusters can be larger than n_clusters in case of equal heights in the dendrogram.
- **threshold** – Threshold on height (optional). If both n_clusters and threshold are `None`, n_clusters is set to 2.
- **sort_clusters** – If `True`, sorts clusters in decreasing order of size.
- **return_dendrogram** – If `True`, returns the dendrogram formed by the clusters up to the root.

Returns

- **labels** (*np.ndarray*) – Cluster of each node.
- **dendrogram_aggregate** (*np.ndarray*) – Dendrogram starting from clusters (leaves = clusters).

Example

```
>>> from sknetwork.hierarchy import cut_straight
>>> dendrogram = np.array([[0, 1, 0, 2], [2, 3, 1, 3]])
>>> cut_straight(dendrogram)
array([0, 0, 1])
```

sknetwork.hierarchy.**cut_balanced**(*dendrogram: numpy.ndarray*, *max_cluster_size: int = 20*, *sort_clusters: bool = True*, *return_dendrogram: bool = False*) → Union[numpy.ndarray, Tuple[numpy.ndarray, numpy.ndarray]]

Cuts a dendrogram with a constraint on the cluster size and returns the corresponding clustering.

Parameters

- **dendrogram** – Dendrogram
- **max_cluster_size** – Maximum size of each cluster.
- **sort_clusters** – If `True`, sort labels in decreasing order of cluster size.
- **return_dendrogram** – If `True`, returns the dendrogram formed by the clusters up to the root.

Returns

- **labels** (*np.ndarray*) – Label of each node.
- **dendrogram_aggregate** (*np.ndarray*) – Dendrogram starting from clusters (leaves = clusters).

Example

```
>>> from sknetwork.hierarchy import cut_balanced
>>> dendrogram = np.array([[0, 1, 0, 2], [2, 3, 1, 3]])
>>> cut_balanced(dendrogram, 2)
array([0, 0, 1])
```

3.10 Ranking

Node ranking algorithms.

The attribute `scores_` assigns a score of importance to each node of the graph.

3.10.1 PageRank

class sknetwork.ranking.**PageRank**(*damping_factor: float = 0.85*, *solver: str = 'piteration'*, *n_iter: int = 10*, *tol: float = 1e-06*)

PageRank of each node, corresponding to its frequency of visit by a random walk.

The random walk restarts with some fixed probability. The restart distribution can be personalized by the user. This variant is known as Personalized PageRank.

Parameters

- **damping_factor** (*float*) – Probability to continue the random walk.
- **solver** (*str*) –
  - `'piteration'`, use power iteration for a given number of iterations.
  - `'diteration'`, use asynchronous parallel diffusion for a given number of iterations.
  - `'lanczos'`, use eigensolver with a given tolerance.
  - `'bicgstab'`, use Biconjugate Gradient Stabilized method for a given tolerance.
  - `'RH'`, use a Ruffini-Horner polynomial evaluation.
  - `'push'`, use push-based algorithm for a given tolerance
- **n_iter** (*int*) – Number of iterations for some solvers.
- **tol** (*float*) – Tolerance for the convergence of some solvers.

Variables

- **scores_** (*np.ndarray*) – PageRank score of each node.
- **scores_row_** (*np.ndarray*) – Scores of rows, for bipartite graphs.
- **scores_col_** (*np.ndarray*) – Scores of columns, for bipartite graphs.

Example

```
>>> from sknetwork.ranking import PageRank
>>> from sknetwork.data import house
>>> pagerank = PageRank()
>>> adjacency = house()
>>> seeds = {0: 1}
>>> scores = pagerank.fit_transform(adjacency, seeds)
>>> np.round(scores, 2)
array([0.29, 0.24, 0.12, 0.12, 0.24])
```

References

Page, L., Brin, S., Motwani, R., & Winograd, T. (1999). The PageRank citation ranking: Bringing order to the web. Stanford InfoLab.

fit(*input_matrix: Union[scipy.sparse.csr.csr_matrix, numpy.ndarray, scipy.sparse.linalg.interface.LinearOperator]*, *seeds: Optional[Union[numpy.ndarray, dict]] = None*, *seeds_row: Optional[Union[numpy.ndarray, dict]] = None*, *seeds_col: Optional[Union[numpy.ndarray, dict]] = None*, *force_bipartite: bool = False*) → *sknetwork.ranking.pagerank.PageRank*

Fit algorithm to data.

Parameters

- **input_matrix** – Adjacency matrix or biadjacency matrix of the graph.
- **seeds** – Parameter to be used for Personalized PageRank. Restart distribution as a vector or a dict (node: weight). If `None`, the uniform distribution is used (no personalization, default).
- **seeds_row** – Parameter to be used for Personalized PageRank on bipartite graphs. Restart distribution as vectors or dicts on rows, columns (node: weight). If both seeds_row and seeds_col are `None` (default), the uniform distribution on rows is used.
- **seeds_col** – Parameter to be used for Personalized PageRank on bipartite graphs. Restart distribution as vectors or dicts on rows, columns (node: weight). If both seeds_row and seeds_col are `None` (default), the uniform distribution on rows is used.
- **force_bipartite** – If `True`, consider the input matrix as the biadjacency matrix of a bipartite graph.

Returns self

Return type *PageRank*

fit_transform(**args*, ***kwargs*) → numpy.ndarray

Fit algorithm to data and return the scores. Same parameters as the `fit` method.

Returns **scores** – Scores.

Return type np.ndarray

3.10.2 Katz

class sknetwork.ranking.**Katz**(*damping_factor: float = 0.5*, *path_length: int = 4*)

Katz centrality, defined by:

$\sum_{k=1}^{K} \alpha^k (A^k)^T \mathbf{1}$.

Parameters

- **damping_factor** (*float*) – Decay parameter for path contributions.
- **path_length** (*int*) – Maximum length of the paths to take into account.

Variables

- **scores_** (*np.ndarray*) – Score of each node.
- **scores_row_** (*np.ndarray*) – Scores of rows, for bipartite graphs.
- **scores_col_** (*np.ndarray*) – Scores of columns, for bipartite graphs.

Examples

```
>>> from sknetwork.data.toy_graphs import house
>>> adjacency = house()
>>> katz = Katz()
>>> scores = katz.fit_transform(adjacency)
>>> np.round(scores, 2)
array([6.5 , 8.25, 5.62, 5.62, 8.25])
```

References

Katz, L. (1953). A new status index derived from sociometric analysis. Psychometrika, 18(1), 39-43.

fit(*input_matrix: Union[scipy.sparse.csr.csr_matrix, numpy.ndarray, scipy.sparse.linalg.interface.LinearOperator]*) → *sknetwork.ranking.katz.Katz*

Katz centrality.

Parameters **input_matrix** – Adjacency matrix or biadjacency matrix of the graph.

Returns **self**

Return type *Katz*

fit_transform(**args*, ***kwargs*) → numpy.ndarray

Fit algorithm to data and return the scores. Same parameters as the `fit` method.

Returns **scores** – Scores.

Return type np.ndarray

3.10.3 HITS

class sknetwork.ranking.**HITS**(*solver: Union[str, sknetwork.linalg.svd_solver.SVDSolver] = 'lanczos'*)

Hub and authority scores of each node. For bipartite graphs, the hub score is computed on rows and the authority score on columns.

Parameters **solver** (`'lanczos'` (default, Lanczos algorithm) or `SVDSolver` (custom solver)) – Which solver to use.

Variables

- **scores_** (*np.ndarray*) – Hub score of each node.
- **scores_row_** (*np.ndarray*) – Hub score of each row, for bipartite graphs.
- **scores_col_** (*np.ndarray*) – Authority score of each column, for bipartite graphs.

Example

```
>>> from sknetwork.ranking import HITS
>>> from sknetwork.data import star_wars
>>> hits = HITS()
>>> biadjacency = star_wars()
>>> scores = hits.fit_transform(biadjacency)
>>> np.round(scores, 2)
array([0.5 , 0.23, 0.69, 0.46])
```

References

Kleinberg, J. M. (1999). Authoritative sources in a hyperlinked environment. Journal of the ACM, 46(5), 604-632.

fit(*adjacency: Union[scipy.sparse.csr.csr_matrix, numpy.ndarray]*) → *sknetwork.ranking.hits.HITS*

Compute HITS algorithm with a spectral method.

Parameters **adjacency** – Adjacency or biadjacency matrix of the graph.

Returns **self**

Return type *HITS*

fit_transform(**args*, ***kwargs*) → numpy.ndarray

Fit algorithm to data and return the scores. Same parameters as the `fit` method.

Returns **scores** – Scores.

Return type np.ndarray

3.10.4 Betweenness centrality

class sknetwork.ranking.**Betweenness**(*normalized: bool = False*)

Betweenness centrality, based on Brandes' algorithm.

Variables **scores_** (*np.ndarray*) – Betweenness centrality value of each node

Example

```
>>> from sknetwork.ranking import Betweenness
>>> from sknetwork.data.toy_graphs import bow_tie
>>> betweenness = Betweenness()
>>> adjacency = bow_tie()
>>> scores = betweenness.fit_transform(adjacency)
>>> scores
array([4., 0., 0., 0., 0.])
```

References

Brandes, Ulrik (2001). A faster algorithm for betweenness centrality. Journal of Mathematical Sociology.

fit(*adjacency: Union[scipy.sparse.csr.csr_matrix, numpy.ndarray]*) → *sknetwork.ranking.betweenness.Betweenness*

Fit Algorithm to the data.

fit_transform(**args*, ***kwargs*) → numpy.ndarray

Fit algorithm to data and return the scores. Same parameters as the `fit` method.

Returns **scores** – Scores.

Return type np.ndarray

3.10.5 Closeness centrality

class sknetwork.ranking.**Closeness**(*method: str = 'exact'*, *tol: float = 0.1*, *n_jobs: Optional[int] = None*)

Closeness centrality of each node in a connected graph, corresponding to the average length of the shortest paths from that node to all the other ones.

For a directed graph, the closeness centrality is computed in terms of outgoing paths.

Parameters

- **method** – Denotes if the results should be exact or approximate.
- **tol** (*float*) – If `method=='approximate'`, the allowed tolerance on each score entry.
- **n_jobs** – If an integer value is given, denotes the number of workers to use (-1 means the maximum number will be used). If `None`, no parallel computations are made.

Variables **scores_** (*np.ndarray*) – Closeness centrality of each node.

Example

```
>>> from sknetwork.ranking import Closeness
>>> from sknetwork.data import cyclic_digraph
>>> closeness = Closeness()
>>> adjacency = cyclic_digraph(3)
>>> scores = closeness.fit_transform(adjacency)
>>> np.round(scores, 2)
array([0.67, 0.67, 0.67])
```

References

Eppstein, D., & Wang, J. (2001, January). Fast approximation of centrality. In Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms (pp. 228-229). Society for Industrial and Applied Mathematics.

fit(*adjacency: Union[scipy.sparse.csr.csr_matrix, numpy.ndarray]*) → *sknetwork.ranking.closeness.Closeness*

Closeness centrality for connected graphs.

Parameters **adjacency** – Adjacency matrix of the graph.

Returns **self**

Return type *Closeness*

fit_transform(**args*, ***kwargs*) → numpy.ndarray

Fit algorithm to data and return the scores. Same parameters as the `fit` method.

Returns **scores** – Scores.

Return type np.ndarray

3.10.6 Harmonic centrality

class sknetwork.ranking.**Harmonic**(*n_jobs: Optional[int] = None*)

Harmonic centrality of each node in a connected graph, corresponding to the average inverse length of the shortest paths from that node to all the other ones.

For a directed graph, the harmonic centrality is computed in terms of outgoing paths.

Parameters **n_jobs** – If an integer value is given, denotes the number of workers to use (-1 means the maximum number will be used). If `None`, no parallel computations are made.

Variables **scores_** (*np.ndarray*) – Score of each node.

Example

```
>>> from sknetwork.ranking import Harmonic
>>> from sknetwork.data import house
>>> harmonic = Harmonic()
>>> adjacency = house()
>>> scores = harmonic.fit_transform(adjacency)
>>> np.round(scores, 2)
array([3. , 3.5, 3. , 3. , 3.5])
```

References

Marchiori, M., & Latora, V. (2000). Harmony in the small-world. Physica A: Statistical Mechanics and its Applications, 285(3-4), 539-546.

fit(*adjacency: Union[scipy.sparse.csr.csr_matrix, numpy.ndarray]*) → *sknetwork.ranking.harmonic.Harmonic*

Harmonic centrality for connected graphs.

Parameters **adjacency** – Adjacency matrix of the graph.

Returns **self**

Return type *Harmonic*

fit_transform(**args*, ***kwargs*) → numpy.ndarray

Fit algorithm to data and return the scores. Same parameters as the `fit` method.

Returns **scores** – Scores.

Return type np.ndarray

3.10.7 Post-processing

sknetwork.ranking.**top_k**(*scores: numpy.ndarray*, *k: int = 1*)

Index of the k elements with highest value.

Parameters

- **scores** (*np.ndarray*) – Array of values.
- **k** (*int*) – Number of elements to return.

Examples

```
>>> scores = np.array([0, 1, 0, 0.5])
>>> top_k(scores, k=2)
array([1, 3])
```

Notes

This is a basic implementation that sorts the entire array to find its top k elements.

3.11 Classification

Node classification algorithms.

The attribute `labels_` assigns a label to each node of the graph.

3.11.1 PageRank

class sknetwork.classification.**PageRankClassifier**(*damping_factor: float = 0.85*, *solver: str = 'piteration'*, *n_iter: int = 10*, *tol: float = 0.0*, *n_jobs: Optional[int] = None*, *verbose: bool = False*)

Node classification by multiple personalized PageRanks.

Parameters

- **damping_factor** – Probability to continue the random walk.
- **solver** (`str`) – Which solver to use: 'piteration', 'diteration', 'bicgstab', 'lanczos'.
- **n_iter** (*int*) – Number of iterations for some of the solvers such as `'piteration'` or `'diteration'`.
- **tol** (*float*) – Tolerance for the convergence of some solvers such as `'bicgstab'` or `'lanczos'`.

Variables

- **labels_** (*np.ndarray, shape (n_labels,)*) – Label of each node.
- **membership_** (*sparse.csr_matrix, shape (n_row, n_labels)*) – Membership matrix.
- **labels_row_** (*np.ndarray*) – Labels of rows, for bipartite graphs.
- **labels_col_** (*np.ndarray*) – Labels of columns, for bipartite graphs.
- **membership_row_** (*sparse.csr_matrix, shape (n_row, n_labels)*) – Membership matrix of rows, for bipartite graphs.
- **membership_col_** (*sparse.csr_matrix, shape (n_col, n_labels)*) – Membership matrix of columns, for bipartite graphs.

Example

```
>>> from sknetwork.classification import PageRankClassifier
>>> from sknetwork.data import karate_club
>>> pagerank = PageRankClassifier()
>>> graph = karate_club(metadata=True)
>>> adjacency = graph.adjacency
>>> labels_true = graph.labels
>>> seeds = {0: labels_true[0], 33: labels_true[33]}
```

(continues on next page)

(continued from previous page)

```
>>> labels_pred = pagerank.fit_transform(adjacency, seeds)
>>> np.round(np.mean(labels_pred == labels_true), 2)
0.97
```

References

Lin, F., & Cohen, W. W. (2010). Semi-supervised classification of network data using very few labels. In IEEE International Conference on Advances in Social Networks Analysis and Mining.

fit(*input_matrix: Union[scipy.sparse.csr.csr_matrix, numpy.ndarray]*, *seeds: Optional[Union[numpy.ndarray, dict]] = None*, *seeds_row: Optional[Union[numpy.ndarray, dict]] = None*, *seeds_col: Optional[Union[numpy.ndarray, dict]] = None*) → sknetwork.classification.base_rank.RankClassifier
Fit algorithm to data.

Parameters

- **input_matrix** – Adjacency matrix or biadjacency matrix of the graph.
- **seeds** – Seed nodes (labels as dictionary or array; negative values ignored).
- **seeds_row** – Seed rows and columns (for bipartite graphs).
- **seeds_col** – Seed rows and columns (for bipartite graphs).

Returns **self**

Return type RankClassifier

fit_transform(**args*, ***kwargs*) → numpy.ndarray
Fit algorithm to the data and return the labels. Same parameters as the `fit` method.

Returns **labels** – Labels.

Return type np.ndarray

score(*label: int*)
Classification scores for a given label.

Parameters **label** (*int*) – The label index of the class.

Returns **scores** – Classification scores of shape (number of nodes,).

Return type np.ndarray

3.11.2 Diffusion

class sknetwork.classification.**DiffusionClassifier**(*n_iter: int = 10*, *damping_factor: Optional[float] = None*, *centering: bool = True*, *n_jobs: Optional[int] = None*)

Node classification using multiple diffusions.

Parameters

- **n_iter** (*int*) – Number of steps of the diffusion in discrete time (must be positive).
- **damping_factor** (*float (optional)*) – Damping factor (default value = 1).
- **centering** (*bool*) – Whether to center the temperatures with respect to the mean after diffusion (default = True).

- **n_jobs** (*int*) – If positive, number of parallel jobs allowed (-1 means maximum number). If None, no parallel computations are made.

Variables

- **labels_** (*np.ndarray, shape (n_labels,)*) – Label of each node.
- **membership_** (*sparse.csr_matrix, shape (n_row, n_labels)*) – Membership matrix.
- **labels_row_** (*np.ndarray*) – Labels of rows, for bipartite graphs.
- **labels_col_** (*np.ndarray*) – Labels of columns, for bipartite graphs.
- **membership_row_** (*sparse.csr_matrix, shape (n_row, n_labels)*) – Membership matrix of rows, for bipartite graphs.
- **membership_col_** (*sparse.csr_matrix, shape (n_col, n_labels)*) – Membership matrix of columns, for bipartite graphs.

Example

```
>>> from sknetwork.data import karate_club
>>> diffusion = DiffusionClassifier()
>>> graph = karate_club(metadata=True)
>>> adjacency = graph.adjacency
>>> labels_true = graph.labels
>>> seeds = {0: labels_true[0], 33: labels_true[33]}
>>> labels_pred = diffusion.fit_transform(adjacency, seeds)
>>> np.round(np.mean(labels_pred == labels_true), 2)
0.94
```

References

- de Lara, N., & Bonald, T. (2020). *A Consistent Diffusion-Based Algorithm for Semi-Supervised Classification on Graphs. <https://arxiv.org/pdf/2008.11944.pdf>* arXiv preprint arXiv:2008.11944.
- Zhu, X., Lafferty, J., & Rosenfeld, R. (2005). *Semi-supervised learning with graphs <http://pages.cs.wisc.edu/~jerryzhu/machineteaching/pub/thesis.pdf>* (Doctoral dissertation, Carnegie Mellon University, language technologies institute, school of computer science).

fit(*input_matrix: Union[scipy.sparse.csr.csr_matrix, numpy.ndarray]*, *seeds: Optional[Union[numpy.ndarray, dict]] = None*, *seeds_row: Optional[Union[numpy.ndarray, dict]] = None*, *seeds_col: Optional[Union[numpy.ndarray, dict]] = None*) → sknetwork.classification.base_rank.RankClassifier
Fit algorithm to data.

Parameters

- **input_matrix** – Adjacency matrix or biadjacency matrix of the graph.
- **seeds** – Seed nodes (labels as dictionary or array; negative values ignored).
- **seeds_row** – Seed rows and columns (for bipartite graphs).
- **seeds_col** – Seed rows and columns (for bipartite graphs).

Returns self

Return type RankClassifier

fit_transform(**args*, ***kwargs*) → numpy.ndarray
Fit algorithm to the data and return the labels. Same parameters as the fit method.

Returns **labels** – Labels.

Return type np.ndarray

score(*label: int*)
Classification scores for a given label.

Parameters **label** (*int*) – The label index of the class.

Returns **scores** – Classification scores of shape (number of nodes,).

Return type np.ndarray

3.11.3 Dirichlet

class sknetwork.classification.**DirichletClassifier**(*n_iter: int = 10*, *damping_factor: Optional[float] = None*, *centering: bool = True*, *n_jobs: Optional[int] = None*, *verbose: bool = False*)
Node classification using multiple Dirichlet problems.

Parameters

- **n_iter** (*int*) – If positive, the solution to the Dirichlet problem is approximated by power iteration for n_iter steps. Otherwise, the solution is computed through BiConjugate Stabilized Gradient descent.
- **damping_factor** (*float (optional)*) – Damping factor (default value = 1).
- **centering** (*bool*) – Whether to center the temperatures with respect to the mean after diffusion (default = True).
- **n_jobs** (*int*) – If an integer value is given, denotes the number of workers to use (-1 means the maximum number will be used). If None, no parallel computations are made.
- **verbose** – Verbose mode.

Variables

- **labels_** (*np.ndarray, shape (n_labels,)*) – Label of each node.
- **membership_** (*sparse.csr_matrix, shape (n_row, n_labels)*) – Membership matrix.
- **labels_row_** (*np.ndarray*) – Labels of rows, for bipartite graphs.
- **labels_col_** (*np.ndarray*) – Labels of columns, for bipartite graphs.
- **membership_row_** (*sparse.csr_matrix, shape (n_row, n_labels)*) – Membership matrix of rows, for bipartite graphs.
- **membership_col_** (*sparse.csr_matrix, shape (n_col, n_labels)*) – Membership matrix of columns, for bipartite graphs.

Example

```
>>> from sknetwork.data import karate_club
>>> dirichlet = DirichletClassifier()
>>> graph = karate_club(metadata=True)
>>> adjacency = graph.adjacency
>>> labels_true = graph.labels
>>> seeds = {0: labels_true[0], 33: labels_true[33]}
>>> labels_pred = dirichlet.fit_transform(adjacency, seeds)
>>> np.round(np.mean(labels_pred == labels_true), 2)
0.97
```

References

Zhu, X., Lafferty, J., & Rosenfeld, R. (2005). Semi-supervised learning with graphs (Doctoral dissertation, Carnegie Mellon University, language technologies institute, school of computer science).

fit(*input_matrix: Union[scipy.sparse.csr.csr_matrix, numpy.ndarray]*, *seeds: Optional[Union[numpy.ndarray, dict]] = None*, *seeds_row: Optional[Union[numpy.ndarray, dict]] = None*, *seeds_col: Optional[Union[numpy.ndarray, dict]] = None*) → sknetwork.classification.base_rank.RankClassifier
Fit algorithm to data.

Parameters

- **input_matrix** – Adjacency matrix or biadjacency matrix of the graph.
- **seeds** – Seed nodes (labels as dictionary or array; negative values ignored).
- **seeds_row** – Seed rows and columns (for bipartite graphs).
- **seeds_col** – Seed rows and columns (for bipartite graphs).

Returns self

Return type RankClassifier

fit_transform(**args*, ***kwargs*) → numpy.ndarray
Fit algorithm to the data and return the labels. Same parameters as the `fit` method.

Returns **labels** – Labels.

Return type np.ndarray

score(*label: int*)
Classification scores for a given label.

Parameters **label** (*int*) – The label index of the class.

Returns **scores** – Classification scores of shape (number of nodes,).

Return type np.ndarray

3.11.4 Propagation

class sknetwork.classification.**Propagation**(*n_iter: float = - 1*, *node_order: Optional[str] = None*, *weighted: bool = True*)

Node classification by label propagation.

Parameters

- **n_iter** (*float*) – Maximum number of iterations (-1 for infinity).
- **node_order** (*str*) –
  - *'random'*: node labels are updated in random order.
  - *'increasing'*: node labels are updated by increasing order of (in-)weight.
  - *'decreasing'*: node labels are updated by decreasing order of (in-)weight.
  - Otherwise, node labels are updated by index order.
- **weighted** (*bool*) – If True, the vote of each neighbor is proportional to the edge weight. Otherwise, all votes have weight 1.

Variables

- **labels_** (*np.ndarray, shape (n_labels,)*) – Label of each node.
- **membership_** (*sparse.csr_matrix, shape (n_row, n_labels)*) – Membership matrix.
- **labels_row_** (*np.ndarray*) – Labels of rows, for bipartite graphs.
- **labels_col_** (*np.ndarray*) – Labels of columns, for bipartite graphs.
- **membership_row_** (*sparse.csr_matrix, shape (n_row, n_labels)*) – Membership matrix of rows, for bipartite graphs.
- **membership_col_** (*sparse.csr_matrix, shape (n_col, n_labels)*) – Membership matrix of columns, for bipartite graphs.

Example

```
>>> from sknetwork.classification import Propagation
>>> from sknetwork.data import karate_club
>>> propagation = Propagation()
>>> graph = karate_club(metadata=True)
>>> adjacency = graph.adjacency
>>> labels_true = graph.labels
>>> seeds = {0: labels_true[0], 33: labels_true[33]}
>>> labels_pred = propagation.fit_transform(adjacency, seeds)
>>> np.round(np.mean(labels_pred == labels_true), 2)
0.94
```

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.

fit(*input_matrix: Union[scipy.sparse.csr.csr_matrix, numpy.ndarray]*, *seeds: Optional[Union[numpy.ndarray, dict]] = None*, *seeds_row: Optional[Union[numpy.ndarray, dict]] = None*, *seeds_col: Optional[Union[numpy.ndarray, dict]] = None*) → *sknetwork.classification.propagation.Propagation*
Node classification by label propagation.

Parameters

- **input_matrix** – Adjacency matrix or biadjacency matrix of the graph.
- **seeds** – Seed nodes. Can be a dict {node: label} or an array where "-1" means no label.
- **seeds_row** – Seeds of rows and columns (for bipartite graphs).
- **seeds_col** – Seeds of rows and columns (for bipartite graphs).

Returns self

Return type *Propagation*

fit_transform(**args*, ***kwargs*) → numpy.ndarray
Fit algorithm to the data and return the labels. Same parameters as the `fit` method.

Returns **labels** – Labels.

Return type np.ndarray

score(*label: int*)
Classification scores for a given label.

Parameters **label** (*int*) – The label index of the class.

Returns **scores** – Classification scores of shape (number of nodes,).

Return type np.ndarray

3.11.5 Nearest neighbors

class sknetwork.classification.**KNN**(*embedding_method: sknetwork.embedding.base.BaseEmbedding = GSVD(n_components=10, regularization=None, factor_row=0.5, factor_col=0.5, factor_singular=0.0, normalized=True, solver='lanczos')*, *n_neighbors: int = 5*, *factor_distance: float = 2*, *leaf_size: int = 16*, *p: float = 2*, *tol_nn: float = 0.01*, *n_jobs: Optional[int] = None*)
Node classification by K-nearest neighbors in the embedding space.

For bigraphs, classify rows only (see `BiKNN` for joint classification of rows and columns).

Parameters

- **embedding_method** – Which algorithm to use to project the nodes in vector space. Default is `GSVD`.
- **n_neighbors** – Number of nearest neighbors to consider.
- **factor_distance** – Power weighting factor α applied to the distance to each neighbor. Neighbor at distance :math:`d` has weight $1/d^{\alpha}$. Default is 2.

- **leaf_size** – Leaf size passed to KDTree.
- **p** – Which Minkowski p-norm to use. Default is 2 (Euclidean distance).
- **tol_nn** – Tolerance in nearest neighbors search; the k-th returned value is guaranteed to be no further than `1 + tol_nn` times the distance to the actual k-th nearest neighbor.
- **n_jobs** – Number of jobs to schedule for parallel processing. If -1 is given all processors are used.

Variables

- **labels_** (*np.ndarray, shape (n_labels,)*) – Label of each node.
- **membership_** (*sparse.csr_matrix, shape (n_row, n_labels)*) – Membership matrix.
- **labels_row_** (*np.ndarray*) – Labels of rows, for bipartite graphs.
- **labels_col_** (*np.ndarray*) – Labels of columns, for bipartite graphs.
- **membership_row_** (*sparse.csr_matrix, shape (n_row, n_labels)*) – Membership matrix of rows, for bipartite graphs.
- **membership_col_** (*sparse.csr_matrix, shape (n_col, n_labels)*) – Membership matrix of columns, for bipartite graphs.

Example

```
>>> from sknetwork.classification import KNN
>>> from sknetwork.embedding import GSVD
>>> from sknetwork.data import karate_club
>>> knn = KNN(GSVD(3), n_neighbors=1)
>>> graph = karate_club(metadata=True)
>>> adjacency = graph.adjacency
>>> labels_true = graph.labels
>>> seeds = {0: labels_true[0], 33: labels_true[33]}
>>> labels_pred = knn.fit_transform(adjacency, seeds)
>>> np.round(np.mean(labels_pred == labels_true), 2)
0.97
```

fit(*input_matrix: Union[scipy.sparse.csr.csr_matrix, numpy.ndarray]*, *seeds: Optional[Union[numpy.ndarray, dict]] = None*, *seeds_row: Optional[Union[numpy.ndarray, dict]] = None*, *seeds_col: Optional[Union[numpy.ndarray, dict]] = None*) → *sknetwork.classification.knn.KNN*
Node classification by k-nearest neighbors in the embedding space.

Parameters

- **input_matrix** – Adjacency matrix or biadjacency matrix of the graph.
- **seeds** – Seed nodes. Can be a dict {node: label} or an array where "-1" means no label.
- **seeds_row** – Seeds of rows and columns (for bipartite graphs).
- **seeds_col** – Seeds of rows and columns (for bipartite graphs).

Returns self

Return type *KNN*

fit_transform(**args*, ***kwargs*) → numpy.ndarray
Fit algorithm to the data and return the labels. Same parameters as the `fit` method.

Returns **labels** – Labels.

Return type np.ndarray

score(*label: int*)
Classification scores for a given label.

Parameters **label** (*int*) – The label index of the class.

Returns **scores** – Classification scores of shape (number of nodes,).

Return type np.ndarray

3.11.6 Metrics

sknetwork.classification.**accuracy_score**(*y_true: numpy.ndarray*, *y_pred: numpy.ndarray*) → float

Accuracy: number of correctly labeled samples over total number of elements. In the case of binary classification, this is

$$P = \frac{TP + TN}{TP + TN + FP + FN}.$$

Parameters

- **y_true** (*np.ndarray*) – True labels.
- **y_pred** (*np.ndarray*) – Predicted labels

Returns **precision** – A score between 0 and 1.

Return type float

Examples

```
>>> import numpy as np
>>> y_true = np.array([0, 0, 1, 1])
>>> y_pred = np.array([0, 0, 0, 1])
>>> accuracy_score(y_true, y_pred)
0.75
```

3.12 Regression

Regression algorithms.

The attribute `values_` assigns a value to each node of the graph.

3.12.1 Diffusion

class sknetwork.regression.**Diffusion**(*n_iter: int = 3*, *damping_factor: Optional[float] = None*)

Regression by diffusion along the edges (heat equation).

Parameters

- **n_iter** (*int*) – Number of steps of the diffusion in discrete time (must be positive).
- **damping_factor** (*float (optional)*) – Damping factor (default value = 1).

Variables

- **values_** (*np.ndarray*) – Value of each node (= temperature).
- **values_row_** (*np.ndarray*) – Values of rows, for bipartite graphs.
- **values_col_** (*np.ndarray*) – Values of columns, for bipartite graphs.

Example

```
>>> from sknetwork.data import house
>>> diffusion = Diffusion(n_iter=2)
>>> adjacency = house()
>>> seeds = {0: 1, 2: 0}
>>> values = diffusion.fit_transform(adjacency, seeds)
>>> np.round(values, 2)
array([0.58, 0.56, 0.38, 0.58, 0.42])
```

References

Chung, F. (2007). The heat kernel as the pagerank of a graph. Proceedings of the National Academy of Sciences.

fit(*input_matrix: Union[scipy.sparse.csr.csr_matrix, numpy.ndarray]*, *seeds: Optional[Union[numpy.ndarray, dict]] = None*, *seeds_row: Optional[Union[numpy.ndarray, dict]] = None*, *seeds_col: Optional[Union[numpy.ndarray, dict]] = None*, *init: Optional[float] = None*) → *sknetwork.regression.diffusion.Diffusion*

Compute the diffusion (temperatures at equilibrium).

Parameters

- **input_matrix** – Adjacency matrix or biadjacency matrix of the graph.
- **seeds** – Temperatures of seed nodes in initial state (dictionary or vector). Negative temperatures ignored.
- **seeds_row** – Temperatures of rows and columns for bipartite graphs. Negative temperatures ignored.
- **seeds_col** – Temperatures of rows and columns for bipartite graphs. Negative temperatures ignored.
- **init** – Temperature of non-seed nodes in initial state. If `None`, use the average temperature of seed nodes (default).

Returns self

Return type *Diffusion*

fit_transform(**args*, ***kwargs*) → numpy.ndarray
Fit algorithm to data and return the scores. Same parameters as the `fit` method.

Returns **values** – Values.

Return type np.ndarray

3.12.2 Dirichlet

class sknetwork.regression.**Dirichlet**(*n_iter: int = 10*, *damping_factor: Optional[float] = None*, *verbose: bool = False*)
Regression by the Dirichlet problem (heat diffusion with boundary constraints).

Parameters

- **n_iter** (*int*) – If positive, number of steps of the diffusion in discrete time. Otherwise, solve the Dirichlet problem by the bi-conjugate gradient stabilized method.
- **damping_factor** (*float (optional)*) – Damping factor (default value = 1).
- **verbose** (*bool*) – Verbose mode.

Variables

- **values_** (*np.ndarray*) – Value of each node (= temperature).
- **values_row_** (*np.ndarray*) – Values of rows, for bipartite graphs.
- **values_col_** (*np.ndarray*) – Values of columns, for bipartite graphs.

Example

```
>>> from sknetwork.regression import Dirichlet
>>> from sknetwork.data import house
>>> dirichlet = Dirichlet()
>>> adjacency = house()
>>> seeds = {0: 1, 2: 0}
>>> values = dirichlet.fit_transform(adjacency, seeds)
>>> np.round(values, 2)
array([1.  , 0.54, 0.  , 0.31, 0.62])
```

References

Chung, F. (2007). The heat kernel as the pagerank of a graph. Proceedings of the National Academy of Sciences.

fit(*input_matrix: Union[scipy.sparse.csr.csr_matrix, numpy.ndarray]*, *seeds: Optional[Union[numpy.ndarray, dict]] = None*, *seeds_row: Optional[Union[numpy.ndarray, dict]] = None*, *seeds_col: Optional[Union[numpy.ndarray, dict]] = None*, *init: Optional[float] = None*) → *sknetwork.regression.diffusion.Dirichlet*
Compute the solution to the Dirichlet problem (temperatures at equilibrium).

Parameters

- **input_matrix** – Adjacency matrix or biadjacency matrix of the graph.
- **seeds** – Temperatures of seed nodes (dictionary or vector). Negative temperatures ignored.

- **seeds_row** – Temperatures of rows and columns for bipartite graphs. Negative temperatures ignored.
- **seeds_col** – Temperatures of rows and columns for bipartite graphs. Negative temperatures ignored.
- **init** – Temperature of non-seed nodes in initial state. If None, use the average temperature of seed nodes (default).

Returns self

Return type *Dirichlet*

fit_transform(**args*, ***kwargs*) → numpy.ndarray
Fit algorithm to data and return the scores. Same parameters as the fit method.

Returns **values** – Values.

Return type np.ndarray

3.13 Embedding

Graph embedding algorithms.

The attribute embedding_ assigns a vector to each node of the graph.

3.13.1 Spectral

class sknetwork.embedding.**Spectral**(*n_components: int = 2*, *decomposition: str = 'rw'*, *regularization: float = - 1*, *normalized: bool = True*)
Spectral embedding of graphs, based the spectral decomposition of the Laplacian matrix $L = D - A$ or the transition matrix of the random walk $P = D^{-1}A$ (default), where D is the diagonal matrix of degrees.

Eigenvectors are considered in increasing order (for the Laplacian matrix L) or decreasing order (for the transition matrix of the random walk P) of eigenvalues, skipping the first.

Parameters

- **n_components** (int (default = 2)) – Dimension of the embedding space.
- **decomposition** (str (laplacian or rw, default = rw)) – Matrix used for the spectral decomposition.
- **regularization** (float (default = -1)) – Regularization factor α so that the adjacency matrix is $A + \alpha \frac{11^T}{n}$. If negative, regularization is applied only if the graph is disconnected; the regularization factor α is then set to the absolute value of the parameter.
- **normalized** (bool (default = True)) – If True, normalized the embedding so that each vector has norm 1 in the embedding space, i.e., each vector lies on the unit sphere.

Variables

- **embedding_** (*array, shape = (n, n_components)*) – Embedding of the nodes.
- **embedding_row_** (*array, shape = (n_row, n_components)*) – Embedding of the rows, for bipartite graphs.
- **embedding_col_** (*array, shape = (n_col, n_components)*) – Embedding of the columns, for bipartite graphs.
- **eigenvalues_** (*array, shape = (n_components)*) – Eigenvalues.

- **eigenvectors_** (*array, shape = (n, n_components)*) – Eigenvectors.

Example

```
>>> from sknetwork.embedding import Spectral
>>> from sknetwork.data import karate_club
>>> spectral = Spectral(n_components=3)
>>> adjacency = karate_club()
>>> embedding = spectral.fit_transform(adjacency)
>>> embedding.shape
(34, 3)
```

References

Belkin, M. & Niyogi, P. (2003). Laplacian Eigenmaps for Dimensionality Reduction and Data Representation, Neural computation.

fit(*input_matrix: Union[scipy.sparse.csr.csr_matrix, numpy.ndarray], force_bipartite: bool = False*) → *sknetwork.embedding.spectral.Spectral*

Compute the graph embedding.

If the input matrix B is not square (e.g., biadjacency matrix of a bipartite graph) or not symmetric (e.g., adjacency matrix of a directed graph), use the adjacency matrix

$$A = \begin{bmatrix} 0 & B \\ B^T & 0 \end{bmatrix}$$

and return the embedding for both rows and columns of the input matrix B.

Parameters

- **input_matrix** – Adjacency matrix or biadjacency matrix of the graph.
- **force_bipartite** (bool (default = `False`)) – If `True`, force the input matrix to be considered as a biadjacency matrix.

Returns self

Return type *Spectral*

fit_transform(**args*, ***kwargs*) → numpy.ndarray

Fit to data and return the embedding. Same parameters as the `fit` method.

Returns **embedding** – Embedding.

Return type np.ndarray

predict(*adjacency_vectors: Union[scipy.sparse.csr.csr_matrix, numpy.ndarray]*) → numpy.ndarray

Predict the embedding of new nodes, when possible (otherwise return 0).

Each new node is defined by its adjacency row vector.

Parameters **adjacency_vectors** – Adjacency vectors of nodes. Array of shape (n_col,) (single vector) or (n_vectors, n_col)

Returns **embedding_vectors** – Embedding of the nodes.

Return type np.ndarray

Example

```
>>> from sknetwork.embedding import Spectral
>>> from sknetwork.data import karate_club
>>> spectral = Spectral(n_components=3)
>>> adjacency = karate_club()
>>> adjacency_vector = np.arange(34) < 5
>>> _ = spectral.fit(adjacency)
>>> len(spectral.predict(adjacency_vector))
3
```

3.13.2 SVD

class sknetwork.embedding.**SVD**(*n_components=2*, *regularization: Optional[float] = None*, *factor_singular: float = 0.0*, *normalized: bool = False*, *solver: Union[str, sknetwork.linalg.svd_solver.SVDSolver] = 'lanczos'*)

Graph embedding by Singular Value Decomposition of the adjacency or biadjacency matrix of the graph.

Parameters

- **n_components** (*int*) – Dimension of the embedding.
- **regularization** (None or float (default = None)) – Regularization factor α so that the matrix is $A + \alpha \frac{1 1^T}{n}$.
- **factor_singular** (*float (default = 0.)*) – Power factor α applied to the singular values on right singular vectors. The embedding of rows and columns are respectively $U\Sigma^{1-\alpha}$ and $V\Sigma^{\alpha}$ where:
  - U is the matrix of left singular vectors, shape (n_row, n_components)
  - V is the matrix of right singular vectors, shape (n_col, n_components)
  - Σ is the diagonal matrix of singular values, shape (n_components, n_components)
- **normalized** (bool (default = False)) – If True, normalized the embedding so that each vector has norm 1 in the embedding space, i.e., each vector lies on the unit sphere.
- **solver** ('lanczos' (Lanczos algorithm, default) or SVDSolver (custom solver)) – Which solver to use.

Variables

- **embedding_** (*array, shape = (n, n_components)*) – Embedding of the nodes.
- **embedding_row_** (*array, shape = (n_row, n_components)*) – Embedding of the rows, for bipartite graphs.
- **embedding_col_** (*array, shape = (n_col, n_components)*) – Embedding of the columns, for bipartite graphs.
- **singular_values_** (*np.ndarray, shape = (n_components)*) – Singular values.
- **singular_vectors_left_** (*np.ndarray, shape = (n_row, n_components)*) – Left singular vectors.
- **singular_vectors_right_** (*np.ndarray, shape = (n_col, n_components)*) – Right singular vectors.

Example

```
>>> from sknetwork.embedding import SVD
>>> from sknetwork.data import karate_club
>>> svd = SVD()
>>> adjacency = karate_club()
>>> embedding = svd.fit_transform(adjacency)
>>> embedding.shape
(34, 2)
```

References

Abdi, H. (2007). Singular value decomposition (SVD) and generalized singular value decomposition. Encyclopedia of measurement and statistics.

fit(*input_matrix: Union[scipy.sparse.csr.csr_matrix, numpy.ndarray]*) → *sknetwork.embedding.svd.GSVD*
Compute the embedding of the graph.

Parameters **input_matrix** – Adjacency matrix or biadjacency matrix of the graph.

Returns **self**

Return type *GSVD*

fit_transform(**args*, ***kwargs*) → numpy.ndarray
Fit to data and return the embedding. Same parameters as the `fit` method.

Returns **embedding** – Embedding.

Return type np.ndarray

predict(*adjacency_vectors: Union[scipy.sparse.csr.csr_matrix, numpy.ndarray]*) → numpy.ndarray
Predict the embedding of new rows, defined by their adjacency vectors.

Parameters **adjacency_vectors** – Adjacency vectors of nodes. Array of shape (n_col,) (single vector) or (n_vectors, n_col)

Returns **embedding_vectors** – Embedding of the nodes.

Return type np.ndarray

3.13.3 GSVD

class sknetwork.embedding.**GSVD**(*n_components=2*, *regularization: Optional[float] = None*, *factor_row: float = 0.5*, *factor_col: float = 0.5*, *factor_singular: float = 0.0*, *normalized: bool = True*, *solver: Union[str, sknetwork.linalg.svd_solver.SVDSolver] = 'lanczos'*)
Graph embedding by Generalized Singular Value Decomposition of the adjacency or biadjacency matrix A. This is equivalent to the Singular Value Decomposition of the matrix $D_1^{-\alpha_1} A D_2^{-\alpha_2}$ where D_1, D_2 are the diagonal matrices of row weights and columns weights, respectively, and α_1, α_2 are parameters.

Parameters

- **n_components** (*int*) – Dimension of the embedding.
- **regularization** (None or float (default = None)) – Regularization factor α so that the matrix is $A + \alpha \frac{1 1^T}{n}$.

- **factor_row** (*float (default = 0.5)*) – Power factor α_1 applied to the diagonal matrix of row weights.
- **factor_col** (*float (default = 0.5)*) – Power factor α_2 applied to the diagonal matrix of column weights.
- **factor_singular** (*float (default = 0.)*) – Parameter α applied to the singular values on right singular vectors. The embedding of rows and columns are respectively $D_1^{-\alpha_1}U\Sigma^{1-\alpha}$ and $D_2^{-\alpha_2}V\Sigma^{\alpha}$ where:
  - U is the matrix of left singular vectors, shape (n_row, n_components)
  - V is the matrix of right singular vectors, shape (n_col, n_components)
  - Σ is the diagonal matrix of singular values, shape (n_components, n_components)
- **normalized** (bool (default = `True`)) – If `True`, normalized the embedding so that each vector has norm 1 in the embedding space, i.e., each vector lies on the unit sphere.
- **solver** (`'lanczos'` (Lanczos algorithm, default) or `SVDSolver` (custom solver)) – Which solver to use.

Variables

- **embedding_** (*array, shape = (n, n_components)*) – Embedding of the nodes.
- **embedding_row_** (*array, shape = (n_row, n_components)*) – Embedding of the rows, for bipartite graphs.
- **embedding_col_** (*array, shape = (n_col, n_components)*) – Embedding of the columns, for bipartite graphs.
- **singular_values_** (*np.ndarray, shape = (n_components)*) – Singular values.
- **singular_vectors_left_** (*np.ndarray, shape = (n_row, n_components)*) – Left singular vectors.
- **singular_vectors_right_** (*np.ndarray, shape = (n_col, n_components)*) – Right singular vectors.
- **weights_col_** (*np.ndarray, shape = (n2)*) – Weights applied to columns.

Example

```
>>> from sknetwork.embedding import GSVD
>>> from sknetwork.data import karate_club
>>> gsvd = GSVD()
>>> adjacency = karate_club()
>>> embedding = gsvd.fit_transform(adjacency)
>>> embedding.shape
(34, 2)
```

References

Abdi, H. (2007). Singular value decomposition (SVD) and generalized singular value decomposition. Encyclopedia of measurement and statistics, 907-912.

fit(*input_matrix: Union[scipy.sparse.csr.csr_matrix, numpy.ndarray]*) → *sknetwork.embedding.svd.GSVD*
Compute the embedding of the graph.

Parameters **input_matrix** – Adjacency matrix or biadjacency matrix of the graph.

Returns **self**

Return type *GSVD*

fit_transform(**args*, ***kwargs*) → numpy.ndarray
Fit to data and return the embedding. Same parameters as the `fit` method.

Returns **embedding** – Embedding.

Return type np.ndarray

predict(*adjacency_vectors: Union[scipy.sparse.csr.csr_matrix, numpy.ndarray]*) → numpy.ndarray
Predict the embedding of new rows, defined by their adjacency vectors.

Parameters **adjacency_vectors** – Adjacency vectors of nodes. Array of shape (n_col,) (single vector) or (n_vectors, n_col)

Returns **embedding_vectors** – Embedding of the nodes.

Return type np.ndarray

3.13.4 PCA

class sknetwork.embedding.**PCA**(*n_components=2*, *normalized: bool = False*, *solver: Union[str, sknetwork.linalg.svd_solver.SVDSolver] = 'lanczos'*)
Graph embedding by Principal Component Analysis of the adjacency or biadjacency matrix.

Parameters

- **n_components** (*int*) – Dimension of the embedding.
- **normalized** (bool (default = `False`)) – If `True`, normalized the embedding so that each vector has norm 1 in the embedding space, i.e., each vector lies on the unit sphere.
- **solver** (`'lanczos'` (Lanczos algorithm, default) or `SVDSolver` (custom solver)) – Which solver to use.

Variables

- **embedding_** (*array, shape = (n, n_components)*) – Embedding of the nodes.
- **embedding_row_** (*array, shape = (n_row, n_components)*) – Embedding of the rows, for bipartite graphs.
- **embedding_col_** (*array, shape = (n_col, n_components)*) – Embedding of the columns, for bipartite graphs.
- **singular_values_** (*np.ndarray, shape = (n_components)*) – Singular values.
- **singular_vectors_left_** (*np.ndarray, shape = (n_row, n_components)*) – Left singular vectors.
- **singular_vectors_right_** (*np.ndarray, shape = (n_col, n_components)*) – Right singular vectors.

Example

```
>>> from sknetwork.embedding import PCA
>>> from sknetwork.data import karate_club
>>> pca = PCA()
>>> adjacency = karate_club()
>>> embedding = pca.fit_transform(adjacency)
>>> embedding.shape
(34, 2)
```

References

Jolliffe, I.T. (2002). *Principal Component Analysis* Series: Springer Series in Statistics.

fit(*adjacency: Union[scipy.sparse.csr.csr_matrix, numpy.ndarray]*) → *sknetwork.embedding.svd.PCA*
Compute the embedding of the graph.

Parameters **adjacency** – Adjacency or biadjacency matrix of the graph.

Returns **self**

Return type *PCA*

fit_transform(**args*, ***kwargs*) → numpy.ndarray
Fit to data and return the embedding. Same parameters as the `fit` method.

Returns **embedding** – Embedding.

Return type np.ndarray

predict(*adjacency_vectors: Union[scipy.sparse.csr.csr_matrix, numpy.ndarray]*) → numpy.ndarray
Predict the embedding of new rows, defined by their adjacency vectors.

Parameters **adjacency_vectors** – Adjacency vectors of nodes. Array of shape (n_col,) (single vector) or (n_vectors, n_col)

Returns **embedding_vectors** – Embedding of the nodes.

Return type np.ndarray

3.13.5 Random Projection

class sknetwork.embedding.**RandomProjection**(*n_components: int = 2*, *alpha: float = 0.5*, *n_iter: int = 3*, *random_walk: bool = False*, *regularization: float = - 1*, *normalized: bool = True*, *random_state: Optional[int] = None*)
Embedding of graphs based the random projection of the adjacency matrix:

$(I + \alpha A + ... + (\alpha A)^K)G$

where A is the adjacency matrix, G is a random Gaussian matrix, α is some smoothing factor and K some non-negative integer.

Parameters

- **n_components** (*int (default = 2)*) – Dimension of the embedding space.
- **alpha** (*float (default = 0.5)*) – Smoothing parameter.
- **n_iter** (*int (default = 3)*) – Number of power iterations of the adjacency matrix.

- **random_walk** (bool (default = `False`)) – If `True`, use the transition matrix of the random walk, $P = D^{-1}A$, instead of the adjacency matrix.
- **regularization** (float (default = `-1`)) – Regularization factor α so that the matrix is $A + \alpha\frac{11^T}{n}$. If negative, regularization is applied only if the graph is disconnected (and then equal to the absolute value of the parameter).
- **normalized** (bool (default = `True`)) – If `True`, normalize the embedding so that each vector has norm 1 in the embedding space, i.e., each vector lies on the unit sphere.
- **random_state** (*int, optional*) – Seed used by the random number generator.

Variables

- **embedding_** (*array, shape = (n, n_components)*) – Embedding of the nodes.
- **embedding_row_** (*array, shape = (n_row, n_components)*) – Embedding of the rows, for bipartite graphs.
- **embedding_col_** (*array, shape = (n_col, n_components)*) – Embedding of the columns, for bipartite graphs.

Example

```
>>> from sknetwork.embedding import RandomProjection
>>> from sknetwork.data import karate_club
>>> projection = RandomProjection()
>>> adjacency = karate_club()
>>> embedding = projection.fit_transform(adjacency)
>>> embedding.shape
(34, 2)
```

References

Zhang, Z., Cui, P., Li, H., Wang, X., & Zhu, W. (2018). Billion-scale network embedding with iterative random projection, ICDM.

fit(*input_matrix: Union[scipy.sparse.csr.csr_matrix, numpy.ndarray], force_bipartite: bool = False*) → *sknetwork.embedding.random_projection.RandomProjection*

Compute the graph embedding.

Parameters

- **input_matrix** – Adjacency matrix or biadjacency matrix of the graph.
- **force_bipartite** (bool (default = `False`)) – If `True`, force the input matrix to be considered as a biadjacency matrix.

Returns **self**

Return type *RandomProjection*

fit_transform(**args*, ***kwargs*) → numpy.ndarray

Fit to data and return the embedding. Same parameters as the `fit` method.

Returns **embedding** – Embedding.

Return type np.ndarray

3.13.6 Louvain

class sknetwork.embedding.**LouvainEmbedding**(*resolution: float = 1*, *modularity: str = 'dugue'*, *tol_optimization: float = 0.001*, *tol_aggregation: float = 0.001*, *n_aggregations: int = - 1*, *shuffle_nodes: bool = False*, *random_state: Optional[Union[numpy.random.mtrand.RandomState, int]] = None*, *isolated_nodes: str = 'remove'*)

Embedding of graphs induced by Louvain clustering. Each component of the embedding corresponds to a cluster obtained by Louvain.

Parameters

- **resolution** (*float*) – Resolution parameter.
- **modularity** (*str*) – Which objective function to maximize. Can be `'dugue'`, `'newman'` or `'potts'`.
- **tol_optimization** – Minimum increase in the objective function to enter a new optimization pass.
- **tol_aggregation** – Minimum increase in the objective function to enter a new aggregation pass.
- **n_aggregations** – Maximum number of aggregations. A negative value is interpreted as no limit.
- **shuffle_nodes** – Enables node shuffling before optimization.
- **random_state** – Random number generator or random seed. If `None`, numpy.random is used.
- **isolated_nodes** (*str*) – What to do with isolated column nodes. Can be `'remove'` (default), `'merge'` or `'keep'`.

Variables

- **embedding_** (*array, shape = (n, n_components)*) – Embedding of the nodes.
- **embedding_row_** (*array, shape = (n_row, n_components)*) – Embedding of the rows, for bipartite graphs.
- **embedding_col_** (*array, shape = (n_col, n_components)*) – Embedding of the columns, for bipartite graphs.
- **labels_row_** (*np.ndarray*) – Labels of the rows (used to build the embedding of the columns).
- **labels_col_** (*np.ndarray*) – Labels of the columns (used to build the embedding of the rows).

Example

```
>>> from sknetwork.embedding import LouvainEmbedding
>>> from sknetwork.data import house
>>> louvain = LouvainEmbedding()
>>> adjacency = house()
>>> embedding = louvain.fit_transform(adjacency)
>>> embedding.shape
(5, 2)
```

fit(*input_matrix: scipy.sparse.csr.csr_matrix*, *force_bipartite: bool = False*)
Embedding of graphs from the clustering obtained with Louvain.

Parameters

- **input_matrix** – Adjacency matrix or biadjacency matrix of the graph.
- **force_bipartite** (bool (default = `False`)) – If `True`, force the input matrix to be considered as a biadjacency matrix.

Returns self

Return type `BiLouvainEmbedding`

fit_transform(**args*, ***kwargs*) → numpy.ndarray
Fit to data and return the embedding. Same parameters as the `fit` method.

Returns **embedding** – Embedding.

Return type np.ndarray

predict(*adjacency_vectors: Union[scipy.sparse.csr.csr_matrix, numpy.ndarray]*) → numpy.ndarray
Predict the embedding of new rows, defined by their adjacency vectors.

Parameters **adjacency_vectors** – Adjacency row vectors. Array of shape (n_col,) (single vector) or (n_vectors, n_col)

Returns **embedding_vectors** – Embedding of the nodes.

Return type np.ndarray

3.13.7 Hierarchical Louvain

class sknetwork.embedding.**LouvainNE**(*n_components: int = 2*, *scale: float = 0.1*, *resolution: float = 1*, *tol_optimization: float = 0.001*, *tol_aggregation: float = 0.001*, *n_aggregations: int = - 1*, *shuffle_nodes: bool = False*, *random_state: Optional[Union[numpy.random.mtrand.RandomState, int]] = None*, *verbose: bool = False*)
Embedding of graphs based on the hierarchical Louvain algorithm with random scattering per level.

Parameters

- **n_components** (*int*) – Dimension of the embedding.
- **scale** (*float*) – Dilution factor to be applied on the random vector to be added at each iteration of the clustering method.
- **resolution** – Resolution parameter.
- **tol_optimization** – Minimum increase in the objective function to enter a new optimization pass.

- **tol_aggregation** – Minimum increase in the objective function to enter a new aggregation pass.
- **n_aggregations** – Maximum number of aggregations. A negative value is interpreted as no limit.
- **shuffle_nodes** – Enables node shuffling before optimization.
- **random_state** – Random number generator or random seed. If None, numpy.random is used.

Variables

- **embedding_** (*array, shape = (n, n_components)*) – Embedding of the nodes.
- **embedding_row_** (*array, shape = (n_row, n_components)*) – Embedding of the rows, for bipartite graphs.
- **embedding_col_** (*array, shape = (n_col, n_components)*) – Embedding of the columns, for bipartite graphs.

Example

```
>>> from sknetwork.embedding import LouvainNE
>>> from sknetwork.data import karate_club
>>> louvain = LouvainNE(n_components=3)
>>> adjacency = karate_club()
>>> embedding = louvain.fit_transform(adjacency)
>>> embedding.shape
(34, 3)
```

References

Bhowmick, A. K., Meneni, K., Danisch, M., Guillaume, J. L., & Mitra, B. (2020, January). LouvainNE: Hierarchical Louvain Method for High Quality and Scalable Network Embedding. In Proceedings of the 13th International Conference on Web Search and Data Mining (pp. 43-51).

fit(*input_matrix: Union[scipy.sparse.csr.csr_matrix, numpy.ndarray], force_bipartite: bool = False*)
Embedding of graphs from a clustering obtained with Louvain.

Parameters

- **input_matrix** – Adjacency matrix or biadjacency matrix of the graph.
- **force_bipartite** – If `True`, force the input matrix to be considered as a biadjacency matrix even if square.

Returns **self**

Return type *LouvainNE*

fit_transform(**args*, ***kwargs*) → numpy.ndarray
Fit to data and return the embedding. Same parameters as the `fit` method.

Returns **embedding** – Embedding.

Return type np.ndarray

3.13.8 Force Atlas

class sknetwork.embedding.**ForceAtlas**(*n_components: int = 2*, *n_iter: int = 50*, *approx_radius: float = - 1*, *lin_log: bool = False*, *gravity_factor: float = 0.01*, *repulsive_factor: float = 0.1*, *tolerance: float = 0.1*, *speed: float = 0.1*, *speed_max: float = 10*)

Force Atlas layout for displaying graphs.

Parameters

- **n_components** (*int*) – Dimension of the graph layout.
- **n_iter** (*int*) – Number of iterations to update positions. If None, use the value of self.n_iter.
- **approx_radius** (*float*) – If a positive value is provided, only the nodes within this distance a given node are used to compute the repulsive force.
- **lin_log** (*bool*) – If True, use lin-log mode.
- **gravity_factor** (*float*) – Gravity force scaling constant.
- **repulsive_factor** (*float*) – Repulsive force scaling constant.
- **tolerance** (*float*) – Tolerance defined in the swinging constant.
- **speed** (*float*) – Speed constant.
- **speed_max** (*float*) – Constant used to impose constrain on speed.

Variables **embedding_** (*np.ndarray*) – Layout in given dimension.

Example

```
>>> from sknetwork.embedding.force_atlas import ForceAtlas
>>> from sknetwork.data import karate_club
>>> force_atlas = ForceAtlas()
>>> adjacency = karate_club()
>>> embedding = force_atlas.fit_transform(adjacency)
>>> embedding.shape
(34, 2)
```

References

Jacomy M., Venturini T., Heymann S., Bastian M. (2014). ForceAtlas2, a Continuous Graph Layout Algorithm for Handy Network Visualization Designed for the Gephi Software. Plos One.

fit(*adjacency: Union[scipy.sparse.csr.csr_matrix, numpy.ndarray]*, *pos_init: Optional[numpy.ndarray] = None*, *n_iter: Optional[int] = None*) → *sknetwork.embedding.force_atlas.ForceAtlas*

Compute layout.

Parameters

- **adjacency** – Adjacency matrix of the graph, treated as undirected.
- **pos_init** – Position to start with. Random if not provided.
- **n_iter** (*int*) – Number of iterations to update positions. If None, use the value of self.n_iter.

Returns self

Return type *ForceAtlas*

fit_transform(**args*, ***kwargs*) → numpy.ndarray
Fit to data and return the embedding. Same parameters as the `fit` method.

Returns **embedding** – Embedding.

Return type np.ndarray

3.13.9 Spring

class sknetwork.embedding.**Spring**(*n_components: int = 2*, *strength: Optional[float] = None*, *n_iter: int = 50*, *tol: float = 0.0001*, *approx_radius: float = - 1*, *position_init: str = 'random'*)
Spring layout for displaying small graphs.

Parameters

- **n_components** (*int*) – Dimension of the graph layout.
- **strength** (*float*) – Intensity of the force that moves the nodes.
- **n_iter** (*int*) – Number of iterations to update positions.
- **tol** (*float*) – Minimum relative change in positions to continue updating.
- **approx_radius** (*float*) – If a positive value is provided, only the nodes within this distance a given node are used to compute the repulsive force.
- **position_init** (*str*) – How to initialize the layout. If 'spectral', use Spectral embedding in dimension 2, otherwise, use random initialization.

Variables **embedding_** (*np.ndarray*) – Layout.

Example

```
>>> from sknetwork.embedding import Spring
>>> from sknetwork.data import karate_club
>>> spring = Spring()
>>> adjacency = karate_club()
>>> embedding = spring.fit_transform(adjacency)
>>> embedding.shape
(34, 2)
```

Notes

Simple implementation designed to display small graphs.

References

Fruchterman, T. M. J., Reingold, E. M. (1991). Graph Drawing by Force-Directed Placement. Software – Practice & Experience.

fit(*adjacency: Union[scipy.sparse.csr.csr_matrix, numpy.ndarray]*, *position_init: Optional[numpy.ndarray] = None*, *n_iter: Optional[int] = None*) → *sknetwork.embedding.spring.Spring*
Compute layout.

Parameters

- **adjacency** – Adjacency matrix of the graph, treated as undirected.
- **position_init** (*np.ndarray*) – Custom initial positions of the nodes. Shape must be (n, 2). If None, use the value of self.pos_init.
- **n_iter** (*int*) – Number of iterations to update positions. If None, use the value of self.n_iter.

Returns **self**

Return type *Spring*

fit_transform(**args*, ***kwargs*) → numpy.ndarray
Fit to data and return the embedding. Same parameters as the fit method.

Returns **embedding** – Embedding.

Return type np.ndarray

predict(*adjacency_vectors: Union[scipy.sparse.csr.csr_matrix, numpy.ndarray]*) → numpy.ndarray
Predict the embedding of new rows, defined by their adjacency vectors.

Parameters **adjacency_vectors** – Adjacency vectors of nodes. Array of shape (n_col,) (single vector) or (n_vectors, n_col)

Returns **embedding_vectors** – Embedding of the nodes.

Return type np.ndarray

3.13.10 Metrics

sknetwork.embedding.**cosine_modularity**(*adjacency*, *embedding: numpy.ndarray*, *embedding_col=None*, *resolution=1.0*, *weights='degree'*, *return_all: bool = False*)
Quality metric of an embedding x defined by:

$$Q = \sum_{ij} \left(\frac{A_{ij}}{w} - \gamma \frac{w_i^+ w_j^-}{w^2} \right) \left(\frac{1 + \cos(x_i, x_j)}{2} \right)$$

where

- w_i^+, w_i^- are the out-weight, in-weight of node i (for digraphs),
- $w = 1^T A 1$ is the total weight of the graph.

For bipartite graphs with column embedding y, the metric is

$$Q = \sum_{ij} \left(\frac{B_{ij}}{w} - \gamma \frac{w_{1,i} w_{2,j}}{w^2} \right) \left(\frac{1 + \cos(x_i, y_j)}{2} \right)$$

where

- $w_{1,i}, w_{2,j}$ are the weights of nodes i (row) and j (column),

- $w = 1^T B 1$ is the total weight of the graph.

Parameters

- **adjacency** – Adjacency matrix of the graph.
- **embedding** – Embedding of the nodes.
- **embedding_col** – Embedding of the columns (for bipartite graphs).
- **resolution** – Resolution parameter.
- **weights** (`'degree'` or `'uniform'`) – Weights of the nodes.
- **return_all** – If `True`, also return fit and diversity

Returns

- **modularity** (*float*)
- **fit** (*float, optional*)
- **diversity** (*float, optional*)

Example

```
>>> from sknetwork.embedding import cosine_modularity
>>> from sknetwork.data import karate_club
>>> graph = karate_club(metadata=True)
>>> adjacency = graph.adjacency
>>> embedding = graph.position
>>> np.round(cosine_modularity(adjacency, embedding), 2)
0.35
```

3.14 Link prediction

Link prediction algorithms.

The method `predict` assigns a scores to edges.

3.14.1 First order methods

class sknetwork.linkpred.**CommonNeighbors**

Link prediction by common neighbors:

$s(i, j) = |\Gamma_i \cap \Gamma_j|$.

Variables

- **indptr_** (*np.ndarray*) – Pointer index for neighbors.
- **indices_** (*np.ndarray*) – Concatenation of neighbors.

Examples

```
>>> from sknetwork.data import house
>>> adjacency = house()
>>> cn = CommonNeighbors()
>>> similarities = cn.fit_predict(adjacency, 0)
>>> similarities
array([2, 1, 1, 1, 1])
>>> similarities = cn.predict([0, 1])
>>> similarities
array([[2, 1, 1, 1, 1],
       [1, 3, 0, 2, 1]])
>>> similarities = cn.predict((0, 1))
>>> similarities
1
>>> similarities = cn.predict([(0, 1), (1, 2)])
>>> similarities
array([1, 0])
```

fit(*adjacency: Union[scipy.sparse.csr.csr_matrix, numpy.ndarray]*)
Fit algorithm to the data.

Parameters **adjacency** – Adjacency matrix of the graph

Returns **self**

Return type FirstOrder

fit_predict(*adjacency*, *query*)
Fit algorithm to data and compute scores for requested edges.

predict(*query: Union[int, Iterable, Tuple]*)
Compute similarity scores.

Parameters **query** (*int, list, array or Tuple*) –

- If int i, return the similarities s(i, j) for all j.
- If list or array integers, return s(i, j) for i in query, for all j as array.
- If tuple (i, j), return the similarity s(i, j).
- If list of tuples or array of shape (n_queries, 2), return s(i, j) for (i, j) in query as array.

Returns **predictions** – The prediction scores.

Return type int, float or array

class sknetwork.linkpred.**JaccardIndex**
Link prediction by Jaccard Index:

$$s(i,j) = \frac{|\Gamma_i \cap \Gamma_j|}{|\Gamma_i \cup \Gamma_j|}.$$

Variables

- **indptr_** (*np.ndarray*) – Pointer index for neighbors.
- **indices_** (*np.ndarray*) – Concatenation of neighbors.

Examples

```
>>> from sknetwork.data import house
>>> adjacency = house()
>>> jaccard = JaccardIndex()
>>> similarities = jaccard.fit_predict(adjacency, 0)
>>> similarities.round(2)
array([1.  , 0.25, 0.33, 0.33, 0.25])
>>> similarities = jaccard.predict([0, 1])
>>> similarities.round(2)
array([[1.  , 0.25, 0.33, 0.33, 0.25],
       [0.25, 1.  , 0.  , 0.67, 0.2 ]])
>>> similarities = jaccard.predict((0, 1))
>>> similarities.round(2)
0.25
>>> similarities = jaccard.predict([(0, 1), (1, 2)])
>>> similarities.round(2)
array([0.25, 0.  ])
```

References

Levandowsky, M., & Winter, D. (1971). Distance between sets. Nature, 234(5323), 34-35.

fit(*adjacency: Union[scipy.sparse.csr.csr_matrix, numpy.ndarray]*)
Fit algorithm to the data.

Parameters **adjacency** – Adjacency matrix of the graph

Returns **self**

Return type FirstOrder

fit_predict(*adjacency*, *query*)
Fit algorithm to data and compute scores for requested edges.

predict(*query: Union[int, Iterable, Tuple]*)
Compute similarity scores.

Parameters **query** (*int, list, array or Tuple*) –

- If int i, return the similarities s(i, j) for all j.
- If list or array integers, return s(i, j) for i in query, for all j as array.
- If tuple (i, j), return the similarity s(i, j).
- If list of tuples or array of shape (n_queries, 2), return s(i, j) for (i, j) in query as array.

Returns **predictions** – The prediction scores.

Return type int, float or array

class sknetwork.linkpred.**SaltonIndex**
Link prediction by Salton Index:

$$s(i,j) = \frac{|\Gamma_i \cap \Gamma_j|}{\sqrt{|\Gamma_i|.|\Gamma_j|}}.$$

Variables

- **indptr_** (*np.ndarray*) – Pointer index for neighbors.

- **indices_** (*np.ndarray*) – Concatenation of neighbors.

Examples

```
>>> from sknetwork.data import house
>>> adjacency = house()
>>> salton = SaltonIndex()
>>> similarities = salton.fit_predict(adjacency, 0)
>>> similarities.round(2)
array([1.  , 0.41, 0.5 , 0.5 , 0.41])
>>> similarities = salton.predict([0, 1])
>>> similarities.round(2)
array([[1.  , 0.41, 0.5 , 0.5 , 0.41],
       [0.41, 1.  , 0.  , 0.82, 0.33]])
>>> similarities = salton.predict((0, 1))
>>> similarities.round(2)
0.41
>>> similarities = salton.predict([(0, 1), (1, 2)])
>>> similarities.round(2)
array([0.41, 0.  ])
```

References

Martínez, V., Berzal, F., & Cubero, J. C. (2016). A survey of link prediction in complex networks. ACM computing surveys (CSUR), 49(4), 1-33.

fit(*adjacency: Union[scipy.sparse.csr.csr_matrix, numpy.ndarray]*)
Fit algorithm to the data.

Parameters **adjacency** – Adjacency matrix of the graph

Returns **self**

Return type FirstOrder

fit_predict(*adjacency*, *query*)
Fit algorithm to data and compute scores for requested edges.

predict(*query: Union[int, Iterable, Tuple]*)
Compute similarity scores.

Parameters **query** (*int, list, array or Tuple*) –

- If int i, return the similarities s(i, j) for all j.
- If list or array integers, return s(i, j) for i in query, for all j as array.
- If tuple (i, j), return the similarity s(i, j).
- If list of tuples or array of shape (n_queries, 2), return s(i, j) for (i, j) in query as array.

Returns **predictions** – The prediction scores.

Return type int, float or array

class sknetwork.linkpred.**SorensenIndex**
Link prediction by Salton Index:

$$s(i,j) = \frac{2|\Gamma_i \cap \Gamma_j|}{|\Gamma_i| + |\Gamma_j|}.$$

Variables

- **indptr_** (*np.ndarray*) – Pointer index for neighbors.
- **indices_** (*np.ndarray*) – Concatenation of neighbors.

Examples

```
>>> from sknetwork.data import house
>>> adjacency = house()
>>> sorensen = SorensenIndex()
>>> similarities = sorensen.fit_predict(adjacency, 0)
>>> similarities.round(2)
array([1. , 0.4, 0.5, 0.5, 0.4])
>>> similarities = sorensen.predict([0, 1])
>>> similarities.round(2)
array([[1.  , 0.4 , 0.5 , 0.5 , 0.4 ],
       [0.4 , 1.  , 0.  , 0.8 , 0.33]])
>>> similarities = sorensen.predict((0, 1))
>>> similarities.round(2)
0.4
>>> similarities = sorensen.predict([(0, 1), (1, 2)])
>>> similarities.round(2)
array([0.4, 0. ])
```

References

- Sørensen, T. J. (1948). A method of establishing groups of equal amplitude in plant sociology based on similarity of species content and its application to analyses of the vegetation on Danish commons. I kommission hos E. Munksgaard.
- Martínez, V., Berzal, F., & Cubero, J. C. (2016). A survey of link prediction in complex networks. ACM computing surveys (CSUR), 49(4), 1-33.

fit(*adjacency: Union[scipy.sparse.csr.csr_matrix, numpy.ndarray]*)
Fit algorithm to the data.

Parameters **adjacency** – Adjacency matrix of the graph

Returns **self**

Return type FirstOrder

fit_predict(*adjacency*, *query*)
Fit algorithm to data and compute scores for requested edges.

predict(*query: Union[int, Iterable, Tuple]*)
Compute similarity scores.

Parameters **query** (*int, list, array or Tuple*) –

- If int i, return the similarities s(i, j) for all j.
- If list or array integers, return s(i, j) for i in query, for all j as array.

- If tuple (i, j), return the similarity s(i, j).
- If list of tuples or array of shape (n_queries, 2), return s(i, j) for (i, j) in query as array.

Returns **predictions** – The prediction scores.

Return type int, float or array

class sknetwork.linkpred.**HubPromotedIndex**

Link prediction by Hub Promoted Index:

$$s(i,j) = \frac{2|\Gamma_i \cap \Gamma_j|}{min(|\Gamma_i|, |\Gamma_j|)}.$$

Variables

- **indptr_** (*np.ndarray*) – Pointer index for neighbors.
- **indices_** (*np.ndarray*) – Concatenation of neighbors.

Examples

```
>>> from sknetwork.data import house
>>> adjacency = house()
>>> hpi = HubPromotedIndex()
>>> similarities = hpi.fit_predict(adjacency, 0)
>>> similarities.round(2)
array([1. , 0.5, 0.5, 0.5, 0.5])
>>> similarities = hpi.predict([0, 1])
>>> similarities.round(2)
array([[1.  , 0.5 , 0.5 , 0.5 , 0.5 ],
       [0.5 , 1.  , 0.  , 1.  , 0.33]])
>>> similarities = hpi.predict((0, 1))
>>> similarities.round(2)
0.5
>>> similarities = hpi.predict([(0, 1), (1, 2)])
>>> similarities.round(2)
array([0.5, 0. ])
```

References

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

fit(*adjacency: Union[scipy.sparse.csr.csr_matrix, numpy.ndarray]*)

Fit algorithm to the data.

Parameters **adjacency** – Adjacency matrix of the graph

Returns **self**

Return type FirstOrder

fit_predict(*adjacency*, *query*)

Fit algorithm to data and compute scores for requested edges.

predict(*query: Union[int, Iterable, Tuple]*)

Compute similarity scores.

Parameters **query** (*int, list, array or Tuple*) –

- If int i, return the similarities s(i, j) for all j.
- If list or array integers, return s(i, j) for i in query, for all j as array.
- If tuple (i, j), return the similarity s(i, j).
- If list of tuples or array of shape (n_queries, 2), return s(i, j) for (i, j) in query as array.

Returns **predictions** – The prediction scores.

Return type int, float or array

class sknetwork.linkpred.**HubDepressedIndex**

Link prediction by Hub Depressed Index:

$$s(i,j) = \frac{2|\Gamma_i \cap \Gamma_j|}{max(|\Gamma_i|, |\Gamma_j|)}.$$

Variables

- **indptr_** (*np.ndarray*) – Pointer index for neighbors.
- **indices_** (*np.ndarray*) – Concatenation of neighbors.

Examples

```
>>> from sknetwork.data import house
>>> adjacency = house()
>>> hdi = HubDepressedIndex()
>>> similarities = hdi.fit_predict(adjacency, 0)
>>> similarities.round(2)
array([1.  , 0.33, 0.5 , 0.5 , 0.33])
>>> similarities = hdi.predict([0, 1])
>>> similarities.round(2)
array([[1.  , 0.33, 0.5 , 0.5 , 0.33],
       [0.33, 1.  , 0.  , 0.67, 0.33]])
>>> similarities = hdi.predict((0, 1))
>>> similarities.round(2)
0.33
>>> similarities = hdi.predict([(0, 1), (1, 2)])
>>> similarities.round(2)
array([0.33, 0.  ])
```

References

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

fit(*adjacency: Union[scipy.sparse.csr.csr_matrix, numpy.ndarray]*)

Fit algorithm to the data.

Parameters **adjacency** – Adjacency matrix of the graph

Returns **self**

Return type FirstOrder

fit_predict(*adjacency*, *query*)
Fit algorithm to data and compute scores for requested edges.

predict(*query: Union[int, Iterable, Tuple]*)
Compute similarity scores.

Parameters **query** (*int, list, array or Tuple*) –

- If int i, return the similarities s(i, j) for all j.
- If list or array integers, return s(i, j) for i in query, for all j as array.
- If tuple (i, j), return the similarity s(i, j).
- If list of tuples or array of shape (n_queries, 2), return s(i, j) for (i, j) in query as array.

Returns **predictions** – The prediction scores.

Return type int, float or array

class sknetwork.linkpred.**AdamicAdar**
Link prediction by Adamic-Adar index:

$$s(i,j) = \sum_{z \in \Gamma_i \cap \Gamma_j} \frac{1}{\log |\Gamma_z|}.$$

Variables

- **indptr_** (*np.ndarray*) – Pointer index for neighbors.
- **indices_** (*np.ndarray*) – Concatenation of neighbors.

Examples

```
>>> from sknetwork.data import house
>>> adjacency = house()
>>> aa = AdamicAdar()
>>> similarities = aa.fit_predict(adjacency, 0)
>>> similarities.round(2)
array([1.82, 0.91, 0.91, 0.91, 0.91])
>>> similarities = aa.predict([0, 1])
>>> similarities.round(2)
array([[1.82, 0.91, 0.91, 0.91, 0.91],
       [0.91, 3.8 , 0.  , 2.35, 1.44]])
>>> similarities = aa.predict((0, 1))
>>> similarities.round(2)
0.91
>>> similarities = aa.predict([(0, 1), (1, 2)])
>>> similarities.round(2)
array([0.91, 0.  ])
```

References

Adamic, L. A., & Adar, E. (2003). Friends and neighbors on the web. Social networks, 25(3), 211-230.

fit(*adjacency: Union[scipy.sparse.csr.csr_matrix, numpy.ndarray]*)
Fit algorithm to the data.

Parameters **adjacency** – Adjacency matrix of the graph

Returns **self**

Return type FirstOrder

fit_predict(*adjacency*, *query*)
Fit algorithm to data and compute scores for requested edges.

predict(*query: Union[int, Iterable, Tuple]*)
Compute similarity scores.

Parameters **query** (*int, list, array or Tuple*) –

- If int i, return the similarities s(i, j) for all j.
- If list or array integers, return s(i, j) for i in query, for all j as array.
- If tuple (i, j), return the similarity s(i, j).
- If list of tuples or array of shape (n_queries, 2), return s(i, j) for (i, j) in query as array.

Returns **predictions** – The prediction scores.

Return type int, float or array

class sknetwork.linkpred.**ResourceAllocation**
Link prediction by Resource Allocation index:

$$s(i,j) = \sum_{z \in \Gamma_i \cap \Gamma_j} \frac{1}{|\Gamma_z|}.$$

Variables

- **indptr_** (*np.ndarray*) – Pointer index for neighbors.
- **indices_** (*np.ndarray*) – Concatenation of neighbors.

Examples

```
>>> from sknetwork.data import house
>>> adjacency = house()
>>> ra = ResourceAllocation()
>>> similarities = ra.fit_predict(adjacency, 0)
>>> similarities.round(2)
array([0.67, 0.33, 0.33, 0.33, 0.33])
>>> similarities = ra.predict([0, 1])
>>> similarities.round(2)
array([[0.67, 0.33, 0.33, 0.33, 0.33],
       [0.33, 1.33, 0.  , 0.83, 0.5 ]])
>>> similarities = ra.predict((0, 1))
>>> similarities.round(2)
0.33
>>> similarities = ra.predict([(0, 1), (1, 2)])
```

(continues on next page)

(continued from previous page)

```
>>> similarities.round(2)
array([0.33, 0.  ])
```

References

Zhou, T., Lü, L., & Zhang, Y. C. (2009). Predicting missing links via local information. The European Physical Journal B, 71(4), 623-630.

fit(*adjacency: Union[scipy.sparse.csr.csr_matrix, numpy.ndarray]*)
Fit algorithm to the data.

Parameters **adjacency** – Adjacency matrix of the graph

Returns **self**

Return type FirstOrder

fit_predict(*adjacency*, *query*)
Fit algorithm to data and compute scores for requested edges.

predict(*query: Union[int, Iterable, Tuple]*)
Compute similarity scores.

Parameters **query** (*int, list, array or Tuple*) –

- If int i, return the similarities s(i, j) for all j.
- If list or array integers, return s(i, j) for i in query, for all j as array.
- If tuple (i, j), return the similarity s(i, j).
- If list of tuples or array of shape (n_queries, 2), return s(i, j) for (i, j) in query as array.

Returns **predictions** – The prediction scores.

Return type int, float or array

class sknetwork.linkpred.**PreferentialAttachment**
Link prediction by Preferential Attachment index:

$s(i, j) = |\Gamma_i||\Gamma_j|$.

Variables

- **indptr_** (*np.ndarray*) – Pointer index for neighbors.
- **indices_** (*np.ndarray*) – Concatenation of neighbors.

Examples

```
>>> from sknetwork.data import house
>>> adjacency = house()
>>> pa = PreferentialAttachment()
>>> similarities = pa.fit_predict(adjacency, 0)
>>> similarities
array([4, 6, 4, 4, 6], dtype=int32)
>>> similarities = pa.predict([0, 1])
>>> similarities
array([[4, 6, 4, 4, 6],
```

(continues on next page)

(continued from previous page)

```
       [6, 9, 6, 6, 9]], dtype=int32)
>>> similarities = pa.predict((0, 1))
>>> similarities
6
>>> similarities = pa.predict([(0, 1), (1, 2)])
>>> similarities
array([6, 6], dtype=int32)
```

References

Albert, R., Barabási, L. (2002). Statistical mechanics of complex networks Reviews of Modern Physics.

fit(*adjacency: Union[scipy.sparse.csr.csr_matrix, numpy.ndarray]*)
Fit algorithm to the data.

Parameters **adjacency** – Adjacency matrix of the graph

Returns **self**

Return type FirstOrder

fit_predict(*adjacency*, *query*)
Fit algorithm to data and compute scores for requested edges.

predict(*query: Union[int, Iterable, Tuple]*)
Compute similarity scores.

Parameters **query** (*int, list, array or Tuple*) –

- If int i, return the similarities s(i, j) for all j.
- If list or array integers, return s(i, j) for i in query, for all j as array.
- If tuple (i, j), return the similarity s(i, j).
- If list of tuples or array of shape (n_queries, 2), return s(i, j) for (i, j) in query as array.

Returns **predictions** – The prediction scores.

Return type int, float or array

3.14.2 Post-processing

skneteork.linkpred.**is_edge**(*adjacency: scipy.sparse.csr.csr_matrix*, *query: Union[int, Iterable, Tuple]*) → Union[bool, numpy.ndarray]
Given a query, return whether each edge is actually in the adjacency.

Parameters

- **adjacency** – Adjacency matrix of the graph.
- **query** (*int, Iterable or Tuple*) –
  - If int i, queries (i, j) for all j.
  - If Iterable of integers, return queries (i, j) for i in query, for all j.
  - If tuple (i, j), queries (i, j).
  - If list of tuples or array of shape (n_queries, 2), queries (i, j) in for each line in query.

Returns **y_true** – For each element in the query, returns `True` if the edge exists in the adjacency and `False` otherwise.

Return type Union[bool, np.ndarray]

Examples

```
>>> from sknetwork.data import house
>>> adjacency = house()
>>> is_edge(adjacency, 0)
array([False,  True, False, False,  True])
>>> is_edge(adjacency, [0, 1])
array([[False,  True, False, False,  True],
       [ True, False,  True, False,  True]])
>>> is_edge(adjacency, (0, 1))
True
>>> is_edge(adjacency, [(0, 1), (0, 2)])
array([ True, False])
```

`sknetwork.linkpred.`**whitened_sigmoid**(*scores: numpy.ndarray*)

Map the entries of a score array to probabilities through

$$\frac{1}{1+\exp(-(x-\mu)/\sigma)},$$

where μ and σ are respectively the mean and standard deviation of x.

Parameters **scores** (*np.ndarray*) – The input array

Returns **probas** – Array with entries between 0 and 1.

Return type np.ndarray

Examples

```
>>> probas = whitened_sigmoid(np.array([1, 5, 0.25]))
>>> probas.round(2)
array([0.37, 0.8 , 0.29])
>>> probas = whitened_sigmoid(np.array([2, 2, 2]))
>>> probas
array([1, 1, 1])
```

3.15 Linear algebra

Tools of linear algebra.

3.15.1 Polynomials

class sknetwork.linalg.**Polynome**(**args*, ***kwargs*)

Polynome of an adjacency matrix as a linear operator

$P(A) = \alpha_k A^k + ... + \alpha_1 A + \alpha_0$.

Parameters

- **adjacency** – Adjacency matrix of the graph
- **coeffs** (*np.ndarray*) – Coefficients of the polynome by increasing order of power.

Examples

```
>>> from scipy import sparse
>>> from sknetwork.linalg import Polynome
>>> adjacency = sparse.eye(2, format='csr')
>>> polynome = Polynome(adjacency, np.arange(3))
>>> x = np.ones(2)
>>> polynome.dot(x)
array([3., 3.])
>>> polynome.T.dot(x)
array([3., 3.])
```

Notes

The polynome is evaluated using the Ruffini-Horner method.

property H

Hermitian adjoint.

Returns the Hermitian adjoint of self, aka the Hermitian conjugate or Hermitian transpose. For a complex matrix, the Hermitian adjoint is equal to the conjugate transpose.

Can be abbreviated self.H instead of self.adjoint().

Returns **A_H** – Hermitian adjoint of self.

Return type LinearOperator

property T

Transpose this linear operator.

Returns a LinearOperator that represents the transpose of this one. Can be abbreviated self.T instead of self.transpose().

adjoint()

Hermitian adjoint.

Returns the Hermitian adjoint of self, aka the Hermitian conjugate or Hermitian transpose. For a complex matrix, the Hermitian adjoint is equal to the conjugate transpose.

Can be abbreviated self.H instead of self.adjoint().

Returns **A_H** – Hermitian adjoint of self.

Return type LinearOperator

dot(*x*)
Matrix-matrix or matrix-vector multiplication.

Parameters **x** (*array_like*) – 1-d or 2-d array, representing a vector or matrix.

Returns **Ax** – 1-d or 2-d array (depending on the shape of x) that represents the result of applying this linear operator on x.

Return type array

matmat(*X*)
Matrix-matrix multiplication.

Performs the operation y=A*X where A is an MxN linear operator and X dense N*K matrix or ndarray.

Parameters **X** (*{matrix, ndarray}*) – An array with shape (N,K).

Returns **Y** – A matrix or ndarray with shape (M,K) depending on the type of the X argument.

Return type {matrix, ndarray}

Notes

This matmat wraps any user-specified matmat routine or overridden _matmat method to ensure that y has the correct type.

matvec(*x*)
Matrix-vector multiplication.

Performs the operation y=A*x where A is an MxN linear operator and x is a column vector or 1-d array.

Parameters **x** (*{matrix, ndarray}*) – An array with shape (N,) or (N,1).

Returns **y** – A matrix or ndarray with shape (M,) or (M,1) depending on the type and shape of the x argument.

Return type {matrix, ndarray}

Notes

This matvec wraps the user-specified matvec routine or overridden _matvec method to ensure that y has the correct shape and type.

rmatmat(*X*)
Adjoint matrix-matrix multiplication.

Performs the operation y = A^H * x where A is an MxN linear operator and x is a column vector or 1-d array, or 2-d array. The default implementation defers to the adjoint.

Parameters **X** (*{matrix, ndarray}*) – A matrix or 2D array.

Returns **Y** – A matrix or 2D array depending on the type of the input.

Return type {matrix, ndarray}

Notes

This rmatmat wraps the user-specified rmatmat routine.

rmatvec(*x*)
Adjoint matrix-vector multiplication.

Performs the operation y = A^H * x where A is an MxN linear operator and x is a column vector or 1-d array.

Parameters **x** (*{matrix, ndarray}*) – An array with shape (M,) or (M,1).

Returns **y** – A matrix or ndarray with shape (N,) or (N,1) depending on the type and shape of the x argument.

Return type {matrix, ndarray}

Notes

This rmatvec wraps the user-specified rmatvec routine or overridden _rmatvec method to ensure that y has the correct shape and type.

transpose()
Transpose this linear operator.

Returns a LinearOperator that represents the transpose of this one. Can be abbreviated self.T instead of self.transpose().

3.15.2 Sparse + Low Rank

class sknetwork.linalg.**SparseLR**(**args*, ***kwargs*)
Class for matrices with "sparse + low rank" structure. Example:

$A + xy^T$

Parameters

- **sparse_mat** (*scipy.spmatrix*) – Sparse component. Is converted to csr format automatically.
- **low_rank_tuples** (*list*) – Single tuple of arrays of list of tuples, representing the low rank components [(x1, y1), (x2, y2),...]. Each low rank component is of the form xy^T.

Examples

```
>>> from scipy import sparse
>>> from sknetwork.linalg import SparseLR
>>> adjacency = sparse.eye(2, format='csr')
>>> slr = SparseLR(adjacency, (np.ones(2), np.ones(2)))
>>> x = np.ones(2)
>>> slr.dot(x)
array([3., 3.])
>>> slr.sum(axis=0)
array([3., 3.])
>>> slr.sum(axis=1)
array([3., 3.])
```

(continues on next page)

(continued from previous page)

```
>>> slr.sum()
6.0
```

References

De Lara (2019). The Sparse + Low Rank trick for Matrix Factorization-Based Graph Algorithms. Proceedings of the 15th International Workshop on Mining and Learning with Graphs (MLG).

property H
Hermitian adjoint.

Returns the Hermitian adjoint of self, aka the Hermitian conjugate or Hermitian transpose. For a complex matrix, the Hermitian adjoint is equal to the conjugate transpose.

Can be abbreviated self.H instead of self.adjoint().

Returns **A_H** – Hermitian adjoint of self.

Return type LinearOperator

property T
Transpose this linear operator.

Returns a LinearOperator that represents the transpose of this one. Can be abbreviated self.T instead of self.transpose().

adjoint()
Hermitian adjoint.

Returns the Hermitian adjoint of self, aka the Hermitian conjugate or Hermitian transpose. For a complex matrix, the Hermitian adjoint is equal to the conjugate transpose.

Can be abbreviated self.H instead of self.adjoint().

Returns **A_H** – Hermitian adjoint of self.

Return type LinearOperator

astype(*dtype: Union[str, numpy.dtype]*)
Change dtype of the object.

dot(*x*)
Matrix-matrix or matrix-vector multiplication.

Parameters **x** (*array_like*) – 1-d or 2-d array, representing a vector or matrix.

Returns **Ax** – 1-d or 2-d array (depending on the shape of x) that represents the result of applying this linear operator on x.

Return type array

left_sparse_dot(*matrix: scipy.sparse.csr.csr_matrix*)
Left dot product with a sparse matrix.

matmat(*X*)
Matrix-matrix multiplication.

Performs the operation y=A*X where A is an MxN linear operator and X dense N*K matrix or ndarray.

Parameters **X** (*{matrix, ndarray}*) – An array with shape (N,K).

Returns **Y** – A matrix or ndarray with shape (M,K) depending on the type of the X argument.

Return type {matrix, ndarray}

Notes

This matmat wraps any user-specified matmat routine or overridden _matmat method to ensure that y has the correct type.

matvec(*x*)
Matrix-vector multiplication.

Performs the operation y=A*x where A is an MxN linear operator and x is a column vector or 1-d array.

Parameters **x** ({*matrix, ndarray*}) – An array with shape (N,) or (N,1).

Returns **y** – A matrix or ndarray with shape (M,) or (M,1) depending on the type and shape of the x argument.

Return type {matrix, ndarray}

Notes

This matvec wraps the user-specified matvec routine or overridden _matvec method to ensure that y has the correct shape and type.

right_sparse_dot(*matrix: scipy.sparse.csr.csr_matrix*)
Right dot product with a sparse matrix.

rmatmat(*X*)
Adjoint matrix-matrix multiplication.

Performs the operation y = A^H * x where A is an MxN linear operator and x is a column vector or 1-d array, or 2-d array. The default implementation defers to the adjoint.

Parameters **X** ({*matrix, ndarray*}) – A matrix or 2D array.

Returns **Y** – A matrix or 2D array depending on the type of the input.

Return type {matrix, ndarray}

Notes

This rmatmat wraps the user-specified rmatmat routine.

rmatvec(*x*)
Adjoint matrix-vector multiplication.

Performs the operation y = A^H * x where A is an MxN linear operator and x is a column vector or 1-d array.

Parameters **x** ({*matrix, ndarray*}) – An array with shape (M,) or (M,1).

Returns **y** – A matrix or ndarray with shape (N,) or (N,1) depending on the type and shape of the x argument.

Return type {matrix, ndarray}

Notes

This rmatvec wraps the user-specified rmatvec routine or overridden _rmatvec method to ensure that y has the correct shape and type.

sum(*axis=None*)
Row-wise, column-wise or total sum of operator's coefficients.

Parameters **axis** – If 0, return column-wise sum. If 1, return row-wise sum. Otherwise, return total sum.

transpose()
Transpose this linear operator.

Returns a LinearOperator that represents the transpose of this one. Can be abbreviated self.T instead of self.transpose().

3.15.3 Operators

class sknetwork.linalg.**Regularizer**(**args*, ***kwargs*)
Regularized matrix as a Scipy LinearOperator.

Defined by $A + \alpha \frac{11^T}{n}$ where A is the input matrix and α the regularization factor.

Parameters

- **input_matrix** – Input matrix.
- **regularization** (*float*) – Regularization factor. Default value = 1.

Examples

```
>>> from sknetwork.data import house
>>> adjacency = house()
>>> regularizer = Regularizer(adjacency)
>>> regularizer.dot(np.ones(5))
array([3., 4., 3., 3., 4.])
```

property H
Hermitian adjoint.

Returns the Hermitian adjoint of self, aka the Hermitian conjugate or Hermitian transpose. For a complex matrix, the Hermitian adjoint is equal to the conjugate transpose.

Can be abbreviated self.H instead of self.adjoint().

Returns **A_H** – Hermitian adjoint of self.

Return type LinearOperator

property T
Transpose this linear operator.

Returns a LinearOperator that represents the transpose of this one. Can be abbreviated self.T instead of self.transpose().

adjoint()
Hermitian adjoint.

Returns the Hermitian adjoint of self, aka the Hermitian conjugate or Hermitian transpose. For a complex matrix, the Hermitian adjoint is equal to the conjugate transpose.

Can be abbreviated self.H instead of self.adjoint().

Returns **A_H** – Hermitian adjoint of self.

Return type LinearOperator

astype(*dtype: Union[str, numpy.dtype]*)
Change dtype of the object.

dot(*x*)
Matrix-matrix or matrix-vector multiplication.

Parameters **x** (*array_like*) – 1-d or 2-d array, representing a vector or matrix.

Returns **Ax** – 1-d or 2-d array (depending on the shape of x) that represents the result of applying this linear operator on x.

Return type array

left_sparse_dot(*matrix: scipy.sparse.csr.csr_matrix*)
Left dot product with a sparse matrix.

matmat(*X*)
Matrix-matrix multiplication.

Performs the operation y=A*X where A is an MxN linear operator and X dense N*K matrix or ndarray.

Parameters **X** (*{matrix, ndarray}*) – An array with shape (N,K).

Returns **Y** – A matrix or ndarray with shape (M,K) depending on the type of the X argument.

Return type {matrix, ndarray}

Notes

This matmat wraps any user-specified matmat routine or overridden _matmat method to ensure that y has the correct type.

matvec(*x*)
Matrix-vector multiplication.

Performs the operation y=A*x where A is an MxN linear operator and x is a column vector or 1-d array.

Parameters **x** (*{matrix, ndarray}*) – An array with shape (N,) or (N,1).

Returns **y** – A matrix or ndarray with shape (M,) or (M,1) depending on the type and shape of the x argument.

Return type {matrix, ndarray}

Notes

This matvec wraps the user-specified matvec routine or overridden _matvec method to ensure that y has the correct shape and type.

right_sparse_dot(*matrix: scipy.sparse.csr.csr_matrix*)
Right dot product with a sparse matrix.

rmatmat(*X*)
Adjoint matrix-matrix multiplication.

Performs the operation y = A^H * x where A is an MxN linear operator and x is a column vector or 1-d array, or 2-d array. The default implementation defers to the adjoint.

Parameters **X** (*{matrix, ndarray}*) – A matrix or 2D array.

Returns **Y** – A matrix or 2D array depending on the type of the input.

Return type {matrix, ndarray}

Notes

This rmatmat wraps the user-specified rmatmat routine.

rmatvec(*x*)
Adjoint matrix-vector multiplication.

Performs the operation y = A^H * x where A is an MxN linear operator and x is a column vector or 1-d array.

Parameters **x** (*{matrix, ndarray}*) – An array with shape (M,) or (M,1).

Returns **y** – A matrix or ndarray with shape (N,) or (N,1) depending on the type and shape of the x argument.

Return type {matrix, ndarray}

Notes

This rmatvec wraps the user-specified rmatvec routine or overridden _rmatvec method to ensure that y has the correct shape and type.

sum(*axis=None*)
Row-wise, column-wise or total sum of operator's coefficients.

Parameters **axis** – If 0, return column-wise sum. If 1, return row-wise sum. Otherwise, return total sum.

transpose()
Transpose this linear operator.

Returns a LinearOperator that represents the transpose of this one. Can be abbreviated self.T instead of self.transpose().

class sknetwork.linalg.**Normalizer**(**args*, ***kwargs*)
Normalized matrix as a Scipy LinearOperator.

Defined by $D^{-1}A$ where A is the regularized adjacency matrix and D the corresponding diagonal matrix of degrees (sums over rows).

Parameters

- **adjacency** – *Adjacency* matrix of the graph.
- **regularization** (*float*) – Regularization factor. Default value = 0.

Examples

```
>>> from sknetwork.data import house
>>> adjacency = house()
>>> normalizer = Normalizer(adjacency)
>>> normalizer.dot(np.ones(5))
array([1., 1., 1., 1., 1.])
```

property H
Hermitian adjoint.

Returns the Hermitian adjoint of self, aka the Hermitian conjugate or Hermitian transpose. For a complex matrix, the Hermitian adjoint is equal to the conjugate transpose.

Can be abbreviated self.H instead of self.adjoint().

Returns **A_H** – Hermitian adjoint of self.

Return type LinearOperator

property T
Transpose this linear operator.

Returns a LinearOperator that represents the transpose of this one. Can be abbreviated self.T instead of self.transpose().

adjoint()
Hermitian adjoint.

Returns the Hermitian adjoint of self, aka the Hermitian conjugate or Hermitian transpose. For a complex matrix, the Hermitian adjoint is equal to the conjugate transpose.

Can be abbreviated self.H instead of self.adjoint().

Returns **A_H** – Hermitian adjoint of self.

Return type LinearOperator

dot(*x*)
Matrix-matrix or matrix-vector multiplication.

Parameters **x** (*array_like*) – 1-d or 2-d array, representing a vector or matrix.

Returns **Ax** – 1-d or 2-d array (depending on the shape of x) that represents the result of applying this linear operator on x.

Return type array

matmat(*X*)
Matrix-matrix multiplication.

Performs the operation y=A*X where A is an MxN linear operator and X dense N*K matrix or ndarray.

Parameters **X** (*{matrix, ndarray}*) – An array with shape (N,K).

Returns **Y** – A matrix or ndarray with shape (M,K) depending on the type of the X argument.

Return type {matrix, ndarray}

Notes

This matmat wraps any user-specified matmat routine or overridden _matmat method to ensure that y has the correct type.

matvec(*x*)

Matrix-vector multiplication.

Performs the operation y=A*x where A is an MxN linear operator and x is a column vector or 1-d array.

Parameters **x** ({*matrix, ndarray*}) – An array with shape (N,) or (N,1).

Returns **y** – A matrix or ndarray with shape (M,) or (M,1) depending on the type and shape of the x argument.

Return type {matrix, ndarray}

Notes

This matvec wraps the user-specified matvec routine or overridden _matvec method to ensure that y has the correct shape and type.

rmatmat(*X*)

Adjoint matrix-matrix multiplication.

Performs the operation y = A^H * x where A is an MxN linear operator and x is a column vector or 1-d array, or 2-d array. The default implementation defers to the adjoint.

Parameters **X** ({*matrix, ndarray*}) – A matrix or 2D array.

Returns **Y** – A matrix or 2D array depending on the type of the input.

Return type {matrix, ndarray}

Notes

This rmatmat wraps the user-specified rmatmat routine.

rmatvec(*x*)

Adjoint matrix-vector multiplication.

Performs the operation y = A^H * x where A is an MxN linear operator and x is a column vector or 1-d array.

Parameters **x** ({*matrix, ndarray*}) – An array with shape (M,) or (M,1).

Returns **y** – A matrix or ndarray with shape (N,) or (N,1) depending on the type and shape of the x argument.

Return type {matrix, ndarray}

Notes

This rmatvec wraps the user-specified rmatvec routine or overridden _rmatvec method to ensure that y has the correct shape and type.

transpose()
Transpose this linear operator.

Returns a LinearOperator that represents the transpose of this one. Can be abbreviated self.T instead of self.transpose().

class sknetwork.linalg.**Laplacian**(**args*, ***kwargs*)
Laplacian matrix as a Scipy LinearOperator.

Defined by $L = D - A$ where A is the regularized adjacency matrix and D the corresponding diagonal matrix of degrees.

If normalized, defined by $L = I - D^{-1/2}AD^{-1/2}$.

Parameters

- **adjacency** – *Adjacency* matrix of the graph.
- **regularization** (*float*) – Regularization factor. Default value = 0.
- **normalized_laplacian** (*bool*) – If `True`, use normalized Laplacian. Default value = `False`.

Examples

```
>>> from sknetwork.data import house
>>> adjacency = house()
>>> laplacian = Laplacian(adjacency)
>>> laplacian.dot(np.ones(5))
array([0., 0., 0., 0., 0.])
```

property H
Hermitian adjoint.

Returns the Hermitian adjoint of self, aka the Hermitian conjugate or Hermitian transpose. For a complex matrix, the Hermitian adjoint is equal to the conjugate transpose.

Can be abbreviated self.H instead of self.adjoint().

Returns **A_H** – Hermitian adjoint of self.

Return type LinearOperator

property T
Transpose this linear operator.

Returns a LinearOperator that represents the transpose of this one. Can be abbreviated self.T instead of self.transpose().

adjoint()
Hermitian adjoint.

Returns the Hermitian adjoint of self, aka the Hermitian conjugate or Hermitian transpose. For a complex matrix, the Hermitian adjoint is equal to the conjugate transpose.

Can be abbreviated self.H instead of self.adjoint().

Returns **A_H** – Hermitian adjoint of self.

Return type LinearOperator

astype(*dtype: Union[str, numpy.dtype]*)
Change dtype of the object.

dot(*x*)
Matrix-matrix or matrix-vector multiplication.

Parameters **x** (*array_like*) – 1-d or 2-d array, representing a vector or matrix.

Returns **Ax** – 1-d or 2-d array (depending on the shape of x) that represents the result of applying this linear operator on x.

Return type array

matmat(*X*)
Matrix-matrix multiplication.

Performs the operation y=A*X where A is an MxN linear operator and X dense N*K matrix or ndarray.

Parameters **X** (*{matrix, ndarray}*) – An array with shape (N,K).

Returns **Y** – A matrix or ndarray with shape (M,K) depending on the type of the X argument.

Return type {matrix, ndarray}

Notes

This matmat wraps any user-specified matmat routine or overridden _matmat method to ensure that y has the correct type.

matvec(*x*)
Matrix-vector multiplication.

Performs the operation y=A*x where A is an MxN linear operator and x is a column vector or 1-d array.

Parameters **x** (*{matrix, ndarray}*) – An array with shape (N,) or (N,1).

Returns **y** – A matrix or ndarray with shape (M,) or (M,1) depending on the type and shape of the x argument.

Return type {matrix, ndarray}

Notes

This matvec wraps the user-specified matvec routine or overridden _matvec method to ensure that y has the correct shape and type.

rmatmat(*X*)
Adjoint matrix-matrix multiplication.

Performs the operation y = A^H * x where A is an MxN linear operator and x is a column vector or 1-d array, or 2-d array. The default implementation defers to the adjoint.

Parameters **X** (*{matrix, ndarray}*) – A matrix or 2D array.

Returns **Y** – A matrix or 2D array depending on the type of the input.

Return type {matrix, ndarray}

Notes

This rmatmat wraps the user-specified rmatmat routine.

rmatvec(*x*)

Adjoint matrix-vector multiplication.

Performs the operation y = A^H * x where A is an MxN linear operator and x is a column vector or 1-d array.

Parameters **x** (*{matrix, ndarray}*) – An array with shape (M,) or (M,1).

Returns **y** – A matrix or ndarray with shape (N,) or (N,1) depending on the type and shape of the x argument.

Return type {matrix, ndarray}

Notes

This rmatvec wraps the user-specified rmatvec routine or overridden _rmatvec method to ensure that y has the correct shape and type.

transpose()

Transpose this linear operator.

Returns a LinearOperator that represents the transpose of this one. Can be abbreviated self.T instead of self.transpose().

class sknetwork.linalg.**CoNeighbor**(**args*, ***kwargs*)

Co-neighborhood adjacency as a LinearOperator.

$\tilde{A} = AF^{-1}A^T$, or $\tilde{B} = BF^{-1}B^T$.

where F is a weight matrix.

Parameters

- **adjacency** – Adjacency or biadjacency of the input graph.
- **normalized** – If `True`, F is the diagonal in-degree matrix $F = \text{diag}(A^T 1)$. Otherwise, F is the identity matrix.

Examples

```
>>> from sknetwork.data import star_wars
>>> biadjacency = star_wars(metadata=False)
>>> d_out = biadjacency.dot(np.ones(3))
>>> coneigh = CoNeighbor(biadjacency)
>>> np.allclose(d_out, coneigh.dot(np.ones(4)))
True
```

property H

Hermitian adjoint.

Returns the Hermitian adjoint of self, aka the Hermitian conjugate or Hermitian transpose. For a complex matrix, the Hermitian adjoint is equal to the conjugate transpose.

Can be abbreviated self.H instead of self.adjoint().

Returns **A_H** – Hermitian adjoint of self.

Return type LinearOperator

`property T`
Transpose this linear operator.

Returns a LinearOperator that represents the transpose of this one. Can be abbreviated self.T instead of self.transpose().

`adjoint()`
Hermitian adjoint.

Returns the Hermitian adjoint of self, aka the Hermitian conjugate or Hermitian transpose. For a complex matrix, the Hermitian adjoint is equal to the conjugate transpose.

Can be abbreviated self.H instead of self.adjoint().

Returns **A_H** – Hermitian adjoint of self.

Return type LinearOperator

`astype(dtype: Union[str, numpy.dtype])`
Change dtype of the object.

`dot(x)`
Matrix-matrix or matrix-vector multiplication.

Parameters **x** (*array_like*) – 1-d or 2-d array, representing a vector or matrix.

Returns **Ax** – 1-d or 2-d array (depending on the shape of x) that represents the result of applying this linear operator on x.

Return type array

`left_sparse_dot(matrix: scipy.sparse.csr.csr_matrix)`
Left dot product with a sparse matrix

`matmat(X)`
Matrix-matrix multiplication.

Performs the operation y=A*X where A is an MxN linear operator and X dense N*K matrix or ndarray.

Parameters **X** (*{matrix, ndarray}*) – An array with shape (N,K).

Returns **Y** – A matrix or ndarray with shape (M,K) depending on the type of the X argument.

Return type {matrix, ndarray}

Notes

This matmat wraps any user-specified matmat routine or overridden _matmat method to ensure that y has the correct type.

`matvec(x)`
Matrix-vector multiplication.

Performs the operation y=A*x where A is an MxN linear operator and x is a column vector or 1-d array.

Parameters **x** (*{matrix, ndarray}*) – An array with shape (N,) or (N,1).

Returns **y** – A matrix or ndarray with shape (M,) or (M,1) depending on the type and shape of the x argument.

Return type {matrix, ndarray}

Notes

This matvec wraps the user-specified matvec routine or overridden _matvec method to ensure that y has the correct shape and type.

right_sparse_dot(*matrix: scipy.sparse.csr.csr_matrix*)
Right dot product with a sparse matrix

rmatmat(*X*)
Adjoint matrix-matrix multiplication.

Performs the operation y = A^H * x where A is an MxN linear operator and x is a column vector or 1-d array, or 2-d array. The default implementation defers to the adjoint.

Parameters **X** (*{matrix, ndarray}*) – A matrix or 2D array.

Returns **Y** – A matrix or 2D array depending on the type of the input.

Return type {matrix, ndarray}

Notes

This rmatmat wraps the user-specified rmatmat routine.

rmatvec(*x*)
Adjoint matrix-vector multiplication.

Performs the operation y = A^H * x where A is an MxN linear operator and x is a column vector or 1-d array.

Parameters **x** (*{matrix, ndarray}*) – An array with shape (M,) or (M,1).

Returns **y** – A matrix or ndarray with shape (N,) or (N,1) depending on the type and shape of the x argument.

Return type {matrix, ndarray}

Notes

This rmatvec wraps the user-specified rmatvec routine or overridden _rmatvec method to ensure that y has the correct shape and type.

transpose()
Transpose this linear operator.

Returns a LinearOperator that represents the transpose of this one. Can be abbreviated self.T instead of self.transpose().

3.15.4 Solvers

class sknetwork.linalg.**LanczosEig**(*which='LM'*, *n_iter: Optional[int] = None*, *tol: float = 0.0*)
Eigenvalue solver using Lanczos method.

Parameters

- **which** (*str*) – Which eigenvectors and eigenvalues to find:
  - `'LM'` : Largest (in modulus) eigenvalues.
  - `'SM'` : Smallest (in modulus) eigenvalues.

- 'LA' : Largest (algebraic) eigenvalues.
- 'SA' : Smallest (algebraic) eigenvalues.

- **n_iter** (*int*) – Maximum number of Arnoldi update iterations allowed. Default = 10 * nb of rows.
- **tol** (*float*) – Relative accuracy for eigenvalues (stopping criterion). Default = 0 (machine precision).

Variables

- **eigenvectors_** (*np.ndarray*) – Two-dimensional array, each column is an eigenvector of the input.
- **eigenvalues_** (*np.ndarray*) – Eigenvalues associated to each eigenvector.

See also:

scipy.sparse.linalg.eigsh

fit(*matrix: Union[scipy.sparse.csr.csr_matrix, scipy.sparse.linalg.interface.LinearOperator]*, *n_components: int = 2*)
Perform spectral decomposition on symmetric input matrix.

Parameters

- **matrix** (*sparse.csr_matrix or linear operator*) – Matrix to decompose.
- **n_components** (*int*) – Number of eigenvectors to compute

Returns self

Return type EigSolver

class sknetwork.linalg.**LanczosSVD**(*n_iter: Optional[int] = None*, *tol: float = 0.0*)
SVD solver using Lanczos method on AA^T or A^TA.

Parameters

- **n_iter** (*int*) – Maximum number of Arnoldi update iterations allowed. Default = 10 * nb or rows or columns.
- **tol** (*float*) – Relative accuracy for eigenvalues (stopping criterion). Default = 0 (machine precision).

Variables

- **singular_vectors_left_** (*np.ndarray*) – Two-dimensional array, each column is a left singular vector of the input.
- **singular_vectors_right_** (*np.ndarray*) – Two-dimensional array, each column is a right singular vector of the input.
- **singular_values_** (*np.ndarray*) – Singular values.

See also:

scipy.sparse.linalg.svds

fit(*matrix: Union[scipy.sparse.csr.csr_matrix, scipy.sparse.linalg.interface.LinearOperator]*, *n_components: int*, *init_vector: Optional[numpy.ndarray] = None*)
Perform singular value decomposition on input matrix.

Parameters

- **matrix** – Matrix to decompose.

- **n_components** (*int*) – Number of singular values to compute
- **init_vector** (*np.ndarray*) – Starting vector for iteration. Default = random.

Returns self

Return type SVDSolver

sknetwork.linalg.ppr_solver.**get_pagerank**(*adjacency: Union[scipy.sparse.csr.csr_matrix, scipy.sparse.linalg.interface.LinearOperator], seeds: numpy.ndarray, damping_factor: float, n_iter: int, tol: float = 1e-06, solver: str = 'piteration'*) → numpy.ndarray

Solve the Pagerank problem. Formally,

$x = \alpha Px + (1 - \alpha)y$,

where $P = (D^{-1}A)^T$ is the transition matrix and y is the personalization probability vector.

Parameters

- **adjacency** (*sparse.csr_matrix*) – Adjacency matrix of the graph.
- **seeds** (*np.ndarray*) – Personalization array. Must be a valid probability vector.
- **damping_factor** (*float*) – Probability to continue the random walk.
- **n_iter** (*int*) – Number of iterations for some of the solvers such as `'piteration'` or `'diteration'`.
- **tol** (*float*) – Tolerance for the convergence of some solvers such as `'bicgstab'` or `'lanczos'` or `'push'`.
- **solver** (str) – Which solver to use: `'piteration'`, `'diteration'`, `'bicgstab'`, `'lanczos'`, `'RH'`, `'push'`.

Returns **pagerank** – Probability vector.

Return type np.ndarray

Examples

```
>>> from sknetwork.data import house
>>> adjacency = house()
>>> seeds = np.array([1, 0, 0, 0, 0])
>>> scores = get_pagerank(adjacency, seeds, damping_factor=0.85, n_iter=10)
>>> np.round(scores, 2)
array([0.29, 0.24, 0.12, 0.12, 0.24])
```

References

- Hong, D. (2012). Optimized on-line computation of pagerank algorithm. arXiv preprint arXiv:1202.6158.
- Van der Vorst, H. A. (1992). Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. SIAM Journal on scientific and Statistical Computing, 13(2), 631-644.
- Lanczos, C. (1950). An iteration method for the solution of the eigenvalue problem of linear differential and integral operators. Los Angeles, CA: United States Governm. Press Office.
- Whang, J. , Lenharth, A. , Dhillon, I. , & Pingali, K. . (2015). *Scalable Data-Driven PageRank: Algorithms, System Issues, and Lessons Learned*. 9233, 438-450. <https://www.cs.utexas.edu/users/inderjit/public_papers/scalable_pagerank_europar15.pdf>

3.15.5 Miscellaneous

sknetwork.linalg.**diag_pinv**(*weights: numpy.ndarray*) → scipy.sparse.csr.csr_matrix
Compute $W^+ = \text{diag}(w)^+$, the pseudo inverse of the diagonal matrix with diagonal the weights w.

Parameters **weights** – The weights to invert.

Returns W^+

Return type sparse.csr_matrix

sknetwork.linalg.**normalize**(*matrix: Union[scipy.sparse.csr.csr_matrix, numpy.ndarray, scipy.sparse.linalg.interface.LinearOperator]*, *p=1*)
Normalize rows of a matrix. Null rows remain null.

Parameters

- **matrix** – Input matrix.
- **p** – Order of the norm

Returns **normalized matrix**

Return type same as input

3.16 Utils

Various tools for graph analysis.

3.16.1 Convert graphs

sknetwork.utils.**bipartite2directed**(*biadjacency: Union[scipy.sparse.csr.csr_matrix, sknetwork.linalg.sparse_lowrank.SparseLR]*) → Union[scipy.sparse.csr.csr_matrix, *sknetwork.linalg.sparse_lowrank.SparseLR*]
Adjacency matrix of the directed graph associated with a bipartite graph (with edges from one part to the other).

The returned adjacency matrix is:

$$A = \begin{bmatrix} 0 & B \\ 0 & 0 \end{bmatrix}$$

where B is the biadjacency matrix.

Parameters **biadjacency** – Biadjacency matrix of the graph.

Returns Adjacency matrix (same format as input).

Return type adjacency

sknetwork.utils.**bipartite2undirected**(*biadjacency: Union[scipy.sparse.csr.csr_matrix, sknetwork.linalg.sparse_lowrank.SparseLR]*) → Union[scipy.sparse.csr.csr_matrix, *sknetwork.linalg.sparse_lowrank.SparseLR*]
Adjacency matrix of a bigraph defined by its biadjacency matrix.

The returned adjacency matrix is:

$$A = \begin{bmatrix} 0 & B \\ B^T & 0 \end{bmatrix}$$

where B is the biadjacency matrix of the bipartite graph.

Parameters **biadjacency** – Biadjacency matrix of the graph.

Returns Adjacency matrix (same format as input).

Return type adjacency

skknetwork.utils.**directed2undirected**(*adjacency: Union[scipy.sparse.csr.csr_matrix, sknetwork.linalg.sparse_lowrank.SparseLR], weighted: bool = True*) → Union[scipy.sparse.csr.csr_matrix, *sknetwork.linalg.sparse_lowrank.SparseLR*]

Adjacency matrix of the undirected graph associated with some directed graph.

The new adjacency matrix becomes either:

$A + A^T$ (default)

or

$\max(A, A^T)$

If the initial adjacency matrix A is binary, bidirectional edges have weight 2 (first method, default) or 1 (second method).

Parameters

- **adjacency** – Adjacency matrix.
- **weighted** – If `True`, return the sum of the weights in both directions of each edge.

Returns New adjacency matrix (same format as input).

Return type new_adjacency

3.16.2 Get neighbors

skknetwork.utils.**get_neighbors**(*input_matrix: scipy.sparse.csr.csr_matrix, node: int, transpose: bool = False*) → numpy.ndarray

Get the neighbors of a node.

Parameters

- **input_matrix** (*sparse.csr_matrix*) – Adjacency or biadjacency matrix.
- **node** (*int*) – Target node.
- **transpose** – If `True`, transpose the input matrix.

Returns **neighbors** – Array of neighbors of the target node (successors for directed graphs; set `transpose=True` for predecessors).

Return type np.ndarray

Example

```
>>> from sknetwork.data import house
>>> adjacency = house()
>>> get_neighbors(adjacency, node=0)
array([1, 4], dtype=int32)
```

3.16.3 Clustering

sknetwork.utils.**membership_matrix**(*labels: numpy.ndarray*, *dtype=<class 'bool'>*, *n_labels: Optional[int] = None*) → scipy.sparse.csr.csr_matrix

Build a n x k matrix of the label assignments, with k the number of labels. Negative labels are ignored.

Parameters

- **labels** – Label of each node.
- **dtype** – Type of the entries. Boolean by default.
- **n_labels** (*int*) – Number of labels.

Returns **membership** – Binary matrix of label assignments.

Return type sparse.csr_matrix

Notes

The inverse operation is simply `labels = membership.indices`.

class sknetwork.utils.**KMeansDense**(*n_clusters: int = 8*, *init: str = '++'*, *n_init: int = 10*, *tol: float = 0.0001*)

Standard KMeansDense clustering based on SciPy function `kmeans2`.

Parameters

- **n_clusters** – Number of desired clusters.
- **init** – Method for initialization. Available methods are 'random', 'points', '++' and 'matrix': * 'random': generate k centroids from a Gaussian with mean and variance estimated from the data. * 'points': choose k observations (rows) at random from data for the initial centroids. * '++': choose k observations accordingly to the kmeans++ method (careful seeding) * 'matrix': interpret the k parameter as a k by M (or length k array for one-dimensional data) array of initial centroids.
- **n_init** – Number of iterations of the k-means algorithm to run.
- **tol** – Relative tolerance with regards to inertia to declare convergence.

Variables

- **labels_** – Label of each sample.
- **cluster_centers_** – A 'k' by 'N' array of centroids found at the last iteration of k-means.

References

- MacQueen, J. (1967, June). Some methods for classification and analysis of multivariate observations. In Proceedings of the fifth Berkeley symposium on mathematical statistics and probability (Vol. 1, No. 14, pp. 281-297).
- Arthur, D., & Vassilvitskii, S. (2007, January). k-means++: The advantages of careful seeding. In Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms (pp. 1027-1035). Society for Industrial and Applied Mathematics.

fit(*x: numpy.ndarray*) → *sknetwork.utils.kmeans.KMeansDense*
Fit algorithm to the data.

Parameters **x** – Data to cluster.

Returns **self**

Return type *KMeansDense*

fit_transform(*x: numpy.ndarray*) → numpy.ndarray
Fit algorithm to the data and return the labels.

Parameters **x** – Data to cluster.

Returns **labels**

Return type np.ndarray

class sknetwork.utils.**WardDense**
Hierarchical clustering by the Ward method based on SciPy.

Variables **dendrogram_** (*np.ndarray (n - 1, 4)*) – Dendrogram.

References

- Ward, J. H., Jr. (1963). Hierarchical grouping to optimize an objective function. Journal of the American Statistical Association, 58, 236–244.
- Murtagh, F., & Contreras, P. (2012). Algorithms for hierarchical clustering: an overview. Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery, 2(1), 86-97.

fit(*x: numpy.ndarray*) → *sknetwork.utils.ward.WardDense*
Apply algorithm to a dense matrix.

Parameters **x** – Data to cluster.

Returns **self**

Return type *WardDense*

fit_transform(*x: numpy.ndarray*) → numpy.ndarray
Apply algorithm to a dense matrix and return the dendrogram.

Parameters **x** – Data to cluster.

Returns **dendrogram**

Return type np.ndarray

3.16.4 Nearest-neighbors

class sknetwork.utils.**KNNDense**(*n_neighbors: int = 5*, *undirected: bool = False*, *leaf_size: int = 16*, *p=2*, *eps: float = 0.01*, *n_jobs=1*)

Extract adjacency from vector data through k-nearest-neighbor search with KD-Tree.

Parameters

- **n_neighbors** – Number of neighbors for each sample in the transformed sparse graph.
- **undirected** – As the nearest neighbor relationship is not symmetric, the graph is directed by default. Setting this parameter to `True` forces the algorithm to return undirected graphs.
- **leaf_size** – Leaf size passed to KDTree. This can affect the speed of the construction and query, as well as the memory required to store the tree.
- **p** – Which Minkowski p-norm to use. 1 is the sum-of-absolute-values "Manhattan" distance, 2 is the usual Euclidean distance infinity is the maximum-coordinate-difference distance. A finite large p may cause a ValueError if overflow can occur.
- **eps** – Return approximate nearest neighbors; the k-th returned value is guaranteed to be no further than (1+tol_nn) times the distance to the real k-th nearest neighbor.
- **n_jobs** – Number of jobs to schedule for parallel processing. If -1 is given all processors are used.

Variables **adjacency_** – Adjacency matrix of the graph.

References

Maneewongvatana, S., & Mount, D. M. (1999, December). It's okay to be skinny, if your friends are fat. In Center for Geometric Computing 4th Annual Workshop on Computational Geometry (Vol. 2, pp. 1-8).

fit(*x: numpy.ndarray*) → *sknetwork.utils.knn.KNNDense*

Fit algorithm to the data.

Parameters **x** – Data to transform into adjacency.

Returns **self**

Return type *KNNDense*

fit_transform(*x: numpy.ndarray*) → scipy.sparse.csr.csr_matrix

Fit algorithm to the data and return the computed adjacency.

Parameters **x** (*np.ndarray*) – Input data.

Returns **adjacency**

Return type sparse.csr_matrix

make_undirected()

Modifies the adjacency to match desired constrains.

class sknetwork.utils.**CNNDense**(*n_neighbors: int = 1*, *undirected: bool = False*)

Extract adjacency from vector data through component-wise k-nearest-neighbor search. KNN is applied independently on each column of the input matrix.

Parameters

- **n_neighbors** – Number of neighbors per dimension.
- **undirected** – As the nearest neighbor relationship is not symmetric, the graph is directed by default. Setting this parameter to `True` forces the algorithm to return undirected graphs.

Variables **adjacency_** – Adjacency matrix of the graph.

fit(*x: numpy.ndarray*) → *sknetwork.utils.knn.CNNDense*
Fit algorithm to the data.

Parameters **x** – Data to transform into adjacency.

Returns **self**

Return type *CNNDense*

fit_transform(*x: numpy.ndarray*) → scipy.sparse.csr.csr_matrix
Fit algorithm to the data and return the computed adjacency.

Parameters **x** (*np.ndarray*) – Input data.

Returns **adjacency**

Return type sparse.csr_matrix

make_undirected()
Modifies the adjacency to match desired constrains.

3.16.5 Co-Neighborhood

sknetwork.utils.**co_neighbor_graph**(*adjacency: Union[scipy.sparse.csr.csr_matrix, numpy.ndarray]*, *normalized: bool = True*, *method='knn'*, *n_neighbors: int = 5*, *n_components: int = 8*) → scipy.sparse.csr.csr_matrix
Compute the co-neighborhood adjacency.

- Graphs
- Digraphs
- Bigraphs

$\tilde{A} = AF^{-1}A^T$,

where F is a weight matrix.

Parameters

- **adjacency** – Adjacency of the input graph.
- **normalized** – If `True`, F is the diagonal in-degree matrix $F = \text{diag}(A^T 1)$. Otherwise, F is the identity matrix.
- **method** – Either `'exact'` or `'knn'`. If 'exact' the output is computed with matrix multiplication. However, the density can be much higher than in the input graph and this can trigger Memory errors. If `'knn'`, the co-neighborhood is approximated through KNNDense-search in an appropriate spectral embedding space.
- **n_neighbors** – Number of neighbors for the KNNDense search. Only useful if `method='knn'`.
- **n_components** – Dimension of the embedding space. Only useful if `method='knn'`.

Returns **adjacency** – Adjacency of the co-neighborhood.

Return type sparse.csr_matrix

3.16.6 Projection

sknetwork.utils.**projection_simplex**(*x: Union[numpy.ndarray, scipy.sparse.csr.csr_matrix]*, *scale: float = 1.0*)

Project each line of the input onto the Euclidean simplex i.e. solve

$\min_w ||w - x_i||_2^2$ s.t. $\sum w_j = z, w_j \geq 0$.

Parameters

- **x** – Data to project. Either one or two dimensional. Can be sparse or dense.
- **scale** (*float*) – Scale of the simplex i.e. sums of the projected coefficients.

Returns **projection** – Array with the same type and shape as the input.

Return type np.ndarray or sparse.csr_matrix

Example

```
>>> X = np.array([[2, 2], [-0.75, 0.25]])
>>> projection_simplex(X)
array([[0.5, 0.5],
       [0. , 1. ]])
>>> X_csr = sparse.csr_matrix(X)
>>> X_proj = projection_simplex(X_csr)
>>> X_proj.nnz
3
>>> X_proj.toarray()
array([[0.5, 0.5],
       [0. , 1. ]])
```

References

Duchi, J., Shalev-Shwartz, S., Singer, Y., & Chandra, T. (2008, July). Efficient projections onto the l 1-ball for learning in high dimensions. In Proceedings of the 25th international conference on Machine learning (pp. 272-279). ACM.

sknetwork.utils.**projection_simplex_array**(*array: numpy.ndarray*, *scale: float = 1*) → numpy.ndarray

Project each line of the input onto the Euclidean simplex i.e. solve

$\min_w ||w - x_i||_2^2$ s.t. $\sum w_j = z, w_j \geq 0$.

Parameters

- **array** (*np.ndarray*) – Data to project. Either one or two dimensional.
- **scale** (*float*) – Scale of the simplex i.e. sums of the projected coefficients.

Returns **projection** – Array with the same shape as the input.

Return type np.ndarray

Example

```
>>> X = np.array([[2, 2], [-0.75, 0.25]])
>>> projection_simplex_array(X)
array([[0.5, 0.5],
       [0. , 1. ]])
```

sknetwork.utils.**projection_simplex_csr**(*matrix: scipy.sparse.csr.csr_matrix*, *scale: float = 1*)

Project each line of the input onto the Euclidean simplex i.e. solve

$\min_w ||w - x_i||_2^2$ s.t. $\sum w_j = z, w_j \geq 0$.

Parameters

- **matrix** (*sparse.csr_matrix*) – Matrix whose rows must be projected.
- **scale** (*float*) – Scale of the simplex i.e. sums of the projected coefficients.

Returns **projection** – Matrix with the same shape as the input.

Return type sparse.csr_matrix

Examples

```
>>> X = sparse.csr_matrix(np.array([[2, 2], [-0.75, 0.25]]))
>>> X_proj = projection_simplex_csr(X)
>>> X_proj.nnz
3
>>> X_proj.toarray()
array([[0.5, 0.5],
       [0. , 1. ]])
```

3.17 Visualization

Visualization tools.

3.17.1 Graphs

skknetwork.visualization.graphs.**svg_graph**(*adjacency: Optional[scipy.sparse.csr.csr_matrix] = None*, *position: Optional[numpy.ndarray] = None*, *names: Optional[numpy.ndarray] = None*, *labels: Optional[Iterable] = None*, *name_position: str = 'right'*, *scores: Optional[Iterable] = None*, *membership: Optional[scipy.sparse.csr.csr_matrix] = None*, *seeds: Optional[Union[list, dict]] = None*, *width: Optional[float] = 400*, *height: Optional[float] = 300*, *margin: float = 20*, *margin_text: float = 3*, *scale: float = 1*, *node_order: Optional[numpy.ndarray] = None*, *node_size: float = 7*, *node_size_min: float = 1*, *node_size_max: float = 20*, *display_node_weight: bool = False*, *node_weights: Optional[numpy.ndarray] = None*, *node_width: float = 1*, *node_width_max: float = 3*, *node_color: str = 'gray'*, *display_edges: bool = True*, *edge_labels: Optional[list] = None*, *edge_width: float = 1*, *edge_width_min: float = 0.5*, *edge_width_max: float = 20*, *display_edge_weight: bool = True*, *edge_color: Optional[str] = None*, *label_colors: Optional[Iterable] = None*, *font_size: int = 12*, *directed: bool = False*, *filename: Optional[str] = None*) → str

Return SVG image of a graph.

Parameters

- **adjacency** – Adjacency matrix of the graph.
- **position** – Positions of the nodes.
- **names** – Names of the nodes.
- **labels** – Labels of the nodes (negative values mean no label).
- **name_position** – Position of the names (left, right, above, below)
- **scores** – Scores of the nodes (measure of importance).
- **membership** – Membership of the nodes (label distribution).
- **seeds** – Nodes to be highlighted (if dict, only keys are considered).
- **width** – Width of the image.
- **height** – Height of the image.
- **margin** – Margin of the image.
- **margin_text** – Margin between node and text.
- **scale** – Multiplicative factor on the dimensions of the image.
- **node_order** – Order in which nodes are displayed.
- **node_size** – Size of nodes.
- **node_size_min** – Minimum size of a node.
- **node_size_max** – Maximum size of a node.
- **node_width** – Width of node circle.
- **node_width_max** – Maximum width of node circle.

- **node_color** – Default color of nodes (svg color).
- **display_node_weight** – If `True`, display node weights through node size.
- **node_weights** – Node weights (used only if **display_node_weight** is `True`).
- **display_edges** – If `True`, display edges.
- **edge_labels** – Labels of the edges, as a list of tuples (source, destination, label)
- **edge_width** – Width of edges.
- **edge_width_min** – Minimum width of edges.
- **edge_width_max** – Maximum width of edges.
- **display_edge_weight** – If `True`, display edge weights through edge widths.
- **edge_color** – Default color of edges (svg color).
- **label_colors** – Colors of the labels (svg colors).
- **font_size** – Font size.
- **directed** – If `True`, considers the graph as directed.
- **filename** – Filename for saving image (optional).

Returns **image** – SVG image.

Return type str

Example

```
>>> from sknetwork.data import karate_club
>>> graph = karate_club(True)
>>> adjacency = graph.adjacency
>>> position = graph.position
>>> from sknetwork.visualization import svg_graph
>>> image = svg_graph(adjacency, position)
>>> image[1:4]
'svg'
```

sknetwork.visualization.graphs.**svg_digraph**(*adjacency: Optional[scipy.sparse.csr.csr_matrix] = None*, *position: Optional[numpy.ndarray] = None*, *names: Optional[numpy.ndarray] = None*, *labels: Optional[Iterable] = None*, *name_position: str = 'right'*, *scores: Optional[Iterable] = None*, *membership: Optional[scipy.sparse.csr.csr_matrix] = None*, *seeds: Optional[Union[list, dict]] = None*, *width: Optional[float] = 400*, *height: Optional[float] = 300*, *margin: float = 20*, *margin_text: float = 10*, *scale: float = 1*, *node_order: Optional[numpy.ndarray] = None*, *node_size: float = 7*, *node_size_min: float = 1*, *node_size_max: float = 20*, *display_node_weight: bool = False*, *node_weights: Optional[numpy.ndarray] = None*, *node_width: float = 1*, *node_width_max: float = 3*, *node_color: str = 'gray'*, *display_edges: bool = True*, *edge_labels: Optional[list] = None*, *edge_width: float = 1*, *edge_width_min: float = 0.5*, *edge_width_max: float = 5*, *display_edge_weight: bool = True*, *edge_color: Optional[str] = None*, *label_colors: Optional[Iterable] = None*, *font_size: int = 12*, *filename: Optional[str] = None*) → str

Return SVG image of a digraph.

Parameters

- **adjacency** – Adjacency matrix of the graph.
- **position** – Positions of the nodes.
- **names** – Names of the nodes.
- **labels** – Labels of the nodes (negative values mean no label).
- **name_position** – Position of the names (left, right, above, below)
- **scores** – Scores of the nodes (measure of importance).
- **membership** – Membership of the nodes (label distribution).
- **seeds** – Nodes to be highlighted (if dict, only keys are considered).
- **width** – Width of the image.
- **height** – Height of the image.
- **margin** – Margin of the image.
- **margin_text** – Margin between node and text.
- **scale** – Multiplicative factor on the dimensions of the image.
- **node_order** – Order in which nodes are displayed.
- **node_size** – Size of nodes.
- **node_size_min** – Minimum size of a node.
- **node_size_max** – Maximum size of a node.
- **display_node_weight** – If `True`, display node in-weights through node size.
- **node_weights** – Node weights (used only if **display_node_weight** is `True`).
- **node_width** – Width of node circle.
- **node_width_max** – Maximum width of node circle.

- **node_color** – Default color of nodes (svg color).
- **display_edges** – If `True`, display edges.
- **edge_labels** – Labels of the edges, as a list of tuples (source, destination, label)
- **edge_width** – Width of edges.
- **edge_width_min** – Minimum width of edges.
- **edge_width_max** – Maximum width of edges.
- **display_edge_weight** – If `True`, display edge weights through edge widths.
- **edge_color** – Default color of edges (svg color).
- **label_colors** – Colors of the labels (svg color).
- **font_size** – Font size.
- **filename** – Filename for saving image (optional).

Returns **image** – SVG image.

Return type str

Example

```
>>> from sknetwork.data import painters
>>> graph = painters(True)
>>> adjacency = graph.adjacency
>>> position = graph.position
>>> from sknetwork.visualization import svg_digraph
>>> image = svg_graph(adjacency, position)
>>> image[1:4]
'svg'
```

sknetwork.visualization.graphs.**svg_bigraph**(*biadjacency: scipy.sparse.csr.csr_matrix*, *names_row: Optional[numpy.ndarray] = None*, *names_col: Optional[numpy.ndarray] = None*, *labels_row: Optional[Union[numpy.ndarray, dict]] = None*, *labels_col: Optional[Union[numpy.ndarray, dict]] = None*, *scores_row: Optional[Union[numpy.ndarray, dict]] = None*, *scores_col: Optional[Union[numpy.ndarray, dict]] = None*, *membership_row: Optional[scipy.sparse.csr.csr_matrix] = None*, *membership_col: Optional[scipy.sparse.csr.csr_matrix] = None*, *seeds_row: Optional[Union[list, dict]] = None*, *seeds_col: Optional[Union[list, dict]] = None*, *position_row: Optional[numpy.ndarray] = None*, *position_col: Optional[numpy.ndarray] = None*, *reorder: bool = True*, *width: Optional[float] = 400*, *height: Optional[float] = 300*, *margin: float = 20*, *margin_text: float = 3*, *scale: float = 1*, *node_size: float = 7*, *node_size_min: float = 1*, *node_size_max: float = 20*, *display_node_weight: bool = False*, *node_weights_row: Optional[numpy.ndarray] = None*, *node_weights_col: Optional[numpy.ndarray] = None*, *node_width: float = 1*, *node_width_max: float = 3*, *color_row: str = 'gray'*, *color_col: str = 'gray'*, *label_colors: Optional[Iterable] = None*, *display_edges: bool = True*, *edge_labels: Optional[list] = None*, *edge_width: float = 1*, *edge_width_min: float = 0.5*, *edge_width_max: float = 10*, *edge_color: str = 'black'*, *display_edge_weight: bool = True*, *font_size: int = 12*, *filename: Optional[str] = None*) → str

Return SVG image of a bigraph.

Parameters

- **biadjacency** – Biadjacency matrix of the graph.
- **names_row** – Names of the rows.
- **names_col** – Names of the columns.
- **labels_row** – Labels of the rows (negative values mean no label).
- **labels_col** – Labels of the columns (negative values mean no label).
- **scores_row** – Scores of the rows (measure of importance).
- **scores_col** – Scores of the columns (measure of importance).
- **membership_row** – Membership of the rows (label distribution).
- **membership_col** – Membership of the columns (label distribution).
- **seeds_row** – Rows to be highlighted (if dict, only keys are considered).
- **seeds_col** – Columns to be highlighted (if dict, only keys are considered).
- **position_row** – Positions of the rows.
- **position_col** – Positions of the columns.
- **reorder** – Use clustering to order nodes.
- **width** – Width of the image.
- **height** – Height of the image.

- **margin** – Margin of the image.
- **margin_text** – Margin between node and text.
- **scale** – Multiplicative factor on the dimensions of the image.
- **node_size** – Size of nodes.
- **node_size_min** – Minimum size of nodes.
- **node_size_max** – Maximum size of nodes.
- **display_node_weight** – If `True`, display node weights through node size.
- **node_weights_row** – Weights of rows (used only if **display_node_weight** is `True`).
- **node_weights_col** – Weights of columns (used only if **display_node_weight** is `True`).
- **node_width** – Width of node circle.
- **node_width_max** – Maximum width of node circle.
- **color_row** – Default color of rows (svg color).
- **color_col** – Default color of cols (svg color).
- **label_colors** – Colors of the labels (svg color).
- **display_edges** – If `True`, display edges.
- **edge_labels** – Labels of the edges, as a list of tuples (source, destination, label)
- **edge_width** – Width of edges.
- **edge_width_min** – Minimum width of edges.
- **edge_width_max** – Maximum width of edges.
- **display_edge_weight** – If `True`, display edge weights through edge widths.
- **edge_color** – Default color of edges (svg color).
- **font_size** – Font size.
- **filename** – Filename for saving image (optional).

Returns **image** – SVG image.

Return type str

Example

```
>>> from sknetwork.data import movie_actor
>>> biadjacency = movie_actor()
>>> from sknetwork.visualization import svg_bigraph
>>> image = svg_bigraph(biadjacency)
>>> image[1:4]
'svg'
```

3.17.2 Dendrograms

sknetwork.visualization.dendrograms.**svg_dendrogram**(*dendrogram: numpy.ndarray*, *names: Optional[numpy.ndarray] = None*, *rotate: bool = False*, *width: float = 400*, *height: float = 300*, *margin: float = 10*, *margin_text: float = 5*, *scale: float = 1*, *line_width: float = 2*, *n_clusters: int = 2*, *color: str = 'black'*, *colors: Optional[Iterable] = None*, *font_size: int = 12*, *reorder: bool = False*, *rotate_names: bool = True*, *filename: Optional[str] = None*)

Return SVG image of a dendrogram.

Parameters

- **dendrogram** – Dendrogram to display.
- **names** – Names of leaves.
- **rotate** – If `True`, rotate the tree so that the root is on the left.
- **width** – Width of the image (margins excluded).
- **height** – Height of the image (margins excluded).
- **margin** – Margin.
- **margin_text** – Margin between leaves and their names, if any.
- **scale** – Scaling factor.
- **line_width** – Line width.
- **n_clusters** – Number of coloured clusters to display.
- **color** – Default SVG color for the dendrogram.
- **colors** – SVG colors of the clusters of the dendrogram (optional).
- **font_size** – Font size.
- **reorder** – If `True`, reorder leaves so that left subtree has more leaves than right subtree.
- **rotate_names** – If `True`, rotate names of leaves (only valid if **rotate** is `False`).
- **filename** – Filename for saving image (optional).

Example

```
>>> dendrogram = np.array([[0, 1, 1, 2], [2, 3, 2, 3]])
>>> from sknetwork.visualization import svg_dendrogram
>>> image = svg_dendrogram(dendrogram)
>>> image[1:4]
'svg'
```

3.18 Data

3.18.1 Load your data

In scikit-network, a graph is represented by its adjacency matrix (or biadjacency matrix for a bipartite graph) in the Compressed Sparse Row format of SciPy.

In this tutorial, we present a few methods to instantiate a graph in this format.

[1]:
```
from IPython.display import SVG

import numpy as np
from scipy import sparse
import pandas as pd

from sknetwork.data import from_edge_list, from_adjacency_list, from_graphml, from_csv
from sknetwork.visualization import svg_graph, svg_digraph, svg_bigraph
```

From a NumPy array

For small graphs, you can instantiate the adjacency matrix as a dense NumPy array and convert it into a sparse matrix in CSR format.

[2]:
```
adjacency = np.array([[0, 1, 1, 0], [1, 0, 1, 1], [1, 1, 0, 0], [0, 1, 0, 0]])
adjacency = sparse.csr_matrix(adjacency)

image = svg_graph(adjacency)
SVG(image)
```

[2]:

From an edge list

Another natural way to build a graph is from a list of edges.

[3]:
```
edge_list = [(0, 1), (1, 2), (2, 3), (3, 0), (0, 2)]
adjacency = from_edge_list(edge_list)

image = svg_graph(adjacency)
SVG(image)
```

[3]:

By default, the graph is undirected, but you can easily make it directed.

[4]:
```
adjacency = from_edge_list(edge_list, directed=True)

image = svg_digraph(adjacency)
SVG(image)
```

[4]:

You might also want to add weights to your edges. Just use triplets instead of pairs!

[5]:
```
edge_list = [(0, 1, 1), (1, 2, 0.5), (2, 3, 1), (3, 0, 0.5), (0, 2, 2)]
adjacency = from_edge_list(edge_list)

image = svg_graph(adjacency)
SVG(image)
```

[5]:

You can instantiate a bipartite graph as well.

[6]:
```
edge_list = [(0, 0), (1, 0), (1, 1), (2, 1)]
biadjacency = from_edge_list(edge_list, bipartite=True)

image = svg_bigraph(biadjacency)
SVG(image)
```

[6]:

If nodes are not indexed, you get an object of type Bunch with graph attributes (node names).

[7]:
```
edge_list = [("Alice", "Bob"), ("Bob", "Carey"), ("Alice", "David"), ("Carey", "David"),␣
↪("Bob", "David")]
graph = from_edge_list(edge_list)
```

[8]:
```
graph
```

[8]:
```
{'names': array(['Alice', 'Bob', 'Carey', 'David'], dtype='<U5'),
 'adjacency': <4x4 sparse matrix of type '<class 'numpy.int64'>'
        with 10 stored elements in Compressed Sparse Row format>}
```

[9]:
```
adjacency = graph.adjacency
names = graph.names
```

[10]:
```
image = svg_graph(adjacency, names=names)
SVG(image)
```

[10]:

By default, the weight of each edge is the number of occurrences of the corresponding link:

[11]:
```
edge_list_new = edge_list + [("Alice", "Bob"), ("Alice", "David"), ("Alice", "Bob")]
graph = from_edge_list(edge_list_new)
```

[12]:
```
adjacency = graph.adjacency
names = graph.names
```

[13]:
```
image = svg_graph(adjacency, names=names)
SVG(image)
```

[13]:

You can make the graph unweighted.

[14]:
```
graph = from_edge_list(edge_list_new, weighted=False)
```

[15]:
```
adjacency = graph.adjacency
names = graph.names
```

[16]:
```
image = svg_graph(adjacency, names=names)
SVG(image)
```

[16]:

Again, you can make the graph directed:

[17]:
```
graph = from_edge_list(edge_list, directed=True)
```

[18]:
```
graph
```

[18]:
```
{'names': array(['Alice', 'Bob', 'Carey', 'David'], dtype='<U5'),
 'adjacency': <4x4 sparse matrix of type '<class 'numpy.int64'>'
        with 5 stored elements in Compressed Sparse Row format>}
```

[19]:
```
adjacency = graph.adjacency
names = graph.names
```

[20]:
```
image = svg_digraph(adjacency, names=names)
SVG(image)
```

[20]:

The graph can also have explicit weights:

[21]:
```
edge_list = [("Alice", "Bob", 3), ("Bob", "Carey", 2), ("Alice", "David", 1), ("Carey",
↪"David", 2), ("Bob", "David", 3)]
graph = from_edge_list(edge_list)
```

[22]:
```
adjacency = graph.adjacency
names = graph.names
```

[23]:
```
image = svg_graph(adjacency, names=names, display_edge_weight=True, display_node_
↪weight=True)
SVG(image)
```

[23]:

For a bipartite graph:

[24]:
```
edge_list = [("Alice", "Football"), ("Bob", "Tennis"), ("David", "Football"), ("Carey",
↪"Tennis"), ("Carey", "Football")]
graph = from_edge_list(edge_list, bipartite=True)
```

[25]:
```
biadjacency = graph.biadjacency
names = graph.names
names_col = graph.names_col
```

[26]:
```
image = svg_bigraph(biadjacency, names_row=names, names_col=names_col)
SVG(image)
```

[26]:

From an adjacency list

You can also load a graph from an adjacency list, given as a list of lists or a dictionary of lists:

[27]:
```
adjacency_list =[[0, 1, 2], [2, 3]]
adjacency = from_adjacency_list(adjacency_list, directed=True)
```

[28]:
```
image = svg_digraph(adjacency)
SVG(image)
```

[28]:

[29]:
```
adjacency_dict = {"Alice": ["Bob", "David"], "Bob": ["Carey", "David"]}
graph = from_adjacency_list(adjacency_dict, directed=True)
```

[30]:
```
adjacency = graph.adjacency
names = graph.names
```

[31]:
```
image = svg_digraph(adjacency, names=names)
SVG(image)
```

[31]:

From a dataframe

Your dataframe might consist of a list of edges.

[32]:
```
df = pd.read_csv('miserables.tsv', sep='\t', names=['character_1', 'character_2'])
```

[33]:
```
df.head()
```

[33]:
```
  character_1      character_2
0      Myriel         Napoleon
1      Myriel  Mlle Baptistine
2      Myriel     Mme Magloire
3      Myriel   Countess de Lo
4      Myriel         Geborand
```

[34]:
```
edge_list = list(df.itertuples(index=False))
```

[35]:
```
graph = from_edge_list(edge_list)
```

[36]:
```
graph
```

[36]:
```
{'names': array(['Anzelma', 'Babet', 'Bahorel', 'Bamatabois', 'Baroness',
        'Blacheville', 'Bossuet', 'Boulatruelle', 'Brevet', 'Brujon',
        'Champmathieu', 'Champtercier', 'Chenildieu', 'Child1', 'Child2',
        'Claquesous', 'Cochepaille', 'Combeferre', 'Cosette', 'Count',
        'Countess de Lo', 'Courfeyrac', 'Cravatte', 'Dahlia', 'Enjolras',
        'Eponine', 'Fameuil', 'Fantine', 'Fauchelevent', 'Favourite',
        'Feuilly', 'Gavroche', 'Geborand', 'Gervais', 'Gillenormand',
```

(continues on next page)

(continued from previous page)

```
       'Grantaire', 'Gribier', 'Gueulemer', 'Isabeau', 'Javert', 'Joly',
       'Jondrette', 'Judge', 'Labarre', 'Listolier', 'Lt Gillenormand',
       'Mabeuf', 'Magnon', 'Marguerite', 'Marius', 'Mlle Baptistine',
       'Mlle Gillenormand', 'Mlle Vaubois', 'Mme Burgon', 'Mme Der',
       'Mme Hucheloup', 'Mme Magloire', 'Mme Pontmercy', 'Mme Thenardier',
       'Montparnasse', 'MotherInnocent', 'MotherPlutarch', 'Myriel',
       'Napoleon', 'Old man', 'Perpetue', 'Pontmercy', 'Prouvaire',
       'Scaufflaire', 'Simplice', 'Thenardier', 'Tholomyes', 'Toussaint',
       'Valjean', 'Woman1', 'Woman2', 'Zephine'], dtype='<U17'),
 'adjacency': <77x77 sparse matrix of type '<class 'numpy.int64'>'
       with 508 stored elements in Compressed Sparse Row format>}
```

[37]:
```
df = pd.read_csv('movie_actor.tsv', sep='\t', names=['movie', 'actor'])
```

[38]:
```
df.head()
```

[38]:
```
                   movie                 actor
0              Inception     Leonardo DiCaprio
1              Inception      Marion Cotillard
2              Inception  Joseph Gordon Lewitt
3  The Dark Knight Rises      Marion Cotillard
4  The Dark Knight Rises  Joseph Gordon Lewitt
```

[39]:
```
edge_list = list(df.itertuples(index=False))
```

[40]:
```
graph = from_edge_list(edge_list, bipartite=True)
```

[41]:
```
graph
```

[41]:
```
{'names_row': array(['007 Spectre', 'Aviator', 'Crazy Stupid Love', 'Drive',
       'Fantastic Beasts 2', 'Inception', 'Inglourious Basterds',
       'La La Land', 'Midnight In Paris', 'Murder on the Orient Express',
       'The Big Short', 'The Dark Knight Rises',
       'The Grand Budapest Hotel', 'The Great Gatsby', 'Vice'],
      dtype='<U28'),
 'names': array(['007 Spectre', 'Aviator', 'Crazy Stupid Love', 'Drive',
       'Fantastic Beasts 2', 'Inception', 'Inglourious Basterds',
       'La La Land', 'Midnight In Paris', 'Murder on the Orient Express',
       'The Big Short', 'The Dark Knight Rises',
       'The Grand Budapest Hotel', 'The Great Gatsby', 'Vice'],
      dtype='<U28'),
 'names_col': array(['Brad Pitt', 'Carey Mulligan', 'Christian Bale',
       'Christophe Waltz', 'Emma Stone', 'Johnny Depp',
       'Joseph Gordon Lewitt', 'Jude Law', 'Lea Seydoux',
       'Leonardo DiCaprio', 'Marion Cotillard', 'Owen Wilson',
       'Ralph Fiennes', 'Ryan Gosling', 'Steve Carell', 'Willem Dafoe'],
      dtype='<U28'),
 'biadjacency': <15x16 sparse matrix of type '<class 'numpy.int64'>'
       with 41 stored elements in Compressed Sparse Row format>}
```

For categorical data, you can use `pandas` to get a bipartite graph between samples and features. We show an example taken from the Adult Income dataset.

[42]:
```
df = pd.read_csv('adult-income.csv')
```

[43]:
```
df.head()
```

[43]:
```
     age          workclass          occupation    relationship   gender  \
0  40-49          State-gov        Adm-clerical   Not-in-family     Male
1  50-59   Self-emp-not-inc     Exec-managerial         Husband     Male
2  40-49            Private   Handlers-cleaners   Not-in-family     Male
3  50-59            Private   Handlers-cleaners         Husband     Male
4  30-39            Private      Prof-specialty            Wife   Female

   income
0   <=50K
1   <=50K
2   <=50K
3   <=50K
4   <=50K
```

[44]:
```
df_binary = pd.get_dummies(df, sparse=True)
```

[45]:
```
df_binary.head()
```

[45]:
```
   age_20-29  age_30-39  age_40-49  age_50-59  age_60-69  age_70-79  \
0          0          0          1          0          0          0
1          0          0          0          1          0          0
2          0          0          1          0          0          0
3          0          0          0          1          0          0
4          0          1          0          0          0          0

   age_80-89  age_90-99  workclass_ ?  workclass_ Federal-gov  ...  \
0          0          0             0                       0  ...
1          0          0             0                       0  ...
2          0          0             0                       0  ...
3          0          0             0                       0  ...
4          0          0             0                       0  ...

   relationship_ Husband  relationship_ Not-in-family  \
0                      0                            1
1                      1                            0
2                      0                            1
3                      1                            0
4                      0                            0

   relationship_ Other-relative  relationship_ Own-child  \
0                             0                        0
1                             0                        0
2                             0                        0
3                             0                        0
4                             0                        0

   relationship_ Unmarried  relationship_ Wife  gender_ Female  gender_ Male  \
0                        0                  0               0             1
1                        0                  0               0             1
```

(continues on next page)

(continued from previous page)

```
2                        0                  0                  0                  1
3                        0                  0                  0                  1
4                        0                  1                  1                  0

   income_ <=50K  income_ >50K
0              1             0
1              1             0
2              1             0
3              1             0
4              1             0

[5 rows x 42 columns]
```

[46]:
```
biadjacency = df_binary.sparse.to_coo()
```

[47]:
```
biadjacency = sparse.csr_matrix(biadjacency)
```

[48]:
```
# biadjacency matrix of the bipartite graph
biadjacency
```

[48]:
```
<32561x42 sparse matrix of type '<class 'numpy.uint8'>'
        with 195366 stored elements in Compressed Sparse Row format>
```

[49]:
```
# names of columns
names_col = list(df_binary)
```

[50]:
```
len(names_col)
```

[50]:
```
42
```

[51]:
```
names_col[:8]
```

[51]:
```
['age_20-29',
 'age_30-39',
 'age_40-49',
 'age_50-59',
 'age_60-69',
 'age_70-79',
 'age_80-89',
 'age_90-99']
```

From a CSV file

You can directly load a graph from a CSV or TSV file:

[52]:
```
graph = from_csv('miserables.tsv')
```

[53]:
```
graph
```

[53]:
```
{'names': array(['Anzelma', 'Babet', 'Bahorel', 'Bamatabois', 'Baroness',
        'Blacheville', 'Bossuet', 'Boulatruelle', 'Brevet', 'Brujon',
```

(continues on next page)

(continued from previous page)

```
        'Champmathieu', 'Champtercier', 'Chenildieu', 'Child1', 'Child2',
        'Claquesous', 'Cochepaille', 'Combeferre', 'Cosette', 'Count',
        'Countess de Lo', 'Courfeyrac', 'Cravatte', 'Dahlia', 'Enjolras',
        'Eponine', 'Fameuil', 'Fantine', 'Fauchelevent', 'Favourite',
        'Feuilly', 'Gavroche', 'Geborand', 'Gervais', 'Gillenormand',
        'Grantaire', 'Gribier', 'Gueulemer', 'Isabeau', 'Javert', 'Joly',
        'Jondrette', 'Judge', 'Labarre', 'Listolier', 'Lt Gillenormand',
        'Mabeuf', 'Magnon', 'Marguerite', 'Marius', 'Mlle Baptistine',
        'Mlle Gillenormand', 'Mlle Vaubois', 'Mme Burgon', 'Mme Der',
        'Mme Hucheloup', 'Mme Magloire', 'Mme Pontmercy', 'Mme Thenardier',
        'Montparnasse', 'MotherInnocent', 'MotherPlutarch', 'Myriel',
        'Napoleon', 'Old man', 'Perpetue', 'Pontmercy', 'Prouvaire',
        'Scaufflaire', 'Simplice', 'Thenardier', 'Tholomyes', 'Toussaint',
        'Valjean', 'Woman1', 'Woman2', 'Zephine'], dtype='<U17'),
 'adjacency': <77x77 sparse matrix of type '<class 'numpy.int64'>'
        with 508 stored elements in Compressed Sparse Row format>}
```

[54]:
```
graph = from_csv('movie_actor.tsv', bipartite=True)
```

[55]:
```
graph
```

[55]:
```
{'names_row': array(['007 Spectre', 'Aviator', 'Crazy Stupid Love', 'Drive',
        'Fantastic Beasts 2', 'Inception', 'Inglourious Basterds',
        'La La Land', 'Midnight In Paris', 'Murder on the Orient Express',
        'The Big Short', 'The Dark Knight Rises',
        'The Grand Budapest Hotel', 'The Great Gatsby', 'Vice'],
       dtype='<U28'),
 'names': array(['007 Spectre', 'Aviator', 'Crazy Stupid Love', 'Drive',
        'Fantastic Beasts 2', 'Inception', 'Inglourious Basterds',
        'La La Land', 'Midnight In Paris', 'Murder on the Orient Express',
        'The Big Short', 'The Dark Knight Rises',
        'The Grand Budapest Hotel', 'The Great Gatsby', 'Vice'],
       dtype='<U28'),
 'names_col': array(['Brad Pitt', 'Carey Mulligan', 'Christian Bale',
        'Christophe Waltz', 'Emma Stone', 'Johnny Depp',
        'Joseph Gordon Lewitt', 'Jude Law', 'Lea Seydoux',
        'Leonardo DiCaprio', 'Marion Cotillard', 'Owen Wilson',
        'Ralph Fiennes', 'Ryan Gosling', 'Steve Carell', 'Willem Dafoe'],
       dtype='<U28'),
 'biadjacency': <15x16 sparse matrix of type '<class 'numpy.int64'>'
        with 41 stored elements in Compressed Sparse Row format>}
```

The graph can also be given in the form of adjacency lists (check the function `from_csv`).

From a GraphML file

You can also load a graph stored in the GraphML format.

[56]:
```
graph = from_graphml('miserables.graphml')
adjacency = graph.adjacency
names = graph.names
```

[57]:
```
# Directed graph
graph = from_graphml('painters.graphml')
adjacency = graph.adjacency
names = graph.names
```

From NetworkX

NetworkX has import and export functions from and towards the CSR format.

Other options

- You want to test our toy graphs
- You want to generate a graph from a model
- You want to load a graph from existing repositories (see NetSet and KONECT)

Take a look at the other tutorials of the **data** section!

3.18.2 Datasets

This tutorial shows how to load graphs from existing datasets, NetSet and Konect.

[1]:
```
from sknetwork.data import load_netset, load_konect
```

NetSet

Loading a graph from the NetSet collection.

[2]:
```
graph = load_netset('openflights')
adjacency = graph.adjacency
names = graph.names
```

```
Parsing files...
Done.
```

[3]:
```
# to get all fields
graph
```

[3]:
```
{'meta': {'name': 'openflights',
  'description': 'Airports with daily number of flights between them.',
  'source': 'https://openflights.org'},
 'adjacency': <3097x3097 sparse matrix of type '<class 'numpy.int64'>'
        with 36386 stored elements in Compressed Sparse Row format>,
```

(continues on next page)

(continued from previous page)

```
 'names': array(['Goroka Airport', 'Madang Airport', 'Mount Hagen Kagamuga Airport',
        ..., 'Saumlaki/Olilit Airport', 'Tarko-Sale Airport',
        'Alashankou Bole (Bortala) airport'], dtype='<U65'),
 'position': array([[145.39199829,  -6.08168983],
        [145.78900147,  -5.20707989],
        [144.29600525,  -5.82678986],
        ...,
        [131.30599976,  -7.98860979],
        [ 77.81809998,  64.93080139],
        [ 82.3       ,  44.895     ]])}
```

[4]:
```
# Directed graph
graph = load_netset('wikivitals')
adjacency = graph.adjacency
names = graph.names
labels = graph.labels
```

```
Parsing files...
Done.
```

[5]:
```
# Bipartite graph
graph = load_netset('cinema')
biadjacency = graph.biadjacency
```

```
Parsing files...
Done.
```

Konect

Loading a graph from the Konect collection.

[6]:
```
graph = load_konect('dolphins')
adjacency = graph.adjacency
```

```
Loading from local bundle...
```

[7]:
```
graph
```

[7]:
```
{'meta': {'name': ' Dolphins',
  'code': ' DO',
  'url': '  http://www-personal.umich.edu/~mejn/netdata/',
  'category': '  Animal',
  'description': ' Dolphin-dolphin associations',
  'cite': ' konect:dolphins',
  'long-description': ' This is a directed social network of bottlenose dolphins.  The␣
↪nodes are the bottlenose dolphins (genus <i>Tursiops</i>) of a bottlenose dolphin␣
↪community living off Doubtful Sound, a fjord in New Zealand (spelled <i>fiord</i> in␣
↪New Zealand).  An edge indicates a frequent association.  The dolphins were observed␣
↪between 1994 and 2001. ',
  'entity-names': ' dolphin',
  'relationship-names': ' association',
  'extr': ' dolphins',
```

(continues on next page)

(continued from previous page)

```
 'timeiso': ' 1994/2001'},
'adjacency': <62x62 sparse matrix of type '<class 'numpy.int64'>'
       with 318 stored elements in Compressed Sparse Row format>,
'names': array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17,
       18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34,
       35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51,
       52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62])}
```

3.18.3 Models

This notebook shows how to load some graphs based on simple models.

[1]:
```
from IPython.display import SVG
```

[2]:
```
import numpy as np
```

[3]:
```
from sknetwork.data import erdos_renyi, block_model, linear_graph, cyclic_graph, linear_
↪digraph, cyclic_digraph, grid, albert_barabasi, watts_strogatz
from sknetwork.visualization import svg_graph, svg_digraph
```

Erdos-Renyi model

[4]:
```
adjacency = erdos_renyi(20, 0.2)
```

[5]:
```
image = svg_graph(adjacency)
SVG(image)
```

[5]:

Stochastic block model

[6]:
```
graph = block_model([20,25,30], p_in=[0.5,0.4,0.3], p_out=0.02, metadata=True)
adjacency = graph.adjacency
labels = graph.labels
```

[7]:
```
image = svg_graph(adjacency, labels=labels)
SVG(image)
```

[7]:

Linear graph

[8]:
```
graph = linear_graph(8, metadata=True)
adjacency = graph.adjacency
position = graph.position
```

[9]:
```
image = svg_graph(adjacency, position)
SVG(image)
```

[9]:

[10]:
```
# adjacency matrix only
adjacency = linear_graph(8)
```

[11]:
```
# directed
graph = linear_digraph(8, metadata=True)
adjacency = graph.adjacency
position = graph.position
```

[12]:
```
image = svg_digraph(adjacency, position)
SVG(image)
```

[12]:

Cyclic graph

[13]:
```
graph = cyclic_graph(8, metadata=True)
adjacency = graph.adjacency
position = graph.position
```

[14]:
```
image = svg_graph(adjacency, position, width=200, height=200)
```

[15]:
```
SVG(image)
```

[15]:

Grid

[16]:
```
graph = grid(6, 4, metadata=True)
adjacency = graph.adjacency
position = graph.position
```

[17]:
```
image = svg_graph(adjacency, position)
```

[18]:
```
SVG(image)
```

[18]:

Albert-Barabasi model

[19]:
```
adjacency = albert_barabasi(n=100, degree=3)
```

[20]:
```
image = svg_graph(adjacency, labels={i:0 for i in range(3)}, display_node_weight=True,␣
↪node_order=np.flip(np.arange(100)))
SVG(image)
```

[20]:

Watts-Strogatz model

[21]:
```
adjacency = watts_strogatz(n=100, degree=6, prob=0.2)
```

[22]:
```
image = svg_graph(adjacency, display_node_weight=True, node_size_max=10)
SVG(image)
```

[22]:

3.18.4 Toy graphs

This notebook shows how to load some toy graphs.

[1]:
```
from IPython.display import SVG
```

[2]:
```
import numpy as np
```

[3]:
```
from sknetwork.data import house, bow_tie, karate_club, miserables, painters, hourglass,␣
↪star_wars, movie_actor
from sknetwork.visualization import svg_graph, svg_digraph, svg_bigraph
```

House graph

[4]:
```
graph = house(metadata=True)
adjacency = graph.adjacency
position = graph.position
```

[5]:
```
image = svg_graph(adjacency, position, scale=0.5)
SVG(image)
```

[5]:

[6]:
```
# adjacency matrix only
adjacency = house()
```

Bow tie

[7]:
```
graph = bow_tie(metadata=True)
adjacency = graph.adjacency
position = graph.position
```

[8]:
```
image = svg_graph(adjacency, position, scale=0.5)
```

[9]:
```
SVG(image)
```

[9]:

Karate club

[10]:
```
graph = karate_club(metadata=True)
adjacency = graph.adjacency
position = graph.position
labels = graph.labels
```

[11]:
```
image = svg_graph(adjacency, position, labels=labels)
SVG(image)
```

[11]:

Les Miserables

[12]:
```
graph = miserables(metadata=True)
adjacency = graph.adjacency
position = graph.position
names = graph.names
```

[13]:
```
image = svg_graph(adjacency, position, names, scale=2)
SVG(image)
```

[13]:

Painters

[14]:
```
graph = painters(metadata=True)
adjacency = graph.adjacency
names = graph.names
position = graph.position
```

[15]:
```
image = svg_digraph(adjacency, position, names)
SVG(image)
```

[15]:

Star wars

[16]:
```
# bipartite graph
graph = star_wars(metadata=True)
biadjacency = graph.biadjacency
names_row = graph.names_row
names_col = graph.names_col
```

[17]:
```
image = svg_bigraph(biadjacency, names_row, names_col)
SVG(image)
```

[17]:

[18]:
```
# biadjacency matrix only
biadjacency = star_wars()
```

Movie-actor

[19]:
```
# bipartite graph
graph = movie_actor(metadata=True)
biadjacency = graph.biadjacency
names_row = graph.names_row
names_col = graph.names_col
```

[20]:
```
image = svg_bigraph(biadjacency, names_row, names_col)
SVG(image)
```

[20]:

3.18.5 Nearest-neighbor graphs

Building graphs from the Iris dataset using k-nearest neighbors.

[1]:
```
from IPython.display import SVG
```

[2]:
```
import pickle
```

[3]:
```
from sknetwork.embedding import Spring
from sknetwork.utils import KNNDense, CNNDense
from sknetwork.visualization import svg_graph
```

Nearest neighbors

[4]:
```
iris = pickle.load(open('iris.p', 'rb'))
```

[5]:
```
data = iris['data']
```

[6]:
```
labels = iris['labels']
```

[7]:
```
knn = KNNDense(n_neighbors=3, undirected=True)
adjacency = knn.fit_transform(data)
```

[8]:
```
image = svg_graph(adjacency, labels=labels, display_edge_weight=False)
```

[9]:
```
SVG(image)
```

[9]:

Component-wise nearest neighbors

Here the k nearest neighbors are searched per component. Thus if data has dimension N, there are at most $k \times N$ nearest neighbors per sample.

[10]:
```
cnn = CNNDense(n_neighbors=2, undirected=True)
adjacency = cnn.fit_transform(data)
```

[11]:
```
image = svg_graph(adjacency, labels=labels, display_edge_weight=False)
```

[12]:
```
SVG(image)
```

[12]:

3.18.6 Save

This notebook shows how to save and load graphs using the Bunch format.

[1]:
```
import numpy as np
from scipy import sparse
```

[2]:
```
from sknetwork.data import Bunch, load, save
```

[3]:
```
# random graph
adjacency = sparse.csr_matrix(np.random.random((10, 10)) < 0.2)
```

[4]:
```
# names
names = list('abcdefghij')
```

[5]:
```
graph = Bunch()
graph.adjacency = adjacency
graph.names = np.array(names)
```

[6]:
```
save('mygraph', graph)
```

[7]:
```
graph = load('mygraph')
```

[8]:
```
graph
```

[8]:
```
{'adjacency': <10x10 sparse matrix of type '<class 'numpy.bool_'>'
        with 19 stored elements in Compressed Sparse Row format>,
 'names': array(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'], dtype='<U1')}
```

3.19 Topology

3.19.1 Connected components

This notebook illustrates the search for connected components in graphs.

[1]:
```
from IPython.display import SVG
```

[2]:
```
import numpy as np
```

[3]:
```
from sknetwork.data import karate_club, painters, movie_actor
from sknetwork.topology import get_connected_components, get_largest_connected_component
from sknetwork.visualization import svg_graph, svg_digraph, svg_bigraph
from sknetwork.utils.format import bipartite2undirected
```

Graphs

[4]:
```
graph = karate_club(metadata=True)
adjacency = graph.adjacency
position = graph.position
```

[5]:
```
# subgraph
k = 15
adjacency = adjacency[:k][:,:k]
position = position[:k]
```

[6]:
```
# connected components
labels = get_connected_components(adjacency)
```

[7]:
```
image = svg_graph(adjacency, position, labels=labels)
SVG(image)
```

[7]:

[8]:
```
# largest connected component
new_adjacency, index = get_largest_connected_component(adjacency, return_index=True)
```

[9]:
```
len(index)
```

[9]:
```
14
```

Directed graphs

[10]:
```
graph = painters(metadata=True)
adjacency = graph.adjacency
names = graph.names
position = graph.position
```

[11]:
```
# weak connected components
labels = get_connected_components(adjacency)
```

[12]:
```
image = svg_digraph(adjacency, position=position, names=names, labels=labels)
SVG(image)
```

[12]:

[13]:
```
# strong connected components
labels = get_connected_components(adjacency, connection='strong')
```

[14]:
```
image = svg_digraph(adjacency, position, names, labels)
SVG(image)
```

[14]:

[15]:
```
# largest connected component
new_adjacency, index = get_largest_connected_component(adjacency, connection='strong',␣
↪return_index=True)
```

[16]:
```
image = svg_digraph(new_adjacency, position[index], names[index])
SVG(image)
```

[16]:

Bipartite graphs

[17]:
```
graph = movie_actor(metadata=True)
biadjacency = graph.biadjacency
names_row = graph.names_row
names_col = graph.names_col
```

[18]:
```
# subgraph
k = 5
biadjacency = biadjacency[k:]
names_row = names_row[k:]
```

[19]:
```
labels = get_connected_components(biadjacency, force_bipartite=True)
```

[20]:
```
n_row, _ = biadjacency.shape
labels_row = labels[:n_row]
labels_col = labels[n_row:]
```

[21]:
```
image = svg_bigraph(biadjacency, names_row, names_col, labels_row, labels_col)
SVG(image)
```

[21]:

[22]:
```
# largest connected component
new_biadjacency, index = get_largest_connected_component(biadjacency, force_
↪bipartite=True, return_index=True)
```

[23]:
```
n_row, n_col = new_biadjacency.shape
index_row = index[:n_row]
index_col = index[n_row:]
```

[24]:
```
image = svg_bigraph(new_biadjacency, names_row[index_row], names_col[index_col])
SVG(image)
```

[24]:

3.19.2 Core decomposition

This notebook illustrates the k-core decomposition of graphs.

[1]:
```
from IPython.display import SVG
```

[2]:
```
import numpy as np
```

[3]:
```
from sknetwork.data import karate_club, painters
from sknetwork.topology import CoreDecomposition
from sknetwork.visualization import svg_graph, svg_digraph
from sknetwork.utils import directed2undirected
```

Graphs

[4]:
```
graph = karate_club(metadata=True)
adjacency = graph.adjacency
position = graph.position
```

[5]:
```
core = CoreDecomposition()
labels = core.fit_transform(adjacency)
```

[6]:
```
image = svg_graph(adjacency, position, scores=labels)
SVG(image)
```

[6]:

Directed graphs

[7]:
```
graph = painters(metadata=True)
adjacency = graph.adjacency
names = graph.names
position = graph.position
```

[8]:
```
labels = core.fit_transform(directed2undirected(adjacency))
```

[9]:
```
image = svg_digraph(adjacency, position, names, scores=labels)
SVG(image)
```

[9]:

3.19.3 Triangles and cliques

This notebook illustrates clique counting and evaluation of the clustering coefficient of a graph.

[1]:
```
from IPython.display import SVG
```

[2]:
```
import numpy as np
```

[3]:
```
from sknetwork.data import karate_club
from sknetwork.topology import Triangles, Cliques
from sknetwork.visualization import svg_graph
```

Triangles

[4]:
```
graph = karate_club(metadata=True)
adjacency = graph.adjacency
position = graph.position
```

[5]:
```
# graph
image = svg_graph(adjacency, position)
SVG(image)
```

[5]:

[6]:
```
# number of triangles
triangles = Triangles()
triangles.fit_transform(adjacency)
```

[6]:
```
45
```

[7]:
```
# coefficient of clustering
np.round(triangles.clustering_coef_, 2)
```

[7]:
```
0.26
```

Cliques

[8]:
```
# number of 4-cliques
cliques = Cliques(4)
cliques.fit_transform(adjacency)
```

[8]:
```
11
```

3.19.4 Graph isomorphism

This notebook illustrates the Weisfeiler-Lehman test of isomorphism.

[1]:
```
from IPython.display import SVG
import numpy as np
from sknetwork.data import house
from sknetwork.topology import WeisfeilerLehman, are_isomorphic
from sknetwork.visualization import svg_graph
```

Graph labeling

[2]:
```
graph = house(metadata=True)
adjacency = graph.adjacency
position = graph.position
```

[3]:
```
weisfeiler_lehman = WeisfeilerLehman()
labels = weisfeiler_lehman.fit_transform(adjacency)
```

[4]:
```
image = svg_graph(adjacency, position, labels=labels)
SVG(image)
```

[4]:

[5]:
```
# first iteration
weisfeiler_lehman = WeisfeilerLehman(max_iter=1)
labels = weisfeiler_lehman.fit_transform(adjacency)
```

[6]:
```
image = svg_graph(adjacency, position, labels=labels)
SVG(image)
```

[6]:

Weisfeiler-Lehman test

[7]:
```
adjacency_1 = house()

n = adjacency_1.indptr.shape[0] - 1
reorder = list(range(n))
np.random.shuffle(reorder)
adjacency_2 = adjacency_1[reorder][:, reorder]

are_isomorphic(adjacency_1, adjacency_2)
```

[7]:
```
True
```

3.20 Path

3.20.1 Distance

This notebook illustrates the computation of distances between nodes in graphs (in number of hops).

[1]:
```
from IPython.display import SVG
```

[2]:
```
import numpy as np
```

[3]:
```
from sknetwork.data import miserables, painters, movie_actor
from sknetwork.path import get_distances
from sknetwork.visualization import svg_graph, svg_digraph, svg_bigraph
from sknetwork.utils import bipartite2undirected
```

Graphs

[4]:
```
graph = miserables(metadata=True)
adjacency = graph.adjacency
names = graph.names
position = graph.position
```

[5]:
```
napoleon = 1
distances = get_distances(adjacency, sources=napoleon)
```

[6]:
```
image = svg_graph(adjacency, position, names, scores = -distances, seeds=[napoleon],␣
↪scale = 1.5)
SVG(image)
```

[6]:

Directed graphs

[7]:
```
graph = painters(metadata=True)
adjacency = graph.adjacency
names = graph.names
position = graph.position
```

[8]:
```
cezanne = 11
distances = get_distances(adjacency, sources=cezanne)
```

[9]:
```
dist_neg= {i: -d for i, d in enumerate(distances) if d < np.inf}
```

[10]:
```
image = svg_digraph(adjacency, position, names, scores=dist_neg , seeds=[cezanne])
SVG(image)
```

[10]:

Bipartite graphs

[11]:
```
graph = movie_actor(metadata=True)
biadjacency = graph.biadjacency
names_row = graph.names_row
names_col = graph.names_col
```

[12]:
```
adjacency = bipartite2undirected(biadjacency)
n_row, _ = biadjacency.shape
```

[13]:
```
seydoux = 9
distances = get_distances(adjacency, sources=seydoux + n_row)
```

[14]:
```
image = svg_bigraph(biadjacency, names_row, names_col, scores_col=-distances[n_row:],␣
↪seeds_col=seydoux)
SVG(image)
```

[14]:

3.20.2 Shortest paths

This notebook illustrates the search for shortest paths in graphs.

[1]:
```
from IPython.display import SVG
```

[2]:
```
import numpy as np
```

[3]:
```
from sknetwork.data import miserables, painters, movie_actor
from sknetwork.path import get_shortest_path
from sknetwork.visualization import svg_graph, svg_digraph, svg_bigraph
from sknetwork.utils import bipartite2undirected
```

Graphs

[4]:
```
graph = miserables(metadata=True)
adjacency = graph.adjacency
names = graph.names
position = graph.position
```

[5]:
```
# shortest path
napoleon = 1
jondrette = 46
path = get_shortest_path(adjacency, sources=napoleon, targets=jondrette)
```

[6]:
```
# visualization
edge_labels = [(path[k], path[k + 1], 0) for k in range(len(path) - 1)]
```

[7]:
```
image = svg_graph(adjacency, position, names, edge_labels=edge_labels, edge_width=3,␣
↪display_edge_weight=False, scale = 1.5)
SVG(image)
```

[7]:

Directed graphs

[8]:
```
graph = painters(metadata=True)
adjacency = graph.adjacency
names = graph.names
position = graph.position
```

[9]:
```
# shortest path
klimt = 6
vinci = 9
path = get_shortest_path(adjacency, sources=klimt, targets=vinci)
```

[10]:
```
edge_labels = [(path[k], path[k + 1], 0) for k in range(len(path) - 1)]
```

[11]:
```
image = svg_digraph(adjacency, position, names, edge_labels=edge_labels, edge_width=2)
SVG(image)
```

[11]:

Bipartite graphs

[12]:
```
graph = movie_actor(metadata=True)
biadjacency = graph.biadjacency
names_row = graph.names_row
names_col = graph.names_col
```

[13]:
```
adjacency = bipartite2undirected(biadjacency)
n_row = biadjacency.shape[0]
```

[14]:
```
# shortest path
seydoux = 9
lewitt = 2
path = get_shortest_path(adjacency, sources=seydoux + n_row, targets=lewitt + n_row)
```

[15]:
```
# visualization
edge_labels = []
labels_row = {}
labels_col = {}
for k in range(len(path) - 1):
    i = path[k]
    j = path[k + 1]
    # row first
    if i > j:
        i, j = j, i
    j -= n_row
    labels_row[i] = 0
    labels_col[j] = 0
    edge_labels.append((i, j, 0))
```

[16]:
```
image = svg_bigraph(biadjacency, names_row, names_col, labels_row, labels_col,
                    edge_labels=edge_labels, edge_color='gray', edge_width=3)
SVG(image)
```

[16]:

3.21 Clustering

3.21.1 Louvain

This notebook illustrates the clustering of a graph by the Louvain algorithm.

[1]:
```
from IPython.display import SVG
```

[2]:
```
import numpy as np
```

[3]:
```
from sknetwork.data import karate_club, painters, movie_actor
from sknetwork.clustering import Louvain, modularity, bimodularity
from sknetwork.linalg import normalize
from sknetwork.utils import bipartite2undirected, membership_matrix
from sknetwork.visualization import svg_graph, svg_digraph, svg_bigraph
```

Graphs

[4]:
```
graph = karate_club(metadata=True)
adjacency = graph.adjacency
position = graph.position
```

[5]:
```
louvain = Louvain()
labels = louvain.fit_transform(adjacency)
```

[6]:
```
labels_unique, counts = np.unique(labels, return_counts=True)
print(labels_unique, counts)
```

```
[0 1 2 3] [12 11  6  5]
```

[7]:
```
image = svg_graph(adjacency, position, labels=labels)
SVG(image)
```

[7]:

[8]:
```
# metric
modularity(adjacency, labels)
```

[8]:
```
0.4188034188034188
```

[9]:
```
# aggregate graph
adjacency_aggregate = louvain.aggregate_
```

[10]:
```
average = normalize(membership_matrix(labels).T)
position_aggregate = average.dot(position)
labels_unique, counts = np.unique(labels, return_counts=True)
```

[11]:
```
image = svg_graph(adjacency_aggregate, position_aggregate, counts, labels=labels_unique,
                  display_node_weight=True, node_weights=counts)
SVG(image)
```

[11]:

[12]:
```
# soft clustering (here probability of label 1)
scores = louvain.membership_[:,1].toarray().ravel()
```

[13]:
```
image = svg_graph(adjacency, position, scores=scores)
SVG(image)
```

[13]:

Directed graphs

[14]:
```
graph = painters(metadata=True)
adjacency = graph.adjacency
names = graph.names
position = graph.position
```

[15]:
```
# clustering
louvain = Louvain()
labels = louvain.fit_transform(adjacency)
```

[16]:
```
labels_unique, counts = np.unique(labels, return_counts=True)
print(labels_unique, counts)
```

```
[0 1 2] [5 5 4]
```

[17]:
```
image = svg_digraph(adjacency, position, names=names, labels=labels)
SVG(image)
```

[17]:

[18]:
```
modularity(adjacency, labels)
```

[18]:
```
0.32480000000000003
```

[19]:
```
# aggregate graph
adjacency_aggregate = louvain.aggregate_
```

[20]:
```
average = normalize(membership_matrix(labels).T)
position_aggregate = average.dot(position)
labels_unique, counts = np.unique(labels, return_counts=True)
```

[21]:
```
image = svg_digraph(adjacency_aggregate, position_aggregate, counts, labels=labels_
↪unique,
                    display_node_weight=True, node_weights=counts)
SVG(image)
```

[21]:

[22]:
```
# soft clustering
scores = louvain.membership_[:,1].toarray().ravel()
```

[23]:
```
image = svg_graph(adjacency, position, scores=scores)
SVG(image)
```

[23]:

Bipartite graphs

[24]:
```
graph = movie_actor(metadata=True)
biadjacency = graph.biadjacency
names_row = graph.names_row
names_col = graph.names_col
```

[25]:
```
# clustering
louvain = Louvain()
louvain.fit(biadjacency)
labels_row = louvain.labels_row_
labels_col = louvain.labels_col_
```

[26]:
```
image = svg_bigraph(biadjacency, names_row, names_col, labels_row, labels_col)
SVG(image)
```

[26]:

[27]:
```
# metric
bimodularity(biadjacency, labels_row, labels_col)
```

[27]:
```
0.5742630385487529
```

[28]:
```
# aggregate graph
biadjacency_aggregate = louvain.aggregate_
```

[29]:
```
labels_unique_row, counts_row = np.unique(labels_row, return_counts=True)
labels_unique_col, counts_col = np.unique(labels_col, return_counts=True)
```

[30]:
```
image = svg_bigraph(biadjacency_aggregate, counts_row, counts_col, labels_unique_row,␣
↪labels_unique_col,
                    display_node_weight=True, node_weights_row=counts_row, node_weights_
↪col=counts_col)
SVG(image)
```

[30]:

[31]:
```
# soft clustering
scores_row = louvain.membership_row_[:,1].toarray().ravel()
scores_col = louvain.membership_col_[:,1].toarray().ravel()
```

[32]:
```
image = svg_bigraph(biadjacency, names_row, names_col, scores_row=scores_row, scores_
↪col=scores_col)
SVG(image)
```

[32]:

3.21.2 Propagation

This notebook illustrates the clustering of a graph by label propagation.

[1]:
```
from IPython.display import SVG
```

[2]:
```
import numpy as np
```

[3]:
```
from sknetwork.data import karate_club, painters, movie_actor
from sknetwork.clustering import PropagationClustering, modularity, bimodularity
from sknetwork.linalg import normalize
from sknetwork.utils import bipartite2undirected, membership_matrix
from sknetwork.visualization import svg_graph, svg_digraph, svg_bigraph
```

Graphs

[4]:
```
graph = karate_club(metadata=True)
adjacency = graph.adjacency
position = graph.position
```

[5]:
```
propagation = PropagationClustering()
labels = propagation.fit_transform(adjacency)
```

[6]:
```
labels_unique, counts = np.unique(labels, return_counts=True)
print(labels_unique, counts)
```

```
[0 1] [19 15]
```

[7]:
```
image = svg_graph(adjacency, position, labels=labels)
SVG(image)
```

[7]:

[8]:
```
# metric
modularity(adjacency, labels)
```

[8]:
```
0.35231755424063116
```

[9]:
```
# aggregate graph
adjacency_aggregate = propagation.aggregate_
```

[10]:
```
average = normalize(membership_matrix(labels).T)
position_aggregate = average.dot(position)
labels_unique, counts = np.unique(labels, return_counts=True)
```

[11]:
```
image = svg_graph(adjacency_aggregate, position_aggregate, counts, labels=labels_unique,
                  display_node_weight=True, node_weights=counts)
SVG(image)
```

[11]:

[12]:
```
# soft clustering
scores = propagation.membership_[:,1].toarray().ravel()
```

[13]:
```
image = svg_graph(adjacency, position, scores=scores)
SVG(image)
```

[13]:

Directed graphs

[14]:
```
graph = painters(metadata=True)
adjacency = graph.adjacency
names = graph.names
position = graph.position
```

[15]:
```
propagation = PropagationClustering()
labels = propagation.fit_transform(adjacency)
```

[16]:
```
labels_unique, counts = np.unique(labels, return_counts=True)
print(labels_unique, counts)
```

```
[0 1] [10  4]
```

[17]:
```
image = svg_digraph(adjacency, position, names=names, labels=labels)
SVG(image)
```

[17]:

[18]:
```
modularity(adjacency, labels)
```

[18]:
```
0.256
```

[19]:
```
# aggregate graph
adjacency_aggregate = propagation.aggregate_
```

[20]:
```
average = normalize(membership_matrix(labels).T)
position_aggregate = average.dot(position)
labels_unique, counts = np.unique(labels, return_counts=True)
```

[21]:
```
image = svg_digraph(adjacency_aggregate, position_aggregate, counts, labels=labels_
↪unique,
                    display_node_weight=True, node_weights=counts)
SVG(image)
```

[21]:

[22]:
```
# soft clustering
scores = propagation.membership_[:,0].toarray().ravel()
```

[23]:
```
image = svg_graph(adjacency, position, scores=scores)
SVG(image)
```

[23]:

Bipartite graphs

[24]:
```
graph = movie_actor(metadata=True)
biadjacency = graph.biadjacency
names_row = graph.names_row
names_col = graph.names_col
```

[25]:
```
propagation = PropagationClustering()
propagation.fit(biadjacency)
labels_row = propagation.labels_row_
labels_col = propagation.labels_col_
```

[26]:
```
image = svg_bigraph(biadjacency, names_row, names_col, labels_row, labels_col)
SVG(image)
```

[26]:

[27]:
```
bimodularity(biadjacency, labels_row, labels_col)
```

[27]:
```
0.41496598639455773
```

[28]:
```
# aggregate graph
biadjacency_aggregate = propagation.aggregate_
```

[29]:
```
labels_unique_row, counts_row = np.unique(labels_row, return_counts=True)
labels_unique_col, counts_col = np.unique(labels_col, return_counts=True)
```

[30]:
```
image = svg_bigraph(biadjacency_aggregate, counts_row, counts_col, labels_unique_row,␣
↪labels_unique_col,
                    display_node_weight=True, node_weights_row=counts_row, node_weights_
↪col=counts_col)
SVG(image)
```

[30]:

[31]:
```
# soft clustering
scores_row = propagation.membership_row_[:,1].toarray().ravel()
scores_col = propagation.membership_col_[:,1].toarray().ravel()
```

[32]:
```
image = svg_bigraph(biadjacency, names_row, names_col, scores_row=scores_row, scores_
↪col=scores_col)
SVG(image)
```

[32]:

3.21.3 K-means

This notebook illustrates the clustering of a graph by k-means. This clustering involves the embedding of the graph in a space of low dimension.

[1]:
```
from IPython.display import SVG
```

[2]:
```
import numpy as np
```

[3]:
```
from sknetwork.data import karate_club, painters, movie_actor
from sknetwork.clustering import KMeans, modularity, bimodularity
from sknetwork.linalg import normalize
from sknetwork.embedding import GSVD
from sknetwork.utils import membership_matrix
from sknetwork.visualization import svg_graph, svg_digraph, svg_bigraph
```

Graphs

[4]:
```
graph = karate_club(metadata=True)
adjacency = graph.adjacency
position = graph.position
```

[5]:
```
kmeans = KMeans(n_clusters=2, embedding_method=GSVD(3))
labels = kmeans.fit_transform(adjacency)
```

[6]:
```
unique_labels, counts = np.unique(labels, return_counts=True)
print(unique_labels, counts)
```

```
[0 1] [19 15]
```

[7]:
```
image = svg_graph(adjacency, position, labels=labels)
SVG(image)
```

[7]:

[8]:
```
# metric
modularity(adjacency, labels)
```

[8]:
```
0.3599605522682445
```

[9]:
```
# aggregate graph
adjacency_aggregate = kmeans.aggregate_
```

[10]:
```
average = normalize(membership_matrix(labels).T)
position_aggregate = average.dot(position)
labels_unique, counts = np.unique(labels, return_counts=True)
```

[11]:
```
image = svg_graph(adjacency_aggregate, position_aggregate, counts, labels=labels_unique,
                  display_node_weight=True, node_weights=counts)
SVG(image)
```

[11]:

[12]:
```
# soft clustering (here probability of label 1)
scores = kmeans.membership_[:,1].toarray().ravel()
```

[13]:
```
image = svg_graph(adjacency, position, scores=scores)
SVG(image)
```

[13]:

Directed graphs

[14]:
```
graph = painters(metadata=True)
adjacency = graph.adjacency
position = graph.position
names = graph.names
```

[15]:
```
kmeans = KMeans(3, GSVD(3), co_cluster=False)
labels = kmeans.fit_transform(adjacency)
```

[16]:
```
image = svg_digraph(adjacency, position, names=names, labels=labels)
SVG(image)
```

[16]:

[17]:
```
modularity(adjacency, labels)
```

[17]:
```
0.23999999999999988
```

[18]:
```
# aggregate graph
adjacency_aggregate = kmeans.aggregate_
```

[19]:
```
average = normalize(membership_matrix(labels).T)
position_aggregate = average.dot(position)
labels_unique, counts = np.unique(labels, return_counts=True)
```

[20]:
```
image = svg_digraph(adjacency_aggregate, position_aggregate, counts, labels=labels_
↪unique,
                    display_node_weight=True, node_weights=counts)
SVG(image)
```

[20]:

[21]:
```
# soft clustering (probability of label 0)
scores = kmeans.membership_[:,0].toarray().ravel()
```

[22]:
```
image = svg_digraph(adjacency, position, scores=scores)
```

[23]:
```
SVG(image)
```

[23]:

Bipartite graphs

[24]:
```
graph = movie_actor(metadata=True)
biadjacency = graph.biadjacency
names_row = graph.names_row
names_col = graph.names_col
```

[25]:
```
kmeans = KMeans(3, GSVD(3), co_cluster=True)
kmeans.fit(biadjacency)
labels_row = kmeans.labels_row_
labels_col = kmeans.labels_col_
```

[26]:
```
image = svg_bigraph(biadjacency, names_row, names_col, labels_row, labels_col)
SVG(image)
```

[26]:

[27]:
```
# metric
bimodularity(biadjacency, labels_row, labels_col)
```

[27]:
```
0.4988662131519276
```

[28]:
```
# aggregate graph
biadjacency_aggregate = kmeans.aggregate_
```

[29]:
```
labels_unique_row, counts_row = np.unique(labels_row, return_counts=True)
labels_unique_col, counts_col = np.unique(labels_col, return_counts=True)
```

[30]:
```
image = svg_bigraph(biadjacency_aggregate, counts_row, counts_col, labels_unique_row,␣
↪labels_unique_col,
                    display_node_weight=True, node_weights_row=counts_row, node_weights_
↪col=counts_col)
SVG(image)
```

[30]:

[31]:
```
# soft clustering (here probability of label 1)
scores_row = kmeans.membership_row_[:,1].toarray().ravel()
scores_col = kmeans.membership_col_[:,1].toarray().ravel()
```

[32]:
```
image = svg_bigraph(biadjacency, names_row, names_col, scores_row=scores_row, scores_
↪col=scores_col)
SVG(image)
```

[32]:

3.22 Hierarchy

3.22.1 Paris

This notebook illustrates the hierarchical clustering of graphs by the Paris algorithm.

[1]:
```
from IPython.display import SVG
```

[2]:
```
import numpy as np
```

[3]:
```
from sknetwork.data import karate_club, painters, movie_actor
from sknetwork.hierarchy import Paris, cut_straight, dasgupta_score, tree_sampling_
↪divergence
from sknetwork.visualization import svg_graph, svg_digraph, svg_bigraph, svg_dendrogram
```

Graphs

[4]:
```
graph = karate_club(metadata=True)
adjacency = graph.adjacency
position = graph.position
```

[5]:
```
# hierarchical clustering
paris = Paris()
dendrogram = paris.fit_transform(adjacency)
```

[6]:
```
image = svg_dendrogram(dendrogram)
SVG(image)
```

[6]:

[7]:
```
# cuts of the dendrogram
labels = cut_straight(dendrogram)
print(labels)
```

```
[1 1 1 1 1 1 1 1 0 1 1 1 1 1 0 0 1 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

[8]:
```
n_clusters = 4
labels, dendrogram_aggregate = cut_straight(dendrogram, n_clusters, return_
↪dendrogram=True)
print(labels)
```

```
[0 0 0 0 3 3 3 0 1 0 3 0 0 0 1 1 3 0 1 0 1 0 1 2 2 2 2 2 2 2 1 2 1 1]
```

[9]:
```
_, counts = np.unique(labels, return_counts=True)
```

[10]:
```
# aggregate dendrogram
image = svg_dendrogram(dendrogram_aggregate, names=counts, rotate_names=False)
SVG(image)
```

[10]:

[11]:
```
# corresponding clustering
image = svg_graph(adjacency, position, labels=labels)
SVG(image)
```

[11]:

[12]:
```
# metrics
dasgupta_score(adjacency, dendrogram)
```

[12]:
```
0.6655354449472097
```

Directed graphs

[13]:
```
graph = painters(metadata=True)
adjacency = graph.adjacency
position = graph.position
names = graph.names
```

[14]:
```
# hierarchical clustering
paris = Paris()
dendrogram = paris.fit_transform(adjacency)
```

[15]:
```
image = svg_dendrogram(dendrogram, names, n_clusters=3, rotate=True)
SVG(image)
```

[15]:

[16]:
```
# cut with 3 clusters
labels = cut_straight(dendrogram, n_clusters = 3)
print(labels)
```

```
[0 0 1 0 1 1 2 0 0 1 0 0 0 2]
```

[17]:
```
image = svg_digraph(adjacency, position, names=names, labels=labels)
SVG(image)
```

[17]:

[18]:
```
# metrics
dasgupta_score(adjacency, dendrogram)
```

[18]:
```
0.5842857142857143
```

Bipartite graphs

[19]:
```
graph = movie_actor(metadata=True)
biadjacency = graph.biadjacency
names_row = graph.names_row
names_col = graph.names_col
```

[20]:
```
# hierarchical clustering
paris = Paris()
paris.fit(biadjacency)
dendrogram_row = paris.dendrogram_row_
dendrogram_col = paris.dendrogram_col_
dendrogram_full = paris.dendrogram_full_
```

[21]:
```
image = svg_dendrogram(dendrogram_row, names_row, n_clusters=4, rotate=True)
SVG(image)
```

[21]:

[22]:
```
image = svg_dendrogram(dendrogram_col, names_col, n_clusters=4, rotate=True)
SVG(image)
```

[22]:

[23]:
```
# cuts
labels = cut_straight(dendrogram_full, n_clusters = 4)
n_row = biadjacency.shape[0]
labels_row = labels[:n_row]
labels_col = labels[n_row:]
```

[24]:
```
image = svg_bigraph(biadjacency, names_row, names_col, labels_row, labels_col)
SVG(image)
```

[24]:

3.22.2 Ward

This notebook illustrates the hierarchical clustering of graphs by the Ward method, after embedding in a space of low dimension.

[1]:
```
from IPython.display import SVG
```

[2]:
```
import numpy as np
```

[3]:
```
from sknetwork.data import karate_club, painters, movie_actor
from sknetwork.embedding import Spectral
from sknetwork.hierarchy import Ward, cut_straight, dasgupta_score, tree_sampling_
↪divergence
from sknetwork.visualization import svg_graph, svg_digraph, svg_bigraph, svg_dendrogram
```

Graphs

[4]:
```
graph = karate_club(metadata=True)
adjacency = graph.adjacency
position = graph.position
```

[5]:
```
# hierarchical clustering
ward = Ward()
dendrogram = ward.fit_transform(adjacency)
```

[6]:
```
image = svg_dendrogram(dendrogram)
SVG(image)
```

[6]:

[7]:
```
# cuts
labels = cut_straight(dendrogram)
print(labels)
```

```
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 1 1]
```

[8]:
```
n_clusters = 4
labels, dendrogram_aggregate = cut_straight(dendrogram, n_clusters, return_
↪dendrogram=True)
print(labels)
```

```
[0 2 0 3 0 0 0 3 0 0 0 0 3 3 1 1 0 2 1 2 1 2 1 0 0 0 0 0 0 0 0 0 1 1]
```

[9]:
```
_, counts = np.unique(labels, return_counts=True)
```

[10]:
```
# aggregate dendrogram
image = svg_dendrogram(dendrogram_aggregate, names=counts, rotate_names=False)
SVG(image)
```

[10]:

[11]:
```
# clustering
image = svg_graph(adjacency, position, labels=labels)
SVG(image)
```

[11]:

[12]:
```
# metrics
dasgupta_score(adjacency, dendrogram)
```

[12]:
```
0.5082956259426847
```

[13]:
```
# other embedding
ward = Ward(embedding_method=Spectral(4))
```

Directed graphs

[14]:
```
graph = painters(metadata=True)
adjacency = graph.adjacency
position = graph.position
names = graph.names
```

[15]:
```
# hierarchical clustering
ward = Ward()
dendrogram = ward.fit_transform(adjacency)
```

[16]:
```
image = svg_dendrogram(dendrogram, names, n_clusters=3, rotate=True)
SVG(image)
```

[16]:

[17]:
```
# cut with 3 clusters
labels = cut_straight(dendrogram, n_clusters = 3)
print(labels)
```

```
[0 0 1 1 0 2 0 0 0 2 0 1 0 0]
```

[18]:
```
image = svg_digraph(adjacency, position, names=names, labels=labels)
SVG(image)
```

[18]:

[19]:
```
# metrics
dasgupta_score(adjacency, dendrogram)
```

[19]:
```
0.31285714285714294
```

Bipartite graphs

[20]:
```
graph = movie_actor(metadata=True)
biadjacency = graph.biadjacency
names_row = graph.names_row
names_col = graph.names_col
```

[21]:
```
# hierarchical clustering
ward = Ward(co_cluster = True)
ward.fit(biadjacency)
```

[21]:
```
Ward(embedding_method=Spectral(n_components=10, decomposition='rw', regularization=-1,␣
↪normalized=True), co_cluster=True)
```

[22]:
```
dendrogram_row = ward.dendrogram_row_
dendrogram_col = ward.dendrogram_col_
dendrogram_full = ward.dendrogram_full_
```

[23]:
```
image = svg_dendrogram(dendrogram_row, names_row, n_clusters=4, rotate=True)
SVG(image)
```

[23]:

[24]:
```
image = svg_dendrogram(dendrogram_col, names_col, n_clusters=4, rotate=True)
SVG(image)
```

[24]:

[25]:
```
# cuts
labels = cut_straight(dendrogram_full, n_clusters = 4)
n_row = biadjacency.shape[0]
labels_row = labels[:n_row]
labels_col = labels[n_row:]
```

[26]:
```
image = svg_bigraph(biadjacency, names_row, names_col, labels_row, labels_col)
SVG(image)
```

[26]:

3.22.3 Louvain

This notebook illustrates the hierarchical clustering of graphs by the Louvain hierarchical algorithm.

[1]:
```
from IPython.display import SVG
```

[2]:
```
import numpy as np
```

[3]:
```
from sknetwork.data import karate_club, painters, movie_actor
from sknetwork.hierarchy import LouvainHierarchy
from sknetwork.hierarchy import cut_straight, dasgupta_score, tree_sampling_divergence
from sknetwork.visualization import svg_graph, svg_digraph, svg_bigraph, svg_dendrogram
```

Graphs

[4]:
```
graph = karate_club(metadata=True)
adjacency = graph.adjacency
position = graph.position
```

[5]:
```
# hierarchical clustering
louvain_hierarchy = LouvainHierarchy()
dendrogram = louvain_hierarchy.fit_transform(adjacency)
```

[6]:
```
image = svg_dendrogram(dendrogram)
SVG(image)
```

[6]:

[7]:
```
# cuts
labels = cut_straight(dendrogram)
print(labels)
```

```
[0 0 0 0 3 3 3 0 1 0 3 0 0 0 1 1 3 0 1 0 1 0 1 2 2 2 1 2 2 1 1 2 1 1]
```

[8]:
```
labels, dendrogram_aggregate = cut_straight(dendrogram, n_clusters=4, return_
↪dendrogram=True)
print(labels)
```

```
[0 0 0 0 3 3 3 0 1 0 3 0 0 0 1 1 3 0 1 0 1 0 1 2 2 2 1 2 2 1 1 2 1 1]
```

[9]:
```
_, counts = np.unique(labels, return_counts=True)
```

[10]:
```
image = svg_dendrogram(dendrogram_aggregate, names=counts, rotate_names=False)
SVG(image)
```

[10]:

[11]:
```
image = svg_graph(adjacency, position, labels=labels)
SVG(image)
```

[11]:

[12]:
```
# metrics
dasgupta_score(adjacency, dendrogram)
```

[12]:
```
0.6293363499245852
```

Directed graphs

[13]:
```
graph = painters(metadata=True)
adjacency = graph.adjacency
position = graph.position
names = graph.names
```

[14]:
```
# hierarchical clustering
louvain_hierarchy = LouvainHierarchy()
dendrogram = louvain_hierarchy.fit_transform(adjacency)
```

[15]:
```
image = svg_dendrogram(dendrogram, names, rotate=True)
SVG(image)
```

[15]:

[16]:
```
# cut with 3 clusters
labels = cut_straight(dendrogram, n_clusters = 3)
print(labels)
```

```
[1 0 2 0 2 2 1 0 1 2 1 0 0 1]
```

[17]:
```
image = svg_digraph(adjacency, position, names=names, labels=labels)
SVG(image)
```

[17]:

[18]:
```
# metrics
dasgupta_score(adjacency, dendrogram)
```

[18]:
```
0.53
```

Bipartite graphs

[19]:
```
graph = movie_actor(metadata=True)
biadjacency = graph.biadjacency
names_row = graph.names_row
names_col = graph.names_col
```

[20]:
```
# hierarchical clustering
louvain_hierarchy = LouvainHierarchy()
louvain_hierarchy.fit(biadjacency)
dendrogram_row = louvain_hierarchy.dendrogram_row_
dendrogram_col = louvain_hierarchy.dendrogram_col_
dendrogram_full = louvain_hierarchy.dendrogram_full_
```

[21]:
```
image = svg_dendrogram(dendrogram_row, names_row, n_clusters=4, rotate=True)
SVG(image)
```

[21]:

[22]:
```
image = svg_dendrogram(dendrogram_col, names_col, n_clusters=4, rotate=True)
SVG(image)
```

[22]:

[23]:
```
# cuts
labels = cut_straight(dendrogram_full, n_clusters = 4)
n_row = biadjacency.shape[0]
labels_row = labels[:n_row]
labels_col = labels[n_row:]
```

[24]:
```
image = svg_bigraph(biadjacency, names_row, names_col, labels_row, labels_col)
SVG(image)
```

[24]:

3.23 Ranking

3.23.1 PageRank

This notebook illustrates the ranking of the nodes of a graph by PageRank.

[1]:
```
from IPython.display import SVG
```

[2]:
```
import numpy as np
```

[3]:
```
from sknetwork.data import karate_club, painters, movie_actor
from sknetwork.ranking import PageRank
from sknetwork.visualization import svg_graph, svg_digraph, svg_bigraph
```

Graphs

[4]:
```
graph = karate_club(metadata=True)
adjacency = graph.adjacency
position = graph.position
```

[5]:
```
# PageRank
pagerank = PageRank()
scores = pagerank.fit_transform(adjacency)
```

[6]:
```
image = svg_graph(adjacency, position, scores=np.log(scores))
SVG(image)
```

[6]:

[7]:
```
# personalized PageRank
seeds = {1: 1, 10: 1}
scores = pagerank.fit_transform(adjacency, seeds)
```

[8]:
```
image = svg_graph(adjacency, position, scores=np.log(scores), seeds=seeds)
SVG(image)
```

[8]:

Directed graphs

[9]:
```
graph = painters(metadata=True)
adjacency = graph.adjacency
names = graph.names
position = graph.position
```

[10]:
```
# PageRank
pagerank = PageRank()
scores = pagerank.fit_transform(adjacency)
```

[11]:
```
image = svg_digraph(adjacency, position, scores=np.log(scores), names=names)
SVG(image)
```

[11]:

[12]:
```
# personalized PageRank
cezanne = 11
seeds = {cezanne:1}
scores = pagerank.fit_transform(adjacency, seeds)
```

[13]:
```
image = svg_digraph(adjacency, position, names, scores=np.log(scores + 1e-6),␣
↪seeds=seeds)
SVG(image)
```

[13]:

Bipartite graphs

[14]:
```
graph = movie_actor(metadata=True)
biadjacency = graph.biadjacency
names_row = graph.names_row
names_col = graph.names_col
```

[15]:
```
pagerank = PageRank()
```

[16]:
```
drive = 3
aviator = 9
seeds_row={drive: 1, aviator: 1}
```

[17]:
```
pagerank.fit(biadjacency, seeds_row)
scores_row = pagerank.scores_row_
scores_col = pagerank.scores_col_
```

[18]:
```
image = svg_bigraph(biadjacency, names_row, names_col,
                    scores_row=np.log(scores_row), scores_col=np.log(scores_col), seeds_
↪row=seeds_row)
```

(continues on next page)

(continued from previous page)

```
SVG(image)
```

[18]:

3.23.2 Katz centrality

This notebook illustrates the ranking of the nodes of a graph by Katz centrality, a weighted average of number of paths of different lengths to each node.

[1]:
```
from IPython.display import SVG
```

[2]:
```
import numpy as np
```

[3]:
```
from sknetwork.data import karate_club, painters, movie_actor
from sknetwork.ranking import Katz
from sknetwork.visualization import svg_graph, svg_digraph, svg_bigraph
```

Graphs

[4]:
```
graph = karate_club(metadata=True)
adjacency = graph.adjacency
position = graph.position
```

[5]:
```
katz = Katz()
scores = katz.fit_transform(adjacency)
```

[6]:
```
image = svg_graph(adjacency, position, scores=scores)
SVG(image)
```

[6]:

Directed graphs

[7]:
```
graph = painters(metadata=True)
adjacency = graph.adjacency
names = graph.names
position = graph.position
```

[8]:
```
katz = Katz()
scores = katz.fit_transform(adjacency)
```

[9]:
```
image = svg_digraph(adjacency, position, scores=scores, names=names)
SVG(image)
```

[9]:

Bipartite graphs

[10]:
```
graph = movie_actor(metadata=True)
biadjacency = graph.biadjacency
names_row = graph.names_row
names_col = graph.names_col
```

[11]:
```
katz = Katz()
katz.fit(biadjacency)
scores_row = katz.scores_row_
scores_col = katz.scores_col_
```

[12]:
```
image = svg_bigraph(biadjacency, names_row, names_col, scores_row=scores_row, scores_
↪col=scores_col)
SVG(image)
```

[12]:

3.24 Classification

3.24.1 PageRank

This notebook illustrates the classification of the nodes of a graph by PageRank, based on the labels of a few nodes.

[1]:
```
from IPython.display import SVG
```

[2]:
```
import numpy as np
```

[3]:
```
from sknetwork.data import karate_club, painters, movie_actor
from sknetwork.classification import PageRankClassifier
from sknetwork.visualization import svg_graph, svg_digraph, svg_bigraph
```

Graphs

[4]:
```
graph = karate_club(metadata=True)
adjacency = graph.adjacency
position = graph.position
labels_true = graph.labels
```

[5]:
```
seeds = {i: labels_true[i] for i in [0, 33]}
```

[6]:
```
pagerank = PageRankClassifier()
labels_pred = pagerank.fit_transform(adjacency, seeds)
```

[7]:
```
precision = np.round(np.mean(labels_pred == labels_true), 2)
precision
```

[7]:
```
0.97
```

[8]:
```
image = svg_graph(adjacency, position, labels=labels_pred, seeds=seeds)
SVG(image)
```

[8]:

[9]:
```
# soft classification (here probability of label 1)
label = 1
membership = pagerank.membership_
scores = membership[:,label].toarray().ravel()
```

[10]:
```
image = svg_graph(adjacency, position, scores=scores, seeds=seeds)
SVG(image)
```

[10]:

Directed graphs

[11]:
```
graph = painters(metadata=True)
adjacency = graph.adjacency
position = graph.position
names = graph.names
```

[12]:
```
rembrandt = 5
klimt = 6
cezanne = 11
seeds = {cezanne: 0, rembrandt: 1, klimt: 2}
```

[13]:
```
pagerank = PageRankClassifier()
labels = pagerank.fit_transform(adjacency, seeds)
```

[14]:
```
image = svg_digraph(adjacency, position, names, labels=labels, seeds=seeds)
SVG(image)
```

[14]:

[15]:
```
# soft classification
membership = pagerank.membership_
scores = membership[:,0].toarray().ravel()
```

[16]:
```
image = svg_digraph(adjacency, position, names, scores=scores, seeds=[cezanne])
SVG(image)
```

[16]:

Bipartite graphs

[17]:
```
graph = movie_actor(metadata=True)
biadjacency = graph.biadjacency
names_row = graph.names_row
names_col = graph.names_col
```

[18]:
```
inception = 0
drive = 3
budapest = 8
```

[19]:
```
seeds_row = {inception: 0, drive: 1, budapest: 2}
```

[20]:
```
pagerank = PageRankClassifier()
pagerank.fit(biadjacency, seeds_row)
labels_row = pagerank.labels_row_
labels_col = pagerank.labels_col_
```

[21]:
```
image = svg_bigraph(biadjacency, names_row, names_col, labels_row, labels_col, seeds_
↪row=seeds_row)
SVG(image)
```

[21]:

[22]:
```
# soft classification
membership_row = pagerank.membership_row_
membership_col = pagerank.membership_col_
```

[23]:
```
label = 1
scores_row = membership_row[:,label].toarray().ravel()
scores_col = membership_col[:,label].toarray().ravel()
```

[24]:
```
image = svg_bigraph(biadjacency, names_row, names_col, scores_row=scores_row, scores_
↪col=scores_col,
                    seeds_row=seeds_row)
SVG(image)
```

[24]:

3.24.2 Diffusion

This notebook illustrates the classification of the nodes of a graph by diffusion, based on the labels of a few nodes.

[1]:
```
from IPython.display import SVG
```

[2]:
```
import numpy as np
```

[3]:
```
from sknetwork.data import karate_club, painters, movie_actor
from sknetwork.classification import DiffusionClassifier
from sknetwork.visualization import svg_graph, svg_digraph, svg_bigraph
```

Graphs

[4]:
```
graph = karate_club(metadata=True)
adjacency = graph.adjacency
position = graph.position
labels_true = graph.labels
```

[5]:
```
seeds = {i: labels_true[i] for i in [0, 33]}
```

[6]:
```
diffusion = DiffusionClassifier()
labels_pred = diffusion.fit_transform(adjacency, seeds)
```

[7]:
```
precision = np.round(np.mean(labels_pred == labels_true), 2)
precision
```

[7]:
```
0.94
```

[8]:
```
image = svg_graph(adjacency, position, labels=labels_pred, seeds=seeds)
SVG(image)
```

[8]:

[9]:
```
# soft classification (here probability of label 1)
scores = diffusion.score(label=1)
```

[10]:
```
image = svg_graph(adjacency, position, scores=scores, seeds=seeds)
SVG(image)
```

[10]:

Directed graphs

[11]:
```
graph = painters(metadata=True)
adjacency = graph.adjacency
position = graph.position
names = graph.names
```

[12]:
```
rembrandt = 5
cezanne = 11
seeds = {cezanne: 0, rembrandt: 1}
```

[13]:
```
diffusion = DiffusionClassifier()
labels = diffusion.fit_transform(adjacency, seeds)
```

[14]:
```
image = svg_digraph(adjacency, position, names, labels=labels, seeds=seeds)
SVG(image)
```

[14]:

[15]:
```
# soft classification (here probability of label 0)
scores = diffusion.score(label=0)
```

[16]:
```
image = svg_digraph(adjacency, position, names=names, scores=scores, seeds=[cezanne])
SVG(image)
```

[16]:

Bipartite graphs

[17]:
```
graph = movie_actor(metadata=True)
biadjacency = graph.biadjacency
names_row = graph.names_row
names_col = graph.names_col
```

[18]:
```
inception = 0
drive = 3
```

[19]:
```
seeds_row = {inception: 0, drive: 1}
```

[20]:
```
diffusion = DiffusionClassifier()
diffusion.fit(biadjacency, seeds_row)
labels_row = diffusion.labels_row_
labels_col = diffusion.labels_col_
```

[21]:
```
image = svg_bigraph(biadjacency, names_row, names_col, labels_row, labels_col, seeds_
↪row=seeds_row)
SVG(image)
```

[21]:

[22]:
```
# soft classification
membership_row = diffusion.membership_row_
membership_col = diffusion.membership_col_
```

[23]:
```
# probability of label 1
scores_row = membership_row[:,1].toarray().ravel()
scores_col = membership_col[:,1].toarray().ravel()
```

[24]:
```
image = svg_bigraph(biadjacency, names_row, names_col, scores_row=scores_row, scores_
↪col=scores_col,
                    seeds_row=seeds_row)
SVG(image)
```

[24]:

3.24.3 Dirichlet

This notebook illustrates the classification of the nodes of a graph by the Dirichlet problem, based on the labels of a few nodes.

[1]:
```
from IPython.display import SVG
```

[2]:
```
import numpy as np
```

[3]:
```
from sknetwork.data import karate_club, painters, movie_actor
from sknetwork.classification import DirichletClassifier
from sknetwork.visualization import svg_graph, svg_digraph, svg_bigraph
```

Graphs

[4]:
```
graph = karate_club(metadata=True)
adjacency = graph.adjacency
position = graph.position
labels_true = graph.labels
```

[5]:
```
seeds = {i: labels_true[i] for i in [0, 33]}
```

[6]:
```
dirichlet = DirichletClassifier()
labels_pred = dirichlet.fit_transform(adjacency, seeds)
```

[7]:
```
precision = np.round(np.mean(labels_pred == labels_true), 2)
precision
```

[7]:
```
0.97
```

[8]:
```
image = svg_graph(adjacency, position, labels=labels_pred, seeds=seeds)
SVG(image)
```

[8]:

[9]:
```
# soft classification (here probability of label 1)
membership = dirichlet.membership_
scores = membership[:,1].toarray().ravel()
```

[10]:
```
image = svg_graph(adjacency, position, scores=scores, seeds=seeds)
SVG(image)
```

[10]:

Directed graphs

[11]:
```
graph = painters(metadata=True)
adjacency = graph.adjacency
position = graph.position
names = graph.names
```

[12]:
```
rembrandt = 5
klimt = 6
cezanne = 11
seeds = {cezanne: 0, rembrandt: 1, klimt: 2}
```

[13]:
```
dirichlet = DirichletClassifier()
labels = dirichlet.fit_transform(adjacency, seeds)
```

[14]:
```
image = svg_digraph(adjacency, position, names, labels=labels, seeds=seeds)
SVG(image)
```

[14]:

[15]:
```
# soft classification (here probability of label 0)
membership = dirichlet.membership_
scores = membership[:,0].toarray().ravel()
```

[16]:
```
image = svg_digraph(adjacency, position, names=names, scores=scores, seeds=[cezanne])
SVG(image)
```

[16]:

Bipartite graphs

[17]:
```
graph = movie_actor(metadata=True)
biadjacency = graph.biadjacency
names_row = graph.names_row
names_col = graph.names_col
```

[18]:
```
inception = 0
drive = 3
budapest = 8
```

[19]:
```
seeds_row = {inception: 0, drive: 1, budapest: 2}
```

[20]:
```
dirichlet = DirichletClassifier()
dirichlet.fit(biadjacency, seeds_row)
labels_row = dirichlet.labels_row_
labels_col = dirichlet.labels_col_
```

[21]:
```
image = svg_bigraph(biadjacency, names_row, names_col, labels_row, labels_col, seeds_
↪row=seeds_row)
SVG(image)
```

[21]:

[22]:
```
# soft classification (here probability of label 1)
membership_row = dirichlet.membership_row_
membership_col = dirichlet.membership_col_
```

[23]:
```
scores_row = membership_row[:,1].toarray().ravel()
scores_col = membership_col[:,1].toarray().ravel()
```

[24]:
```
image = svg_bigraph(biadjacency, names_row, names_col, scores_row=scores_row, scores_
↪col=scores_col,
                    seeds_row=seeds_row)
SVG(image)
```

[24]:

3.24.4 Propagation

This notebook illustrates the classification of the nodes of a graph by label propagation.

[1]:
```
from IPython.display import SVG
```

[2]:
```
import numpy as np
```

[3]:
```
from sknetwork.data import karate_club, painters, movie_actor
from sknetwork.classification import Propagation
from sknetwork.visualization import svg_graph, svg_digraph, svg_bigraph
```

Graphs

[4]:
```
graph = karate_club(metadata=True)
adjacency = graph.adjacency
position = graph.position
labels_true = graph.labels
```

[5]:
```
seeds = {i: labels_true[i] for i in [0, 33]}
```

[6]:
```
propagation = Propagation()
labels_pred = propagation.fit_transform(adjacency, seeds)
```

[7]:
```
image = svg_graph(adjacency, position, labels=labels_pred, seeds=seeds)
SVG(image)
```

[7]:

[8]:
```
# soft classification
label = 1
membership = propagation.membership_
scores = membership[:,label].toarray().ravel()
```

[9]:
```
image = svg_graph(adjacency, position, scores=scores, seeds=seeds)
SVG(image)
```

[9]:

Directed graphs

[10]:
```
graph = painters(metadata=True)
adjacency = graph.adjacency
position = graph.position
names = graph.names
```

[11]:
```
rembrandt = 5
klimt = 6
cezanne = 11
seeds = {cezanne: 0, rembrandt: 1, klimt: 2}
```

[12]:
```
propagation = Propagation()
labels = propagation.fit_transform(adjacency, seeds)
```

[13]:
```
image = svg_digraph(adjacency, position, names, labels=labels, seeds=seeds)
SVG(image)
```

[13]:

[14]:
```
# soft classification
membership = propagation.membership_
scores = membership[:,0].toarray().ravel()
```

[15]:
```
image = svg_digraph(adjacency, position, names, scores=scores, seeds=[cezanne])
SVG(image)
```

[15]:

Bipartite graphs

[16]:
```
graph = movie_actor(metadata=True)
biadjacency = graph.biadjacency
names_row = graph.names_row
names_col = graph.names_col
```

[17]:
```
inception = 0
drive = 3
budapest = 8
```

[18]:
```
seeds_row = {inception: 0, drive: 1, budapest: 2}
```

[19]:
```
propagation = Propagation()
labels_row = propagation.fit_transform(biadjacency, seeds_row)
labels_col = propagation.labels_col_
```

[20]:
```
image = svg_bigraph(biadjacency, names_row, names_col, labels_row, labels_col, seeds_
↪row=seeds_row)
SVG(image)
```

[20]:

[21]:
```
# soft classification
membership_row = propagation.membership_row_
membership_col = propagation.membership_col_
```

[22]:
```
scores_row = membership_row[:,1].toarray().ravel()
scores_col = membership_col[:,1].toarray().ravel()
```

[23]:
```
image = svg_bigraph(biadjacency, names_row, names_col, scores_row=scores_row, scores_
↪col=scores_col,
                    seeds_row=seeds_row)
SVG(image)
```

[23]:

3.24.5 Nearest neighbors

This notebook illustrates the classification of the nodes of a graph by the k-nearest neighbors algorithm, based on the labels of a few nodes.

[1]:
```
from IPython.display import SVG
```

[2]:
```
import numpy as np
```

[3]:
```
from sknetwork.data import karate_club, painters, movie_actor
from sknetwork.classification import KNN
from sknetwork.embedding import GSVD
from sknetwork.visualization import svg_graph, svg_digraph, svg_bigraph
```

Graphs

[4]:
```
graph = karate_club(metadata=True)
adjacency = graph.adjacency
position = graph.position
labels_true = graph.labels
```

[5]:
```
seeds = {i: labels_true[i] for i in [0, 33]}
```

[6]:
```
knn = KNN(GSVD(3), n_neighbors=1)
labels_pred = knn.fit_transform(adjacency, seeds)
```

[7]:
```
precision = np.round(np.mean(labels_pred == labels_true), 2)
precision
```

[7]:
```
0.97
```

[8]:
```
image = svg_graph(adjacency, position, labels=labels_pred, seeds=seeds)
SVG(image)
```

[8]:

[9]:
```
# soft classification (here probability of label 1)
knn = KNN(GSVD(3), n_neighbors=2)
knn.fit(adjacency, seeds)
membership = knn.membership_
```

[10]:
```
scores = membership[:,1].toarray().ravel()
```

[11]:
```
image = svg_graph(adjacency, position, scores=scores, seeds=seeds)
SVG(image)
```

[11]:

Directed graphs

[12]:
```
graph = painters(metadata=True)
adjacency = graph.adjacency
position = graph.position
names = graph.names
```

[13]:
```
rembrandt = 5
klimt = 6
cezanne = 11
seeds = {cezanne: 0, rembrandt: 1, klimt: 2}
```

[14]:
```
knn = KNN(GSVD(3), n_neighbors=2)
labels = knn.fit_transform(adjacency, seeds)
```

[15]:
```
image = svg_digraph(adjacency, position, names, labels=labels, seeds=seeds)
SVG(image)
```

[15]:

[16]:
```
# soft classification
membership = knn.membership_
scores = membership[:,0].toarray().ravel()
```

[17]:
```
image = svg_digraph(adjacency, position, names, scores=scores, seeds=[cezanne])
SVG(image)
```

[17]:

Bipartite graphs

[18]:
```
graph = movie_actor(metadata=True)
biadjacency = graph.biadjacency
names_row = graph.names_row
names_col = graph.names_col
```

[19]:
```
inception = 0
drive = 3
budapest = 8
```

[20]:
```
seeds_row = {inception: 0, drive: 1, budapest: 2}
```

[21]:
```
knn = KNN(GSVD(3), n_neighbors=2)
labels_row = knn.fit_transform(biadjacency, seeds_row)
labels_col = knn.labels_col_
```

[22]:
```
image = svg_bigraph(biadjacency, names_row, names_col, labels_row, labels_col, seeds_
↪row=seeds_row)
SVG(image)
```

[22]:

[23]:
```
# soft classification
membership_row = knn.membership_row_
membership_col = knn.membership_col_
```

[24]:
```
scores_row = membership_row[:,1].toarray().ravel()
scores_col = membership_col[:,1].toarray().ravel()
```

[25]:
```
image = svg_bigraph(biadjacency, names_row, names_col, scores_row=scores_row, scores_
↪col=scores_col,
                    seeds_row=seeds_row)
SVG(image)
```

[25]:

3.25 Regression

3.25.1 Diffusion

This notebook illustrates a regression task by heat diffusion.

[1]:
```
from IPython.display import SVG
```

[2]:
```
import numpy as np
```

[3]:
```
from sknetwork.data import karate_club, painters, movie_actor
from sknetwork.regression import Diffusion
from sknetwork.visualization import svg_graph, svg_digraph, svg_bigraph
```

Graphs

[4]:
```
graph = karate_club(metadata=True)
adjacency = graph.adjacency
position = graph.position
labels_true = graph.labels
```

[5]:
```
# heat diffusion
diffusion = Diffusion()
seeds = {0: 0, 33: 1}
values = diffusion.fit_transform(adjacency, seeds)
```

[6]:
```
image = svg_graph(adjacency, position, scores=values, seeds=seeds)
SVG(image)
```

[6]:

Directed graphs

[7]:
```
graph = painters(metadata=True)
adjacency = graph.adjacency
position = graph.position
names = graph.names
```

[8]:
```
picasso = 0
manet = 3
```

[9]:
```
diffusion = Diffusion()
seeds = {picasso: 1, manet: 1}
values = diffusion.fit_transform(adjacency, seeds, init=0)
```

[10]:
```
image = svg_digraph(adjacency, position, names, scores=values, seeds=seeds)
SVG(image)
```

[10]:

Bipartite graphs

[11]:
```
graph = movie_actor(metadata=True)
biadjacency = graph.biadjacency
names_row = graph.names_row
names_col = graph.names_col
```

[12]:
```
drive = 3
aviator = 9
```

[13]:
```
diffusion = Diffusion()
seeds_row = {drive: 0, aviator: 1}
diffusion.fit(biadjacency, seeds_row=seeds_row)
values_row = diffusion.values_row_
values_col = diffusion.values_col_
```

[14]:
```
image = svg_bigraph(biadjacency, names_row, names_col, scores_row=values_row, scores_
↪col=values_col,
                    seeds_row=seeds_row)
SVG(image)
```

[14]:

Since seeds are on movies, you need an even number of iterations to get non-trivial ranking of movies. This is due to the bipartite structure of the graph.

[15]:
```
# changing the number of iterations
diffusion = Diffusion(n_iter=4)
seeds_row = {drive: 0, aviator: 1}
diffusion.fit(biadjacency, seeds_row=seeds_row)
values_row = diffusion.values_row_
values_col = diffusion.values_col_
```

[16]:
```
image = svg_bigraph(biadjacency, names_row, names_col, scores_row=values_row, scores_
↪col=values_col,
                    seeds_row=seeds_row)
SVG(image)
```

[16]:

3.25.2 Dirichlet

This notebook illustrates a regression task as a solution of the Dirichlet problem (heat diffusion with constraints).

[1]:
```
from IPython.display import SVG
```

[2]:
```
import numpy as np
```

[3]:
```
from sknetwork.data import karate_club, painters, movie_actor
from sknetwork.regression import Dirichlet
from sknetwork.visualization import svg_graph, svg_digraph, svg_bigraph
```

Graphs

[4]:
```
graph = karate_club(metadata=True)
adjacency = graph.adjacency
position = graph.position
labels_true = graph.labels
```

[5]:
```
# heat diffusion
dirichlet = Dirichlet()
seeds = {0: 0, 33: 1}
values = dirichlet.fit_transform(adjacency, seeds)
```

[6]:
```
image = svg_graph(adjacency, position, scores=values, seeds=seeds)
SVG(image)
```

[6]:

Directed graphs

[7]:
```
graph = painters(metadata=True)
adjacency = graph.adjacency
position = graph.position
names = graph.names
```

[8]:
```
picasso = 0
monet = 1
```

[9]:
```
dirichlet = Dirichlet()
seeds = {picasso: 0, monet: 1}
values = dirichlet.fit_transform(adjacency, seeds)
```

[10]:
```
image = svg_digraph(adjacency, position, names, scores=values, seeds=seeds)
SVG(image)
```

[10]:

Bipartite graphs

[11]:
```
graph = movie_actor(metadata=True)
biadjacency = graph.biadjacency
names_row = graph.names_row
names_col = graph.names_col
```

[12]:
```
dirichlet = Dirichlet()
```

[13]:
```
drive = 3
aviator = 9
```

[14]:
```
seeds_row = {drive: 0, aviator: 1}
dirichlet.fit(biadjacency, seeds_row)
values_row = dirichlet.values_row_
values_col = dirichlet.values_col_
```

[15]:
```
image = svg_bigraph(biadjacency, names_row, names_col, scores_row=values_row, scores_
↪col=values_col,
                    seeds_row=seeds_row)
SVG(image)
```

[15]:

3.26 Embedding

3.26.1 Spectral

This notebook illustrates the spectral embedding of a graph.

[1]:
```
from IPython.display import SVG
```

[2]:
```
import numpy as np
from scipy import sparse
```

[3]:
```
from sknetwork.data import karate_club, painters, movie_actor
from sknetwork.embedding import Spectral
from sknetwork.visualization import svg_graph, svg_digraph, svg_bigraph
```

Graphs

[4]:
```
graph = karate_club(metadata=True)
adjacency = graph.adjacency
labels = graph.labels
```

[5]:
```
# embedding in dimension 2
spectral = Spectral(2)
embedding = spectral.fit_transform(adjacency)
```

[6]:
```
# visualization
image = svg_graph(adjacency, position=embedding, labels=labels)
SVG(image)
```

[6]:

[7]:
```
# find the embedding of a new node
adjacency_vector = np.zeros(adjacency.shape[0], dtype = int)
adjacency_vector[:6] = np.ones(6, dtype = int)
embedding_vector = spectral.predict(adjacency_vector)
```

[8]:
```
# visualization
adjacency_extend = sparse.vstack([adjacency, adjacency_vector])
adjacency_extend = sparse.hstack([adjacency_extend, sparse.csr_matrix((35, 1))], format=
↪'csr')
embedding_extend = np.vstack([embedding, embedding_vector])
labels_extend = list(labels) + [-1]
```

[9]:
```
image = svg_graph(adjacency_extend, position=embedding_extend, labels=labels_extend,␣
↪seeds={34:1})
SVG(image)
```

[9]:

Directed graphs

[10]:
```
graph = painters(metadata=True)
adjacency = graph.adjacency
position = graph.position
names = graph.names
```

[11]:
```
# embedding
spectral = Spectral()
embedding = spectral.fit_transform(adjacency)
```

[12]:
```
image = svg_digraph(adjacency, position=embedding, names=names)
SVG(image)
```

[12]:

Bipartite graphs

[13]:
```
graph = movie_actor(metadata=True)
biadjacency = graph.biadjacency
names_row = graph.names_row
names_col = graph.names_col
```

[14]:
```
# embedding
spectral = Spectral()
spectral.fit(biadjacency)
```

[14]:
```
Spectral(n_components=2, decomposition='rw', regularization=-1, normalized=True)
```

[15]:
```
embedding_row = spectral.embedding_row_
embedding_col = spectral.embedding_col_
```

[16]:
```
image = svg_bigraph(biadjacency, names_row, names_col,
                    position_row=embedding_row, position_col=embedding_col,
                    color_row='blue', color_col='red')
SVG(image)
```

[16]:

3.26.2 SVD

This notebook illustrates the embedding of a graph through the singular value decomposition of the adjacency matrix.

[1]:
```
from IPython.display import SVG
```

[2]:
```
import numpy as np
```

[3]:
```
from sknetwork.data import karate_club, painters, movie_actor
from sknetwork.embedding import SVD, cosine_modularity
from sknetwork.visualization import svg_graph, svg_digraph, svg_bigraph
```

Graphs

[4]:
```
graph = karate_club(metadata=True)
adjacency = graph.adjacency
labels = graph.labels
```

[5]:
```
svd = SVD(2)
embedding = svd.fit_transform(adjacency)
```

[6]:
```
image = svg_graph(adjacency, embedding, labels=labels)
SVG(image)
```

[6]:

Directed graphs

[7]:
```
graph = painters(metadata=True)
adjacency = graph.adjacency
names = graph.names
```

[8]:
```
svd = SVD(2)
embedding = svd.fit_transform(adjacency)
```

[9]:
```
image = svg_digraph(adjacency, embedding, names=names)
SVG(image)
```

[9]:

Bipartite graphs

[10]:
```
graph = movie_actor(metadata=True)
biadjacency = graph.biadjacency
names_row = graph.names_row
names_col = graph.names_col
```

[11]:
```
svd = SVD(2, normalized=False)
svd.fit(biadjacency)
```

[11]:
```
SVD(n_components=2, regularization=None, factor_singular=0.0, normalized=False,␣
↪solver=LanczosSVD(n_iter=None, tol=0.0))
```

[12]:
```
embedding_row = svd.embedding_row_
embedding_col = svd.embedding_col_
```

[13]:
```
image = svg_bigraph(biadjacency, names_row, names_col,
                    position_row=embedding_row, position_col=embedding_col,
                    color_row='blue', color_col='red', scale=1.5)
SVG(image)
```

[13]:

3.26.3 GSVD

This notebook illustrates the embedding of a graph through the generalized singular value decomposition of the adjacency matrix.

[1]:
```
from IPython.display import SVG
```

[2]:
```
import numpy as np
```

[3]:
```
from sknetwork.data import karate_club, painters, movie_actor
from sknetwork.embedding import GSVD
from sknetwork.visualization import svg_graph, svg_digraph, svg_bigraph
```

Graphs

[4]:
```
graph = karate_club(metadata=True)
adjacency = graph.adjacency
labels = graph.labels
```

[5]:
```
gsvd = GSVD(2, normalized=False)
embedding = gsvd.fit_transform(adjacency)
```

[6]:
```
image = svg_graph(adjacency, embedding, labels=labels)
SVG(image)
```

[6]:

Directed graphs

[7]:
```
graph = painters(metadata=True)
adjacency = graph.adjacency
position = graph.position
names = graph.names
```

[8]:
```
gsvd = GSVD(2, normalized=False)
embedding = gsvd.fit_transform(adjacency)
```

[9]:
```
image = svg_digraph(adjacency, embedding, names=names)
SVG(image)
```

[9]:

Bipartite graphs

[10]:
```
graph = movie_actor(metadata=True)
biadjacency = graph.biadjacency
names_row = graph.names_row
names_col = graph.names_col
```

[11]:
```
gsvd = GSVD(2, normalized=False)
gsvd.fit(biadjacency)
```

[11]:
```
GSVD(n_components=2, regularization=None, factor_row=0.5, factor_col=0.5, factor_
↪singular=0.0, normalized=False, solver=LanczosSVD(n_iter=None, tol=0.0))
```

[12]:
```
embedding_row = gsvd.embedding_row_
embedding_col = gsvd.embedding_col_
```

[13]:
```
image = svg_bigraph(biadjacency, names_row, names_col,
                    position_row=embedding_row, position_col=embedding_col,
                    color_row='blue', color_col='red')
SVG(image)
```

[13]:

3.26.4 PCA

This notebook illustrates the embedding of a graph through the principal component analysis of the adjacency matrix.

[1]:
```
from IPython.display import SVG
```

[2]:
```
import numpy as np
```

[3]:
```
from sknetwork.data import karate_club, painters, movie_actor
from sknetwork.embedding import PCA
from sknetwork.visualization import svg_graph, svg_digraph, svg_bigraph
```

Graphs

[4]:
```
graph = karate_club(metadata=True)
adjacency = graph.adjacency
labels = graph.labels
```

[5]:
```
pca = PCA(2)
embedding = pca.fit_transform(adjacency)
```

[6]:
```
image = svg_graph(adjacency, embedding, labels=labels)
SVG(image)
```

[6]:

Directed graphs

[7]:
```
graph = painters(metadata=True)
adjacency = graph.adjacency
names = graph.names
```

[8]:
```
pca = PCA(2)
embedding = pca.fit_transform(adjacency)
```

[9]:
```
image = svg_digraph(adjacency, embedding, names=names)
SVG(image)
```

[9]:

Bipartite graphs

[10]:
```
graph = movie_actor(metadata=True)
biadjacency = graph.biadjacency
names_row = graph.names_row
names_col = graph.names_col
```

[11]:
```
pca = PCA(2)
pca.fit(biadjacency)
```

[11]:
```
PCA(n_components=2, normalized=False, solver=LanczosSVD(n_iter=None, tol=0.0))
```

[12]:
```
embedding_row = pca.embedding_row_
embedding_col = pca.embedding_col_
```

[13]:
```
image = svg_bigraph(biadjacency, names_row, names_col,
                    position_row=embedding_row, position_col=embedding_col,
                    color_row='blue', color_col='red')
SVG(image)
```

[13]:

3.26.5 Random Projection

This notebook illustrates the embedding of a graph through random projection.

[1]:
```
from IPython.display import SVG
```

[2]:
```
import numpy as np
```

[3]:
```
from sknetwork.data import karate_club, painters, movie_actor
from sknetwork.embedding import RandomProjection
from sknetwork.visualization import svg_graph, svg_digraph, svg_bigraph
```

Graphs

[4]:
```
graph = karate_club(metadata=True)
adjacency = graph.adjacency
labels = graph.labels
```

[5]:
```
projection = RandomProjection(2)
embedding = projection.fit_transform(adjacency)
```

[6]:
```
image = svg_graph(adjacency, position=embedding, labels=labels)
SVG(image)
```

[6]:

Directed graphs

[7]:
```
graph = painters(metadata=True)
adjacency = graph.adjacency
position = graph.position
names = graph.names
```

[8]:
```
projection = RandomProjection(2)
embedding = projection.fit_transform(adjacency)
```

[9]:
```
image = svg_digraph(adjacency, position=embedding, names=names)
SVG(image)
```

[9]:

Bipartite graphs

[10]:
```
graph = movie_actor(metadata=True)
biadjacency = graph.biadjacency
names_row = graph.names_row
names_col = graph.names_col
```

[11]:
```
projection = RandomProjection(2, normalized=False)
projection.fit(biadjacency)
```

[11]:
```
RandomProjection(n_components=2, alpha=0.5, n_iter=3, random_walk=False, regularization=-
↪1, normalized=False)
```

[12]:
```
embedding_row = projection.embedding_row_
embedding_col = projection.embedding_col_
```

[13]:
```
image = svg_bigraph(biadjacency, names_row, names_col,
                    position_row=embedding_row, position_col=embedding_col,
                    color_row='blue', color_col='red')
SVG(image)
```

[13]:

3.26.6 Louvain

This notebook illustrates the embedding of a graph through Louvain clustering.

[1]:
```
from IPython.display import SVG
```

[2]:
```
import numpy as np
```

[3]:
```
from sknetwork.data import karate_club, painters, movie_actor
from sknetwork.embedding import LouvainEmbedding
from sknetwork.visualization import svg_graph, svg_digraph, svg_bigraph
```

Graphs

[4]:
```
graph = karate_club(metadata=True)
adjacency = graph.adjacency
labels = graph.labels
```

[5]:
```
louvain = LouvainEmbedding()
embedding = louvain.fit_transform(adjacency)
embedding.shape
```

[5]:
```
(34, 4)
```

[6]:
```
position = embedding[:, :2]
```

[7]:
```
image = svg_graph(adjacency, position=position, labels=labels)
```

[8]:
```
SVG(image)
```

[8]:

Directed graphs

[9]:
```
graph = painters(metadata=True)
adjacency = graph.adjacency
names = graph.names
```

[10]:
```
louvain = LouvainEmbedding()
embedding = louvain.fit_transform(adjacency)
embedding.shape
```

[10]:
```
(14, 3)
```

[11]:
```
position = embedding[:, :2]
```

[12]:
```
image = svg_digraph(adjacency, position=position, names=names)
SVG(image)
```

[12]:

Bipartite graphs

[13]:
```
graph = movie_actor(metadata=True)
biadjacency = graph.biadjacency
names_row = graph.names_row
names_col = graph.names_col
```

[14]:
```
louvain = LouvainEmbedding()
louvain.fit(biadjacency)
```

[14]:
```
LouvainEmbedding(resolution=1, modularity='dugue', tol_optimization=0.001, tol_
↪aggregation=0.001, n_aggregations=-1, shuffle_nodes=False, isolated_nodes='remove')
```

[15]:
```
embedding_row = louvain.embedding_row_
embedding_col = louvain.embedding_col_
```

[16]:
```
position_row = embedding_row[:, :2]
position_col = embedding_col[:, :2]
```

[17]:
```
image = svg_bigraph(biadjacency, names_row, names_col,
                    position_row=position_row, position_col=position_col,
                    color_row='blue', color_col='red')
SVG(image)
```

[17]:

3.26.7 Louvain Hierarchy

This notebook illustrates the embedding of a graph by the hierarchical Louvain algorithm.

[1]:
```
from IPython.display import SVG
```

[2]:
```
import numpy as np
```

[3]:
```
from sknetwork.data import karate_club, painters, movie_actor
from sknetwork.embedding import LouvainNE
from sknetwork.visualization import svg_graph, svg_digraph, svg_bigraph
```

Graphs

[4]:
```
graph = karate_club(metadata=True)
adjacency = graph.adjacency
labels = graph.labels
```

[5]:
```
louvain = LouvainNE(2)
embedding = louvain.fit_transform(adjacency)
```

[6]:
```
image = svg_graph(adjacency, position=embedding, labels=labels)
SVG(image)
```

[6]:

Directed graphs

[7]:
```
graph = painters(metadata=True)
adjacency = graph.adjacency
position = graph.position
names = graph.names
```

[8]:
```
louvain = LouvainNE(2)
embedding = louvain.fit_transform(adjacency)
```

[9]:
```
image = svg_digraph(adjacency, position=embedding, names=names)
SVG(image)
```

[9]:

Bipartite graphs

[10]:
```
graph = movie_actor(metadata=True)
biadjacency = graph.biadjacency
names_row = graph.names_row
names_col = graph.names_col
```

[11]:
```
louvain = LouvainNE()
louvain.fit(biadjacency)
```

[11]:
```
LouvainNE(n_components=2, scale=0.1)
```

[12]:
```
embedding_row = louvain.embedding_row_
embedding_col = louvain.embedding_col_
```

[13]:
```
image = svg_bigraph(biadjacency, names_row, names_col,
                    position_row=embedding_row, position_col=embedding_col,
                    color_row='blue', color_col='red')
SVG(image)
```

[13]:

3.26.8 Spring

This notebook illustrates the 2D embedding of a graph through the force-directed algorithm.

[1]:
```
from IPython.display import SVG
```

[2]:
```
import numpy as np
```

[3]:
```
from sknetwork.data import karate_club, painters
from sknetwork.embedding import Spring
from sknetwork.visualization import svg_graph, svg_digraph
```

Graphs

[4]:
```
graph = karate_club(metadata=True)
adjacency = graph.adjacency
labels = graph.labels
```

[5]:
```
spring = Spring(2)
embedding = spring.fit_transform(adjacency)
```

[6]:
```
image = svg_graph(adjacency, position=embedding, labels=labels)
SVG(image)
```

[6]:

Directed graphs

[7]:
```
graph = painters(metadata=True)
adjacency = graph.adjacency
names = graph.names
```

[8]:
```
spring = Spring(2)
embedding = spring.fit_transform(adjacency)
embedding.shape
```

[8]:
```
(14, 2)
```

[9]:
```
image = svg_digraph(adjacency, position=embedding, names=names)
SVG(image)
```

[9]:

3.26.9 ForceAtlas

This notebook illustrates the embedding of a graph through the force-directed algorithm Force Atlas 2.

[1]:
```
from IPython.display import SVG
```

[2]:
```
from sknetwork.data import karate_club
from sknetwork.embedding.force_atlas import ForceAtlas
from sknetwork.visualization import svg_graph
```

[3]:
```
graph = karate_club(metadata=True)
adjacency = graph.adjacency
labels = graph.labels
```

[4]:
```
forceatlas2 = ForceAtlas()
embedding = forceatlas2.fit_transform(adjacency)
image = svg_graph(adjacency, embedding, labels=labels)
SVG(image)
```

[4]:

Here we illustrate the influences of the different settings offered to the user.

Replace the linear attraction force with a logarithmic attraction force.

[5]:
```
forceatlas2 = ForceAtlas(lin_log = True)
embedding = forceatlas2.fit_transform(adjacency)
image = svg_graph(adjacency, embedding, labels=labels)
SVG(image)
```

[5]:

Set the gravity and repulsion force constants (`gravity_factor` and `repulsion_factor`) to set the importance of each force in the layout. Keep values between 0.01 and 0.1.

[6]:
```
forceatlas2 = ForceAtlas(gravity_factor = 0.1)
embedding = forceatlas2.fit_transform(adjacency)
image = svg_graph(adjacency, embedding, labels=labels)
SVG(image)
```

[6]:

Set the amount of swinging tolerated. Lower swinging yields less speed and more precision.

[7]:
```
forceatlas2 = ForceAtlas(tolerance=1.5)
embedding = forceatlas2.fit_transform(adjacency)
image = svg_graph(adjacency, embedding, labels=labels)
SVG(image)
```

[7]:

[8]:
```
forceatlas2 = ForceAtlas(approx_radius=2)
embedding = forceatlas2.fit_transform(adjacency)
```

(continues on next page)

(continued from previous page)

```
image = svg_graph(adjacency, embedding, labels=labels)
SVG(image)
```

[8]:

3.27 Link prediction

3.27.1 First-order methods

[1]:
```
from numpy import argsort

from sknetwork.classification import accuracy_score
from sknetwork.data import karate_club, painters, movie_actor
from sknetwork.linkpred import JaccardIndex, AdamicAdar, is_edge, whitened_sigmoid
```

Adamic-Adar

[2]:
```
adjacency = karate_club()
```

[3]:
```
aa = AdamicAdar()
aa.fit(adjacency)
```

[3]:
```
AdamicAdar()
```

[4]:
```
edges = [(0, 5), (4, 7), (15, 23), (17, 30)]
y_true = is_edge(adjacency, edges)

scores = aa.predict(edges)
y_pred = whitened_sigmoid(scores) > 0.75

accuracy_score(y_true, y_pred)
```

[4]:
```
1.0
```

[5]:
```
# directed graph
graph = painters(metadata=True)
adjacency = graph.adjacency
names = graph.names
```

[6]:
```
picasso = 0
```

[7]:
```
aa.fit(adjacency)
```

[7]:
```
AdamicAdar()
```

[8]:
```
scores = aa.predict(picasso)

names[argsort(-scores)]
```

[8]:
```
array(['Pablo Picasso', 'Paul Cezanne', 'Claude Monet', 'Edgar Degas',
       'Vincent van Gogh', 'Henri Matisse', 'Pierre-Auguste Renoir',
       'Michel Angelo', 'Edouard Manet', 'Peter Paul Rubens', 'Rembrandt',
       'Gustav Klimt', 'Leonardo da Vinci', 'Egon Schiele'], dtype='<U21')
```

[9]:
```
# bipartite graph
graph = movie_actor(metadata=True)
biadjacency = graph.biadjacency
names = graph.names
```

[10]:
```
inception = 0
```

[11]:
```
aa.fit(biadjacency)
```

[11]:
```
AdamicAdar()
```

[12]:
```
scores = aa.predict(inception)

names[argsort(-scores)]
```

[12]:
```
array(['Inception', 'The Dark Knight Rises', 'The Great Gatsby',
       'Aviator', 'Midnight In Paris', 'The Big Short', 'Drive',
       'La La Land', 'Crazy Stupid Love', 'Vice',
       'The Grand Budapest Hotel', '007 Spectre', 'Inglourious Basterds',
       'Murder on the Orient Express', 'Fantastic Beasts 2'], dtype='<U28')
```

Jaccard Index

[13]:
```
adjacency = karate_club()
```

[14]:
```
ji = JaccardIndex()
ji.fit(adjacency)
```

[14]:
```
JaccardIndex()
```

[15]:
```
edges = [(0, 5), (4, 7), (15, 23), (17, 30)]
y_true = is_edge(adjacency, edges)

scores = ji.predict(edges)
y_pred = whitened_sigmoid(scores) > 0.75

accuracy_score(y_true, y_pred)
```

[15]:
```
0.5
```

[16]:
```
# directed graph
graph = painters(metadata=True)
adjacency = graph.adjacency
names = graph.names
```

[17]:
```
picasso = 0
```

[18]:
```
ji.fit(adjacency)
```

[18]:
```
JaccardIndex()
```

[19]:
```
scores = ji.predict(picasso)

names[argsort(-scores)]
```

[19]:
```
array(['Pablo Picasso', 'Paul Cezanne', 'Claude Monet',
       'Pierre-Auguste Renoir', 'Henri Matisse', 'Edgar Degas',
       'Vincent van Gogh', 'Michel Angelo', 'Edouard Manet',
       'Peter Paul Rubens', 'Rembrandt', 'Gustav Klimt',
       'Leonardo da Vinci', 'Egon Schiele'], dtype='<U21')
```

[20]:
```
# bipartite graph
graph = movie_actor(metadata=True)
biadjacency = graph.biadjacency
names = graph.names
```

[21]:
```
inception = 0
```

[22]:
```
ji.fit(biadjacency)
```

[22]:
```
JaccardIndex()
```

[23]:
```
scores = ji.predict(inception)

names[argsort(-scores)]
```

[23]:
```
array(['Inception', 'The Dark Knight Rises', 'The Great Gatsby',
       'Aviator', 'Midnight In Paris', 'The Big Short', 'Drive',
       'La La Land', 'Crazy Stupid Love', 'Vice',
       'The Grand Budapest Hotel', '007 Spectre', 'Inglourious Basterds',
       'Murder on the Orient Express', 'Fantastic Beasts 2'], dtype='<U28')
```

3.28 Visualization

3.28.1 Graphs

Visualization of graphs as SVG images.

[1]:
```
from IPython.display import SVG
```

[2]:
```
import numpy as np
```

[3]:
```
from sknetwork.data import karate_club, painters, movie_actor, load_netset
from sknetwork.visualization import svg_graph, svg_digraph, svg_bigraph
```

Graphs

[4]:
```
graph = karate_club(metadata=True)
adjacency = graph.adjacency
position = graph.position
labels = graph.labels
```

[5]:
```
# graph
image = svg_graph(adjacency, position, labels=labels)
SVG(image)
```

[5]:

[6]:
```
# export
image = svg_graph(adjacency, position, labels=labels, filename='karate_club')
```

[7]:
```
# adding names
image = svg_graph(adjacency, position, names=np.arange(34), name_position='below',␣
↪labels=labels)
SVG(image)
```

[7]:

[8]:
```
# node size
image = svg_graph(adjacency, position, labels=labels, display_node_weight=True)
SVG(image)
```

[8]:

[9]:
```
# scores (here = degrees)
degrees = adjacency.dot(np.ones(adjacency.shape[0]))
image = svg_graph(adjacency, position, scores=degrees)
SVG(image)
```

[9]:

[10]:
```
# seeds (here 2 nodes of highest degrees)
seeds = list(np.argsort(-degrees)[:2])
image = svg_graph(adjacency, position, labels=labels, seeds=seeds)
SVG(image)
```

[10]:

[11]:
```
# no edge
graph = load_netset('openflights')
adjacency = graph.adjacency
position = graph.position
```

```
Parsing files...
Done.
```

[12]:
```
weights = adjacency.dot(np.ones(adjacency.shape[0]))
image = svg_graph(adjacency, position, scores=np.log(weights), node_order=np.
↪argsort(weights),
                  node_size_min=2, node_size_max=10, height=400, width=800,
                  display_node_weight=True, display_edges=False)
SVG(image)
```

[12]:

Directed graphs

[13]:
```
graph = painters(metadata=True)
adjacency = graph.adjacency
names = graph.names
position = graph.position
```

[14]:
```
image = svg_digraph(adjacency, position, names, name_position='above')
SVG(image)
```

[14]:

Bipartite graphs

[15]:
```
graph = movie_actor(metadata=True)
biadjacency = graph.biadjacency
names_row = graph.names_row
names_col = graph.names_col
```

[16]:
```
# default layout
image = svg_bigraph(biadjacency, names_row, names_col, color_row='blue', color_col='red')
SVG(image)
```

[16]:

[17]:
```
# keep original order of rows and columns
image = svg_bigraph(biadjacency, names_row, names_col=names_col, color_row='blue', color_
↪col='red',
                    reorder=False)
SVG(image)
```

[17]:

3.28.2 Paths

Visualization of paths.

[1]:
```
from IPython.display import SVG
```

[2]:
```
from sknetwork.data import house, cyclic_digraph, star_wars
from sknetwork.visualization import svg_graph, svg_digraph, svg_bigraph
```

Graphs

[3]:
```
graph = house(True)
adjacency = graph.adjacency
position = graph.position
```

[4]:
```
path = [(0, 1), (1, 2), (2, 3)]
edge_labels = [(*edge, 0) for edge in path]
```

[5]:
```
image = svg_graph(adjacency, position, edge_width=3, edge_labels=edge_labels)
SVG(image)
```

[5]:

[6]:
```
image = svg_graph(None, position, edge_width=3, edge_labels=edge_labels)
SVG(image)
```

[6]:

Directed graphs

[7]:
```
graph = cyclic_digraph(10, metadata=True)
adjacency = graph.adjacency
position = graph.position
```

[8]:
```
paths = [[(0, 1), (1, 2), (2, 3)], [(6, 7), (7, 8)]]
edge_labels = [(*edge, label) for label, path in enumerate(paths) for edge in path]
```

[9]:
```
image = svg_digraph(adjacency, position, width=200, height=None, edge_width=3, edge_
↪labels=edge_labels)
SVG(image)
```

[9]:

Bipartite graphs

[10]:
```
graph = star_wars(True)
biadjacency = graph.biadjacency
names_row = graph.names_row
names_col = graph.names_col
```

[11]:
```
path = [(0, 1), (2, 1)]
edge_labels = [(*edge, 0) for edge in path]
```

[12]:
```
image = svg_bigraph(biadjacency, names_row=names_row, names_col=names_col, edge_width=3,␣
↪edge_labels=edge_labels)
SVG(image)
```

[12]:

3.28.3 Dendrograms

Visualization of dendrograms as SVG images.

[1]:
```
from IPython.display import SVG
```

[2]:
```
import numpy as np
```

[3]:
```
from sknetwork.data import karate_club, painters, movie_actor
from sknetwork.hierarchy import Paris
from sknetwork.visualization import svg_graph, svg_digraph, svg_bigraph
from sknetwork.visualization import svg_dendrogram
```

Graphs

[4]:
```
graph = karate_club(metadata=True)
adjacency = graph.adjacency
position = graph.position
labels = graph.labels
```

[5]:
```
# graph
image = svg_graph(adjacency, position, labels=labels)
SVG(image)
```

[5]:

[6]:
```
# hierarchical clustering
paris = Paris()
dendrogram = paris.fit_transform(adjacency)
```

[7]:
```
# visualization
image = svg_dendrogram(dendrogram)
SVG(image)
```

[7]:

[8]:
```
# add names, set colors
n = adjacency.shape[0]
image = svg_dendrogram(dendrogram, names=np.arange(n), n_clusters=5, color='gray')
SVG(image)
```

[8]:

[9]:
```
# export
svg_dendrogram(dendrogram, filename='dendrogram_karate_club')
```

[9]:
```
'<svg width="420" height="320"  xmlns="http://www.w3.org/2000/svg"><path stroke-width="2
↪" stroke="red" d="M 351.1764705882353 310 351.1764705882353 306.05522660108716" />
↪<path stroke-width="2" stroke="red" d="M 362.9411764705883 310 362.9411764705883 306.
↪05522660108716" /><path stroke-width="2" stroke="red" d="M 351.1764705882353 306.
↪05522660108716 362.9411764705883 306.05522660108716" /><path stroke-width="2" stroke=
↪"blue" d="M 33.529411764705884 310 33.529411764705884 306.05522660108716" /><path␣
↪stroke-width="2" stroke="blue" d="M 45.294117647058826 310 45.294117647058826 306.
↪05522660108716" /><path stroke-width="2" stroke="blue" d="M 33.529411764705884 306.
↪05522660108716 45.294117647058826 306.05522660108716" /><path stroke-width="2" stroke=
↪"red" d="M 386.47058823529414 310 386.47058823529414 305.5621300890031" /><path stroke-
↪width="2" stroke="red" d="M 398.2352941176471 310 398.2352941176471 305.5621300890031"␣
↪/><path stroke-width="2" stroke="red" d="M 386.47058823529414 305.5621300890031 398.
↪2352941176471 305.5621300890031" /><path stroke-width="2" stroke="blue" d="M 80.
↪58823529411765 310 80.58823529411765 305.5621300890031" /><path stroke-width="2"␣
↪stroke="blue" d="M 92.3529411764706 310 92.3529411764706 305.5621300890031" /><path␣
↪stroke-width="2" stroke="blue" d="M 80.58823529411765 305.5621300890031 92.
```

(continues on next page)

(continued from previous page)

Directed graphs

[10]:
```
graph = painters(metadata=True)
adjacency = graph.adjacency
names = graph.names
position = graph.position
```

[11]:
```
# graph
image = svg_digraph(adjacency, position, names)
SVG(image)
```

[11]:

[12]:
```
# hierarchical clustering
paris = Paris()
dendrogram = paris.fit_transform(adjacency)
```

[13]:
```
# visualization
image = svg_dendrogram(dendrogram, names, n_clusters=3, rotate=True)
SVG(image)
```

[13]:

Bipartite graphs

[14]:
```
graph = movie_actor(metadata=True)
biadjacency = graph.biadjacency
names_row = graph.names_row
names_col = graph.names_col
```

[15]:
```
# graph
image = svg_bigraph(biadjacency, names_row, names_col)
SVG(image)
```

[15]:

[16]:
```
# hierarchical clustering
paris = Paris()
paris.fit(biadjacency)
dendrogram_row = paris.dendrogram_row_
dendrogram_col = paris.dendrogram_col_
dendrogram_full = paris.dendrogram_full_
```

[17]:
```
# visualization
image = svg_dendrogram(dendrogram_row, names_row, n_clusters=3, rotate=True)
SVG(image)
```

[17]:

[18]:
```
image = svg_dendrogram(dendrogram_col, names_col, n_clusters=3, rotate=True)
SVG(image)
```

[18]:

3.28.4 Pie-chart nodes

Visualization of membership matrices with pie-chart nodes.

[1]:
```
from IPython.display import SVG
from scipy import sparse
```

[2]:
```
from sknetwork.data import bow_tie, karate_club, painters
from sknetwork.visualization import svg_graph, svg_digraph
from sknetwork.clustering import Louvain
```

Graphs

[3]:
```
graph = bow_tie(True)
adjacency = graph.adjacency
position = graph.position
```

[4]:
```
# probabilities
probs = [.5, 0, 0, 1, 1]
membership = sparse.csr_matrix([[p, 1-p] for p in probs])
```

[5]:
```
image = svg_graph(adjacency, position, membership=membership, node_size=10)
SVG(image)
```

[5]:

[6]:
```
graph = karate_club(True)
adjacency = graph.adjacency
position = graph.position
```

[7]:
```
# soft clustering
louvain = Louvain()
louvain.fit(adjacency)
membership = louvain.membership_
```

[8]:
```
image = svg_graph(adjacency, position, membership=membership)
SVG(image)
```

[8]:

Directed graphs

[9]:
```
graph = painters(True)
adjacency = graph.adjacency
names = graph.names
```

[10]:
```
# soft clustering
louvain = Louvain()
louvain.fit(adjacency)
membership = louvain.membership_
```

[11]:
```
image = svg_digraph(adjacency, names=names, membership=membership, node_size=10)
SVG(image)
```

[11]:

3.29 Text mining

We show how to use scikit-network for text mining. We here consider the novel Les Misérables by Victor Hugo (Project Gutenberg, translation by Isabel F. Hapgood). By considering the graph between words and paragraphs, we can embed both words and paragraphs in the same vector space and compute cosine similarity between them.

Each word is considered as in the original text; more advanced tokenizers can be used instead.

Other graphs can be considered, like the graph of co-occurrence of words within a window of 5 words, or the graph of chapters and words. These graphs can be combined to get richer information and better embeddings.

[1]:
```
from re import sub
```

[2]:
```
import numpy as np
```

[3]:
```
from sknetwork.data import from_adjacency_list
from sknetwork.embedding import Spectral
from sknetwork.linalg import normalize
```

3.29.1 Load data

[4]:
```
filename = 'miserables-en.txt'
```

[5]:
```
with open(filename, 'r') as f:
    text = f.read()
```

[6]:
```
len(text)
```

[6]:
```
3254333
```

[7]:
```
print(text[:494])
```

```
The Project Gutenberg EBook of Les Misérables, by Victor Hugo

This eBook is for the use of anyone anywhere at no cost and with almost
no restrictions whatsoever. You may copy it, give it away or re-use
it under the terms of the Project Gutenberg License included with this
eBook or online at www.gutenberg.org


Title: Les Misérables
```

(continues on next page)

(continued from previous page)

```
       Complete in Five Volumes

Author: Victor Hugo

Translator: Isabel F. Hapgood

Release Date: June 22, 2008 [EBook #135]
Last Updated: January 18, 2016
```

3.29.2 Pre-processing

[8]:
```
# extract main text
main = text.split('LES MISÉRABLES')[-2].lower()
```

[9]:
```
len(main)
```

[9]:
```
3215017
```

[10]:
```
# remove ponctuation
main = sub(r"[,.;:()@#?!&$'_*]", " ", main)
main = sub(r'["-]', ' ', main)
```

[11]:
```
# extract paragraphs
sep = '|||'
main = sub(r'\n\n+', sep, main)
main = sub('\n', ' ', main)
paragraphs = main.split(sep)
```

[12]:
```
len(paragraphs)
```

[12]:
```
13499
```

[13]:
```
paragraphs[1000]
```

[13]:
```
'after leaving the asses there was a fresh delight  they crossed the seine in a boat ␣
↪and proceeding from passy on foot they reached the barrier of l étoile  they had been␣
↪up since five o clock that morning  as the reader will remember  but  bah  there is no␣
↪such thing as fatigue on sunday   said favourite   on sunday fatigue does not work  '
```

3.29.3 Build graph

[14]:
```
paragraph_words = [paragraph.split(' ') for paragraph in paragraphs]
```

[15]:
```
graph = from_adjacency_list(paragraph_words, bipartite=True)
```

[16]:
```
biadjacency = graph.biadjacency
words = graph.names_col
```

[17]:
```
biadjacency
```

[17]:
```
<13499x23093 sparse matrix of type '<class 'numpy.int64'>'
        with 416331 stored elements in Compressed Sparse Row format>
```

[18]:
```
len(words)
```

[18]:
```
23093
```

3.29.4 Statistics

[19]:
```
n_row, n_col = biadjacency.shape
```

[20]:
```
paragraph_lengths = biadjacency.dot(np.ones(n_col))
```

[21]:
```
np.quantile(paragraph_lengths, [0.1, 0.5, 0.9, 0.99])
```

[21]:
```
array([  6.,  23., 127., 379.])
```

[22]:
```
word_counts = biadjacency.T.dot(np.ones(n_row))
```

[23]:
```
np.quantile(word_counts, [0.1, 0.5, 0.9, 0.99])
```

[23]:
```
array([  1.  ,   2.  ,  23.  , 282.08])
```

3.29.5 Embedding

[24]:
```
dimension = 50
spectral = Spectral(dimension, regularization=100)
```

[25]:
```
spectral.fit(biadjacency)
```

[25]:
```
Spectral(n_components=50, decomposition='rw', regularization=100, normalized=True)
```

[26]:
```
embedding_paragraph = spectral.embedding_row_
embedding_word = spectral.embedding_col_
```

[27]:
```
# some word
i = int(np.argwhere(words == 'love'))
```

[28]:
```
# most similar words
cosines_word = embedding_word.dot(embedding_word[i])
words[np.argsort(-cosines_word)[:20]]
```

[28]:
```
array(['love', 'kiss', 'ye', 'celestial', 'hearts', 'loved', 'tender',
       'roses', 'joys', 'sweet', 'wedded', 'charming', 'angelic', 'adore',
       'aurora', 'pearl', 'voluptuousness', 'chaste', 'innumerable',
       'heart'], dtype='<U21')
```

[29]:
```
np.quantile(cosines_word, [0.01, 0.1, 0.5, 0.9, 0.99])
```

[29]:
```
array([-0.24307366, -0.14047851, -0.02607974,  0.14319717,  0.42843234])
```

[30]:
```
# some paragraph
i = 1000
print(paragraphs[i])
```

```
after leaving the asses there was a fresh delight  they crossed the seine in a boat  and␣
↪proceeding from passy on foot they reached the barrier of l étoile  they had been up␣
↪since five o clock that morning  as the reader will remember  but  bah  there is no␣
↪such thing as fatigue on sunday  said favourite  on sunday fatigue does not work
```

[31]:
```
# most similar paragraphs
cosines_paragraph = embedding_paragraph.dot(embedding_paragraph[i])
for j in np.argsort(-cosines_paragraph)[:3]:
    print(paragraphs[j])
    print()
```

```
after leaving the asses there was a fresh delight  they crossed the seine in a boat  and␣
↪proceeding from passy on foot they reached the barrier of l étoile  they had been up␣
↪since five o clock that morning  as the reader will remember  but  bah  there is no␣
↪such thing as fatigue on sunday  said favourite  on sunday fatigue does not work

he was a man of lofty stature  half peasant  half artisan  he wore a huge leather apron ␣
↪which reached to his left shoulder  and which a hammer  a red handkerchief  a powder␣
↪horn  and all sorts of objects which were upheld by the girdle  as in a pocket  caused␣
↪to bulge out  he carried his head thrown backwards  his shirt  widely opened and␣
↪turned back  displayed his bull neck  white and bare  he had thick eyelashes  enormous␣
↪black whiskers  prominent eyes  the lower part of his face like a snout  and besides␣
↪all this  that air of being on his own ground  which is indescribable

this was the state which the shepherd idyl  begun at five o clock in the morning  had␣
↪reached at half past four in the afternoon  the sun was setting  their appetites were␣
↪satisfied
```

[32]:
```
np.quantile(cosines_paragraph, [0.01, 0.1, 0.5, 0.9, 0.99])
```

[32]:
```
array([-0.30671191, -0.17309593, -0.00319729,  0.21574375,  0.45969887])
```

3.30 Wikipedia

This notebook shows how to apply scikit-network to analyse the network structure of Wikipedia, through its hyperlinks.

We consider the Wikivitals dataset of the netset collection. This dataset consists of the (approximately) top 10,000 (vital) articles of Wikipedia.

[1]:
```
from IPython.display import SVG
```

[2]:
```
import numpy as np
```

[3]:
```
from sknetwork.data import load_netset
from sknetwork.ranking import PageRank, top_k
from sknetwork.embedding import Spectral
from sknetwork.clustering import Louvain
from sknetwork.classification import DirichletClassifier
from sknetwork.utils import WardDense, get_neighbors
from sknetwork.visualization import svg_dendrogram
```

3.30.1 Data

All datasets of the netset collection can be easily imported with scikit-network.

[4]:
```
wikivitals = load_netset('wikivitals')
```

```
Parsing files...
Done.
```

[5]:
```
# graph of links
adjacency = wikivitals.adjacency
names = wikivitals.names
labels = wikivitals.labels
names_labels = wikivitals.names_labels
```

[6]:
```
adjacency
```

[6]:
```
<10011x10011 sparse matrix of type '<class 'numpy.bool_'>'
        with 824999 stored elements in Compressed Sparse Row format>
```

[7]:
```
# categories
print(names_labels)
```

```
['Arts' 'Biological and health sciences' 'Everyday life' 'Geography'
 'History' 'Mathematics' 'People' 'Philosophy and religion'
 'Physical sciences' 'Society and social sciences' 'Technology']
```

[8]:
```
# get label
label_id = {name: i for i, name in enumerate(names_labels)}
```

3.30.2 Sample

Let's have a look at an article.

[9]:
```
i = 10000
print(names[i])
```

```
Édouard Manet
```

[10]:
```
# label
label = labels[i]
print(names_labels[label])
```

```
People
```

[11]:
```
# some hyperlinks
neighbors = get_neighbors(adjacency, i)
print(names[neighbors[:10]])
```

```
['Adolphe Thiers' 'American Civil War' 'Bordeaux' 'Camille Pissarro'
 'Carmen' 'Charles Baudelaire' 'Claude Monet' 'Diego Velázquez'
 'Edgar Allan Poe' 'Edgar Degas']
```

[12]:
```
len(neighbors)
```

[12]:
```
38
```

3.30.3 PageRank

We first use (personalized) PageRank to select typical articles of each category.

[13]:
```
pagerank = PageRank()
```

[14]:
```
# number of articles per category
n_selection = 50
```

[15]:
```
# selection of articles
selection = []
for label in np.arange(len(names_labels)):
    ppr = pagerank.fit_transform(adjacency, seeds=(labels==label))
    scores = ppr * (labels==label)
    selection.append(top_k(scores, n_selection))
selection = np.array(selection)
```

[16]:
```
selection.shape
```

[16]:
```
(11, 50)
```

[17]:
```
# show selection
for label, name_label in enumerate(names_labels):
    print('---')
    print(label, name_label)
    print(names[selection[label, :5]])
```

```
---
0 Arts
['Encyclopædia Britannica' 'Romanticism' 'Jazz' 'Modernism' 'Baroque']
---
1 Biological and health sciences
['Taxonomy (biology)' 'Animal' 'Chordate' 'Plant' 'Species']
---
2 Everyday life
['Olympic Games' 'Association football' 'Basketball' 'Baseball' 'Softball']
---
3 Geography
['Geographic coordinate system' 'United States' 'China' 'France' 'India']
---
4 History
['World War II' 'World War I' 'Roman Empire' 'Ottoman Empire'
 'Middle Ages']
---
5 Mathematics
['Real number' 'Function (mathematics)' 'Complex number'
 'Set (mathematics)' 'Integer']
---
6 People
['Aristotle' 'Plato' 'Augustine of Hippo' 'Winston Churchill'
 'Thomas Aquinas']
---
7 Philosophy and religion
['Christianity' 'Islam' 'Buddhism' 'Hinduism' 'Catholic Church']
---
8 Physical sciences
['Oxygen' 'Hydrogen' 'Earth' 'Kelvin' 'Density']
---
9 Society and social sciences
['The New York Times' 'Latin' 'English language' 'French language'
 'United Nations']
---
10 Technology
['NASA' 'Internet' 'Operating system' 'Computer network' 'Computer']
```

3.30.4 Embedding

We now represent each node of the graph by a vector in low dimension, and use hierarchical clustering to visualize the structure of this embedding.

[18]:
```
# dimension of the embedding
n_components = 20
```

[19]:
```
# embedding
spectral = Spectral(n_components)
embedding = spectral.fit_transform(adjacency)
```

[20]:
```
embedding.shape
```

[20]:
```
(10011, 20)
```

[21]:
```
ward = WardDense()
```

[22]:
```
# hierarchy of articles
label = label_id['Physical sciences']
index = selection[label]
dendrogram_articles = ward.fit_transform(embedding[index])
```

[23]:
```
# visualization
image = svg_dendrogram(dendrogram_articles, names=names[index], rotate=True, width=200,␣
↪scale=2, n_clusters=4)
SVG(image)
```

[23]:

3.30.5 Clustering

We now apply Louvain to get a clustering of the graph, independently of the known labels.

[24]:
```
algo = Louvain()
```

[25]:
```
labels_pred = algo.fit_transform(adjacency)
```

[26]:
```
np.unique(labels_pred, return_counts=True)
```

[26]:
```
(array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10]),
 array([1800, 1368, 1315, 1225, 1165, 1067,  804,  672,  468,   97,   30]))
```

We use again PageRank to get the top pages of each cluster.

[27]:
```
n_selection = 5
```

[28]:
```
selection = []
for label in np.arange(len(set(labels_pred))):
    ppr = pagerank.fit_transform(adjacency, seeds=(labels_pred==label))
    scores = ppr * (labels_pred==label)
    selection.append(top_k(scores, n_selection))
selection = np.array(selection)
```

[29]:
```
# show selection
for label in np.arange(len(set(labels_pred))):
    print('---')
    print(label)
    print(names[selection[label]])
```

```
---
0
['Taxonomy (biology)' 'Animal' 'Plant' 'Protein' 'Species']
---
1
['Hydrogen' 'Oxygen' 'Kelvin' 'Electron' 'Physics']
```

(continues on next page)

(continued from previous page)

```
---
2
['Latin' 'World War I' 'Roman Empire' 'Middle Ages' 'Greek language']
---
3
['United States' 'World War II' 'Geographic coordinate system'
 'United Kingdom' 'France']
---
4
['Christianity' 'Aristotle' 'Plato' 'Catholic Church'
 'Age of Enlightenment']
---
5
['China' 'India' 'Buddhism' 'Islam' 'Chinese language']
---
6
['The New York Times' 'New York City' 'Time (magazine)' 'BBC'
 'The Washington Post']
---
7
['Earth' 'Atlantic Ocean' 'Europe' 'Drainage basin' 'Pacific Ocean']
---
8
['Real number' 'Function (mathematics)' 'Complex number'
 'Set (mathematics)' 'Mathematical analysis']
---
9
['Marriage' 'Incest' 'Adoption' 'Kinship' 'Human sexuality']
---
10
['Handbag' 'Hat' 'Veil' 'Uniform' 'Clothing']
```

3.30.6 Classification

Finally, we use the Dirichlet classifier to predict the closest category for each page in the People category.

[30]:
```
algo = DirichletClassifier()
```

[31]:
```
people = label_id['People']
```

[32]:
```
labels_people = algo.fit_transform(adjacency, seeds = {i: label for i, label in␣
↪enumerate(labels) if label != people})
```

[33]:
```
n_selection = 5
```

[34]:
```
selection = []
for label in np.arange(len(names_labels)):
    if label != people:
        ppr = pagerank.fit_transform(adjacency, seeds=(labels==people)*(labels_
↪people==label))
```

(continues on next page)

(continued from previous page)

```
        scores = ppr * (labels==people)*(labels_people==label)
        selection.append(top_k(scores, n_selection))
selection = np.array(selection)
```

[35]:
```
# show selection
i = 0
for label, name_label in enumerate(names_labels):
    if label != people:
        print('---')
        print(label, name_label)
        print(names[selection[i]])
        i += 1
```

```
---
0 Arts
['Richard Wagner' 'Igor Stravinsky' 'Bob Dylan' 'Fred Astaire'
 'Ludwig van Beethoven']
---
1 Biological and health sciences
['Charles Darwin' 'Francis Crick' 'Robert Koch' 'Alexander Fleming'
 'Carl Linnaeus']
---
2 Everyday life
['Wayne Gretzky' 'Jim Thorpe' 'Jackie Robinson' 'LeBron James'
 'Willie Mays']
---
3 Geography
['Elizabeth II' 'Carl Lewis' 'Dwight D. Eisenhower' 'Vladimir Putin'
 'Muhammad Ali']
---
4 History
['Alexander the Great' 'Napoleon' 'Charlemagne' 'Philip II of Spain'
 'Charles V, Holy Roman Emperor']
---
5 Mathematics
['Euclid' 'Augustin-Louis Cauchy' 'Archimedes' 'John von Neumann'
 'Pierre de Fermat']
---
7 Philosophy and religion
['Augustine of Hippo' 'Aristotle' 'Thomas Aquinas' 'Plato' 'Immanuel Kant']
---
8 Physical sciences
['Albert Einstein' 'Isaac Newton' 'J. J. Thomson' 'Marie Curie'
 'Niels Bohr']
---
9 Society and social sciences
['Barack Obama' 'Noam Chomsky' 'Karl Marx' 'Ralph Waldo Emerson'
 'Jean-Paul Sartre']
---
10 Technology
['Tim Berners-Lee' 'Donald Knuth' 'Edsger W. Dijkstra' 'Douglas Engelbart'
 'Dennis Ritchie']
```

3.31 Recommendation

This notebook shows how to apply scikit-network for content recommendation.

We use consider the Movielens dataset of the netset collection, corresponding to ratings of 9066 movies by 671 users.

[1]:
```
from IPython.display import SVG
```

[2]:
```
import numpy as np
```

[3]:
```
from sknetwork.data import load_netset
from sknetwork.ranking import PageRank, top_k
from sknetwork.embedding import Spectral
from sknetwork.utils import WardDense, get_neighbors
from sknetwork.visualization import svg_dendrogram
```

3.31.1 Data

[4]:
```
dataset = load_netset('movielens')
```

```
Parsing files...
Done.
```

[5]:
```
biadjacency = dataset.biadjacency
names = dataset.names
labels = dataset.labels
names_labels = dataset.names_labels
```

[6]:
```
biadjacency
```

[6]:
```
<9066x671 sparse matrix of type '<class 'numpy.float64'>'
        with 100004 stored elements in Compressed Sparse Row format>
```

[7]:
```
n_movies, n_users = biadjacency.shape
```

[8]:
```
# ratings
np.unique(biadjacency.data, return_counts=True)
```

[8]:
```
(array([0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5, 5. ]),
 array([ 1101,  3326,  1687,  7271,  4449, 20064, 10538, 28750,  7723,
        15095]))
```

[9]:
```
# positive ratings
positive = biadjacency >= 3
```

[10]:
```
positive
```

[10]:
```
<9066x671 sparse matrix of type '<class 'numpy.bool_'>'
        with 82170 stored elements in Compressed Sparse Row format>
```

[11]:
```
names_labels
```

[11]:
```
array(['Action', 'Adventure', 'Animation', 'Children', 'Comedy', 'Crime',
       'Documentary', 'Drama', 'Fantasy', 'Film-Noir', 'Horror', 'IMAX',
       'Musical', 'Mystery', 'Romance', 'Sci-Fi', 'Thriller', 'War',
       'Western'], dtype='<U11')
```

[12]:
```
labels.shape
```

[12]:
```
(9066, 19)
```

3.31.2 PageRank

We first use (personalized) PageRank to get the most popular movies of each category.

[13]:
```
pagerank = PageRank()
```

[14]:
```
# top-10 movies
scores = pagerank.fit_transform(positive)
names[top_k(scores, 10)]
```

[14]:
```
array(['Forrest Gump (1994)', 'Pulp Fiction (1994)',
       'Shawshank Redemption, The (1994)',
       'Silence of the Lambs, The (1991)',
       'Star Wars: Episode IV - A New Hope (1977)', 'Matrix, The (1999)',
       'Jurassic Park (1993)', "Schindler's List (1993)",
       'Back to the Future (1985)',
       'Star Wars: Episode V - The Empire Strikes Back (1980)'],
      dtype=object)
```

[15]:
```
# number of movies per genre
n_selection = 10
```

[16]:
```
# selection
selection = []
for label in np.arange(len(names_labels)):
    ppr = pagerank.fit_transform(positive, seeds=labels[:, label])
    scores = ppr * labels[:, label]
    selection.append(top_k(scores, n_selection))
selection = np.array(selection)
```

[17]:
```
# show selection (some movies may have several genres)
for label, name_label in enumerate(names_labels):
    print('---')
    print(label, name_label)
    print(names[selection[label, :5]])
```

```
---
0 Action
['Star Wars: Episode IV - A New Hope (1977)' 'Matrix, The (1999)'
 'Jurassic Park (1993)'
 'Star Wars: Episode V - The Empire Strikes Back (1980)'
```

(continues on next page)

(continued from previous page)

```
 'Terminator 2: Judgment Day (1991)']
---
1 Adventure
['Star Wars: Episode IV - A New Hope (1977)' 'Jurassic Park (1993)'
 'Star Wars: Episode V - The Empire Strikes Back (1980)'
 'Back to the Future (1985)' 'Toy Story (1995)']
---
2 Animation
['Gnomeo & Juliet (2011)' 'Hop (2011)'
 "Lion King II: Simba's Pride, The (1998)" 'Mars Needs Moms (2011)'
 'Once Upon a Forest (1993)']
---
3 Children
['Spy Kids 3-D: Game Over (2003)' 'Race to Witch Mountain (2009)'
 'G-Force (2009)' 'Prancer (1989)' 'Diary of a Wimpy Kid (2010)']
---
4 Comedy
['Forrest Gump (1994)' 'Pulp Fiction (1994)' 'Back to the Future (1985)'
 'Toy Story (1995)' 'Fargo (1996)']
---
5 Crime
['Pulp Fiction (1994)' 'Shawshank Redemption, The (1994)'
 'Silence of the Lambs, The (1991)' 'Fargo (1996)' 'Godfather, The (1972)']
---
6 Documentary
['Queen of Versailles, The (2012)' 'Powaqqatsi (1988)'
 'Fragile Trust: Plagiarism, Power, and Jayson Blair at the New York Times, A (2013)'
 'African Cats (2011)' 'Eddie Murphy Delirious (1983)']
---
7 Drama
['Pulp Fiction (1994)' 'Forrest Gump (1994)'
 'Shawshank Redemption, The (1994)' "Schindler's List (1993)"
 'American Beauty (1999)']
---
8 Fantasy
['The Pumaman (1980)'
 'Golem, The (Golem, wie er in die Welt kam, Der) (1920)'
 'Twilight Saga: New Moon, The (2009)'
 'Highlander: Endgame (Highlander IV) (2000)'
 'Ghost Rider: Spirit of Vengeance (2012)']
---
9 Film-Noir
['No Way Out (1950)' 'Johnny Eager (1942)'
 'Lady from Shanghai, The (1947)' 'This World, Then the Fireworks (1997)'
 'T-Men (1947)']
---
10 Horror
['Silence of the Lambs, The (1991)' 'Carnosaur 3: Primal Species (1996)'
 "Devil's Chair, The (2006)" 'AVPR: Aliens vs. Predator - Requiem (2007)'
 'Jason Goes to Hell: The Final Friday (1993)']
---
11 IMAX
```

(continues on next page)

(continued from previous page)

```
["Madagascar 3: Europe's Most Wanted (2012)" 'Final Destination 5 (2011)'
 'Jack the Giant Slayer (2013)' "Dr. Seuss' The Lorax (2012)"
 'After Earth (2013)']
---
12 Musical
['Gypsy (1993)' "Breakin' (1984)" "Breakin' 2: Electric Boogaloo (1984)"
 'True Stories (1986)' 'Camp Rock (2008)']
---
13 Mystery
['Double, The (2011)' 'In the Electric Mist (2009)'
 'Spirits of the Dead (1968)' 'Nomads (1986)'
 'Fast and the Furious, The (1955)']
---
14 Romance
['Forrest Gump (1994)' 'American Beauty (1999)'
 'Princess Bride, The (1987)' 'Beauty and the Beast (1991)'
 'Good Will Hunting (1997)']
---
15 Sci-Fi
['Star Wars: Episode IV - A New Hope (1977)' 'Matrix, The (1999)'
 'Star Wars: Episode V - The Empire Strikes Back (1980)'
 'Jurassic Park (1993)' 'Back to the Future (1985)']
---
16 Thriller
['Pulp Fiction (1994)' 'Silence of the Lambs, The (1991)'
 'Matrix, The (1999)' 'Jurassic Park (1993)' 'Fargo (1996)']
---
17 War
['Pathfinder (2007)' 'Green Berets, The (1968)'
 'They Were Expendable (1945)' 'Legionnaire (1998)' 'Iron Eagle II (1988)']
---
18 Western
['Bandidas (2006)' "'Neath the Arizona Skies (1934)"
 'American Outlaws (2001)' 'The Missouri Breaks (1976)'
 'Stagecoach (1966)']
```

We now apply PageRank to get the most relevant movies associated with a given movie.

[18]:
```
target = {i: name for i, name in enumerate(names) if 'Cherbourg' in name}
```

[19]:
```
target
```

[19]:
```
{175: 'Umbrellas of Cherbourg, The (Parapluies de Cherbourg, Les) (1964)'}
```

[20]:
```
scores_ppr = pagerank.fit_transform(positive, seeds={175:1})
```

[21]:
```
names[top_k(scores_ppr - scores, 10)]
```

[21]:
```
array(['Umbrellas of Cherbourg, The (Parapluies de Cherbourg, Les) (1964)',
       'Fargo (1996)', 'Pulp Fiction (1994)',
       'Star Wars: Episode IV - A New Hope (1977)',
       'L.A. Confidential (1997)', 'Matrix, The (1999)',
```

(continues on next page)

(continued from previous page)

```
       'Shawshank Redemption, The (1994)', 'American Beauty (1999)',
       'Clockwork Orange, A (1971)', 'Jurassic Park (1993)'], dtype=object)
```

We can also apply PageRank to make recommend movies to a user.

[22]:
```
user = 1
targets = get_neighbors(positive, user, transpose=True)
```

[23]:
```
# seen movies (sample)
names[targets][:10]
```

[23]:
```
array(['GoldenEye (1995)', 'Sense and Sensibility (1995)',
       'Clueless (1995)', 'Seven (a.k.a. Se7en) (1995)',
       'Usual Suspects, The (1995)', 'Mighty Aphrodite (1995)',
       "Mr. Holland's Opus (1995)", 'Braveheart (1995)',
       'Brothers McMullen, The (1995)', 'Apollo 13 (1995)'], dtype=object)
```

[24]:
```
mask = np.zeros(len(names), dtype=bool)
mask[targets] = 1
```

[25]:
```
scores_ppr = pagerank.fit_transform(positive, seeds=mask)
```

[26]:
```
# top-10 recommendation
names[top_k((scores_ppr - scores) * (1 - mask), 10)]
```

[26]:
```
array(['Shawshank Redemption, The (1994)', 'True Lies (1994)',
       'Star Wars: Episode IV - A New Hope (1977)',
       'Beauty and the Beast (1991)', 'Toy Story (1995)',
       'Twelve Monkeys (a.k.a. 12 Monkeys) (1995)', 'Fargo (1996)',
       'Independence Day (a.k.a. ID4) (1996)', 'Matrix, The (1999)',
       'Star Wars: Episode V - The Empire Strikes Back (1980)'],
      dtype=object)
```

3.31.3 Embedding

We now represent each movie by a vector in low dimension, and use hierarchical clustering to visualize the structure of this embedding for top-100 movies.

[27]:
```
# embedding
spectral = Spectral(10)
embedding = spectral.fit_transform(positive)
```

[28]:
```
ward = WardDense()
```

[29]:
```
# top-100 movies
scores = pagerank.fit_transform(positive)
index = top_k(scores, 100)
dendrogram = ward.fit_transform(embedding[index])
```

[30]:
```
# visualization
image = svg_dendrogram(dendrogram, names=names[index], rotate=True, width=200,␣
↪height=1000, n_clusters=6)
SVG(image)
```

[30]:

3.32 Politics

We show how to use scikit-network to analyse the way deputies vote and their proximity to the majority. We here consider the French National Assembly (XVth legislature, from 2017 to 2020). The considered graph is the bipartite graph between deputies and bills.

[1]:
```
from IPython.display import SVG
```

[2]:
```
import numpy as np
```

[3]:
```
from sknetwork.data import load_netset
from sknetwork.clustering import Louvain
from sknetwork.regression import Diffusion
from sknetwork.ranking import top_k, PageRank
from sknetwork.embedding import Spectral
from sknetwork.visualization import svg_graph, svg_dendrogram
```

3.32.1 Data

The dataset is part of the NetSet collection.

[4]:
```
graph = load_netset('national_assembly')
```
```
Parsing files...
Done.
```

[5]:
```
biadjacency = graph.biadjacency
position = graph.position
names = graph.names_row
bills = graph.names_col
labels = graph.labels
label_colors = graph.label_colors
names_labels = graph.names_labels
```

[6]:
```
n_deputy, n_bill = biadjacency.shape
```

[7]:
```
print(names_labels)
```
```
['Non inscrit' 'Les Républicains'
 'Les Constructifs : républicains, UDI, indépendants'
 'Libertés et Territoires' 'UDI et Indépendants'
 'UDI, Agir et Indépendants' 'Mouvement Démocrate et apparentés'
 'La République en Marche' 'Socialistes et apparentés' 'Nouvelle Gauche'
 'Gauche démocrate et républicaine' 'La France insoumise']
```

[8]:
```
# parameters for visualization
node_size = 4
width = 480
height = 300
```

[9]:
```
image = svg_graph(position=position, labels=labels, node_size=node_size, width=width,␣
↪height=height,
                  label_colors=label_colors)
SVG(image)
```

[9]:

[10]:
```
labels_majority = [6,7]
```

[11]:
```
print(names_labels[labels_majority])
```

```
['Mouvement Démocrate et apparentés' 'La République en Marche']
```

[12]:
```
# majority
majority = np.isin(labels, labels_majority)
np.sum(majority)
```

[12]:
```
356
```

[13]:
```
# opposition
opposition = ~np.isin(labels, labels_majority)
len(opposition)
```

[13]:
```
577
```

3.32.2 Votes

[14]:
```
biadjacency_for = biadjacency > 0
biadjacency_against = (-biadjacency > 0)
```

[15]:
```
biadjacency_for
```

[15]:
```
<577x2807 sparse matrix of type '<class 'numpy.bool_'>'
        with 119901 stored elements in Compressed Sparse Row format>
```

[16]:
```
bills_for = biadjacency_for.T.dot(np.ones(n_deputy))
bills_against = biadjacency_against.T.dot(np.ones(n_deputy))
bills_total = bills_for + bills_against
```

[17]:
```
print('Average participation = ', np.round(np.sum(bills_total) / n_bill / n_deputy, 2))
```

```
Average participation =  0.17
```

3.32.3 Clustering

[18]:
```
louvain = Louvain()
```

[19]:
```
labels_pred = louvain.fit_transform(biadjacency_for)
```

[20]:
```
np.unique(labels_pred, return_counts=True)
```

[20]:
```
(array([0, 1, 2]), array([213, 363,   1]))
```

[21]:
```
labels_pred_majority, counts_majority = np.unique(labels_pred[majority], return_
↪counts=True)
```

[22]:
```
label_pred_majority = labels_pred_majority[np.argmax(counts_majority)]
```

[23]:
```
image = svg_graph(position=position, labels=labels_pred, node_size=node_size,␣
↪width=width, height=height, label_colors=['red','blue', 'lightgrey'])
SVG(image)
```

[23]:

[24]:
```
neutral = np.argwhere(labels_pred==2).ravel()
```

[25]:
```
print(names[neutral])
```

```
['Laure de La Raudière']
```

[26]:
```
# dissident
print(names[majority * (labels_pred!=label_pred_majority)])
```

```
['Frédérique Dumas' 'Maud Petit' 'Sonia Krimi' 'Richard Ramos'
 'Sébastien Nadot']
```

3.32.4 Diffusion

[27]:
```
diffusion = Diffusion(n_iter=4)
```

[28]:
```
values = diffusion.fit_transform(biadjacency_for, seeds_row=majority)
```

[29]:
```
image = svg_graph(position=position, scores=values, node_size=node_size,
                  width=width, height=height)
SVG(image)
```

[29]:

[30]:
```
# top-10 deputies for majority
index = np.argwhere(majority).ravel()
top = index[top_k(values[index], 10)]
print(names[top])
```

```
['Émilie Chalas' 'Didier Paris' 'Richard Ferrand'
 'Élise Fajgeles|||Benjamin Griveaux' 'Guillaume Vuilletet'
 'Yaël Braun-Pivet' 'Marie Guévenoux' 'Jean Terlier' 'Sacha Houlié'
 'Thomas Rudigoz']
```

[31]:
```
# bottom-10 deputies for majority
bottom = index[top_k(-values[index], 10)]
print(names[bottom])
```

```
['Frédérique Dumas' 'Maud Petit' 'Richard Ramos' 'Agnès Thill'
 'Brahim Hammouche' 'Sébastien Nadot' 'Jimmy Pahun' 'Josy Poueyto'
 'Sonia Krimi' 'Max Mathiasin']
```

[32]:
```
# top-10 deputies for opposition
index = np.argwhere(opposition).ravel()
top = index[top_k(-values[index], 10)]
print(names[top])
```

```
['Ugo Bernalicis' 'Adrien Quatennens' 'Alain Bruneel' 'Clémentine Autain'
 'Nicolas Dupont-Aignan' 'Alexis Corbière' 'Jean-Luc Mélenchon'
 'Sébastien Jumel' 'Jean-Hugues Ratenon' 'Pierre Dharréville']
```

[33]:
```
# bottom-10 deputies for opposition
bottom = index[top_k(values[index], 10)]
print(names[bottom])
```

```
['Jean-Luc Warsmann' 'Olivier Dassault' 'Napole Polutele|||Sylvain Brial'
 'Stéphane Demilly' 'Michèle Tabarot' 'Bernard Deflesselles'
 'Franck Riester|||Patricia Lemoine' 'Thierry Robert|||Jean-Luc Poudroux'
 'Laure de La Raudière' 'Philippe Gomès']
```

3.32.5 Bills

[34]:
```
# labels are on deputies so you need an odd number of iterations
diffusion = Diffusion(n_iter=5)
```

[35]:
```
diffusion.fit(biadjacency_for, seeds_row=majority)
```

[35]:
```
Diffusion(n_iter=5, damping_factor=1.0)
```

[36]:
```
values_bill = diffusion.values_col_
```

[37]:
```
# top-5 bills for majority
for i in top_k(values_bill, 5):
    print(bills[i] + '\n')
```

```
l'article 27 du projet de loi de programmation 2018-2022 et de réforme pour la justice␣
↪(première lecture).

l'amendement de suppression n° 72 du Gouvernement à l'article 9 de la proposition de loi␣
↪d'orientation et de programmation relative à la sécurité intérieure (première lecture).

l'amendement de suppression n° 71 du Gouvernement à l'article 3 de la proposition de loi␣
↪d'orientation et de programmation relative à la sécurité intérieure (première lecture).

l'amendement n° 2362  de Mme Marsaud après l'article 60 du projet de loi portant␣
↪évolution du logement, de l'aménagement et du numérique (première lecture)
```

(continues on next page)

(continued from previous page)

```
l'article 13 du projet de loi de programmation 2018-2022 et de réforme pour la justice␣
↪(première lecture).
```

[38]:
```
# top-5 bills for opposition
for i in top_k(-values_bill, 5):
    print(bills[i] + '\n')
```

```
l'amendement n° 602 de M. Bernalicis à l'article 32 du projet de loi de programmation␣
↪2018-2022 et de réforme pour la justice (première lecture).

l'amendement n° 208 de Mme Obono après l'article 31 bis du projet de loi de␣
↪programmation 2018-2022 et de réforme pour la justice (première lecture).

l'amendement n° 205 de Mme Obono après l'article 30 du projet de loi de programmation␣
↪2018-2022 et de réforme pour la justice (première lecture).

l'amendement n° 11 de M. Bernalicis à l'article unique de la proposition de loi␣
↪renforçant la lutte contre les rodéos motorisés (première lecture).

l'amendement n° 272 de Mme Obono à l'article 50 du projet de loi de programmation 2018-
↪2022 et de réforme pour la justice (première lecture).
```

[39]:
```
# top-5 controversial
for i in top_k(-np.abs(values_bill-0.5), 5):
    print(bills[i] + '\n')
```

```
l'article unique de la proposition de loi organique visant à permettre l'inscription d
↪'office sur la liste électorale spéciale à la consultation sur l'accession à la pleine␣
↪souveraineté de la Nouvelle-Calédonie (première lecture).

l'amendement n° 66 de Mme Auconie et les amendements identiques suivants à l'article␣
↪premier du projet de loi renforçant la lutte contre les violences sexuelles et␣
↪sexistes (première lecture).

l'amendement de suppression n° 168 de Mme Obono et les amendements identiques suivants à␣
↪l'article 9 ter du projet de loi pour une immigration ma^ıtrisée, un droit d'asile␣
↪effectif et une intégration réussie (nouvelle lecture).

l'amendement n° 962 de la commission du développement durable et l'amendement identique␣
↪suivant après l'article 14 septies du projet de loi pour l'équilibre des relations␣
↪commerciales dans le secteur agricole et alimentaire et une alimentation saine et␣
↪durable (première lecture).

l'amendement n° 1118 de M. Clément à l'article premier du projet de loi pour une␣
↪immigration ma^ıtrisée, un droit d'asile effectif et une intégration réussie (première␣
↪lecture).
```

3.32.6 Embedding

[40]:
```
spectral = Spectral(2, normalized=False)
```

[41]:
```
embedding = spectral.fit_transform(biadjacency_for)
```

[42]:
```
image = svg_graph(position=embedding, names=names, labels=labels, node_size=5, width=400,
↪ height=1000,
                  label_colors=label_colors)
SVG(image)
```

[42]:

3.33 Sport

This notebook shows how to use scikit-network to analyse sport data.

We here consider the results of tennis matches of ATP Tour in the period 2001–2016.

[1]:
```
from IPython.display import SVG
```

[2]:
```
import numpy as np
import pandas as pd
from scipy import sparse
```

[3]:
```
from sknetwork.data import from_edge_list
from sknetwork.ranking import PageRank, top_k
from sknetwork.topology import CoreDecomposition
from sknetwork.utils import directed2undirected
from sknetwork.embedding import Spectral
from sknetwork.visualization import svg_digraph, svg_graph
```

3.33.1 Load data

[4]:
```
filename = 'atp.csv'
```

[5]:
```
df = pd.read_csv(filename, sep=';')
```

[6]:
```
df.head()
```

[6]:
```
   ATP Location                          Tournament        Date  \
0   25  Houston  U.S. Men's Clay Court Championships  2005-04-21
1   26  Estoril                        Estoril Open  2005-04-27
2   28     Rome          Telecom Italia Masters Roma  2005-05-03
3   28     Rome          Telecom Italia Masters Roma  2005-05-04
4   29  Hamburg                         Hamburg TMS  2005-05-11

                 Series    Court Surface      Round  Best of          Winner  \
0         International  Outdoor    Clay  2nd Round        3         Haas T.
1  International Series  Outdoor    Clay  2nd Round        3       Gaudio G.
```

(continues on next page)

(continued from previous page)

```
2               Masters  Outdoor    Clay  1st Round        3  Sanguinetti D.
3               Masters  Outdoor    Clay  2nd Round        3      Almagro N.
4               Masters  Outdoor    Clay  2nd Round        3       Hrbaty D.

   ...  L4  W5  L5  Wsets  Lsets    Comment  MaxW  MaxL  AvgW  AvgL
0  ... NaN NaN NaN    2.0    0.0  Completed   NaN   NaN   NaN   NaN
1  ... NaN NaN NaN    2.0    0.0  Completed   NaN   NaN   NaN   NaN
2  ... NaN NaN NaN    2.0    1.0  Completed   NaN   NaN   NaN   NaN
3  ... NaN NaN NaN    2.0    0.0  Completed   NaN   NaN   NaN   NaN
4  ... NaN NaN NaN    2.0    0.0  Completed   NaN   NaN   NaN   NaN

[5 rows x 32 columns]
```

[7]:
```
df = df[df['Comment']=='Completed']
```

[8]:
```
len(df)
```

[8]:
```
42261
```

3.33.2 Build graph

[9]:
```
edge_list = list(df[['Winner', 'Loser']].itertuples(index=False, name=None))
```

[10]:
```
len(edge_list)
```

[10]:
```
42261
```

[11]:
```
graph = from_edge_list(edge_list, directed=True)
```

[12]:
```
adjacency = graph.adjacency
names = graph.names
```

[13]:
```
adjacency
```

[13]:
```
<1255x1255 sparse matrix of type '<class 'numpy.int64'>'
        with 28212 stored elements in Compressed Sparse Row format>
```

[14]:
```
len(names)
```

[14]:
```
1255
```

3.33.3 Ranking

[15]:
```
# top-10 players in number of wins
out_weights = adjacency.dot(np.ones(len(names)))
print(names[top_k(out_weights, 10)])
```

```
['Federer R.' 'Nadal R.' 'Djokovic N.' 'Ferrer D.' 'Murray A.'
 'Roddick A.' 'Berdych T.' 'Robredo T.' 'Davydenko N.' 'Hewitt L.']
```

[16]:
```
# top-10 players in terms of PageRank
pagerank = PageRank()
adjacency_transpose = sparse.csr_matrix(adjacency.T)
scores = pagerank.fit_transform(adjacency_transpose)
print(names[top_k(scores, 10)])
```

```
['Federer R.' 'Nadal R.' 'Djokovic N.' 'Murray A.' 'Ferrer D.'
 'Roddick A.' 'Berdych T.' 'Hewitt L.' 'Davydenko N.' 'Wawrinka S.']
```

[17]:
```
index = top_k(scores, 10)
sub_adjacency = adjacency[index][:, index]
```

[18]:
```
SVG(svg_digraph(sub_adjacency, names=names[index], scores=scores[index]))
```

[18]:

3.33.4 Core decomposition

[19]:
```
algo = CoreDecomposition()
```

[20]:
```
adjacency_sym = directed2undirected(adjacency)
```

[21]:
```
labels = algo.fit_transform(adjacency_sym)
```

[22]:
```
print(names[labels == algo.core_value_])
```

```
['Acasuso J.' 'Almagro N.' 'Ancic M.' 'Anderson K.' 'Andreev I.'
 'Andujar P.' 'Baghdatis M.' 'Beck K.' 'Becker B.' 'Bellucci T.'
 'Benneteau J.' 'Berdych T.' 'Berlocq C.' 'Berrer M.' 'Bjorkman J.'
 'Blake J.' 'Bolelli S.' 'Calleri A.' 'Canas G.' 'Chardy J.' 'Chela J.I.'
 'Cilic M.' 'Clement A.' 'Coria G.' 'Cuevas P.' 'Darcis S.' 'Davydenko N.'
 'Del Potro J.M.' 'Dent T.' 'Dimitrov G.' 'Djokovic N.' 'Dodig I.'
 'Dolgopolov O.' 'Falla A.' 'Federer R.' 'Ferrer D.' 'Ferrero J.C.'
 'Fish M.' 'Fognini F.' 'Gabashvili T.' 'Garcia-Lopez G.' 'Gasquet R.'
 'Gaudio G.' 'Gicquel M.' 'Gimeno-Traver D.' 'Ginepri R.' 'Giraldo S.'
 'Golubev A.' 'Gonzalez F.' 'Granollers M.' 'Grosjean S.' 'Gulbis E.'
 'Haas T.' 'Haase R.' 'Hanescu V.' 'Harrison R.' 'Henman T.' 'Hernych J.'
 'Hewitt L.' 'Horna L.' 'Hrbaty D.' 'Isner J.' 'Istomin D.' 'Johansson T.'
 'Karlovic I.' 'Kiefer N.' 'Kohlschreiber P.' 'Korolev E.' 'Koubek S.'
 'Kubot L.' 'Kunitsyn I.' 'Lapentti N.' 'Lee H.T.' 'Ljubicic I.'
 'Llodra M.' 'Lopez F.' 'Lu Y.H.' 'Mahut N.' 'Malisse X.' 'Mannarino A.'
 'Martin A.' 'Massu N.' 'Mathieu P.H.' 'Mayer F.' 'Mayer L.' 'Melzer J.'
 'Mirnyi M.' 'Monaco J.' 'Monfils G.' 'Montanes A.' 'Moya C.' 'Muller G.'
```

(continues on next page)

(continued from previous page)

```
 'Murray A.' 'Nadal R.' 'Nalbandian D.' 'Nieminen J.' 'Nishikori K.'
 'Novak J.' 'Paire B.' 'Pavel A.' 'Petzschner P.' 'Phau B.' 'Querrey S.'
 'Ramirez-Hidalgo R.' 'Raonic M.' 'Robredo T.' 'Rochus C.' 'Rochus O.'
 'Roddick A.' 'Roger-Vasselin E.' 'Rosol L.' 'Russell M.' 'Safin M.'
 'Santoro F.' 'Schuettler R.' 'Sela D.' 'Seppi A.' 'Serra F.' 'Simon G.'
 'Soderling R.' 'Spadea V.' 'Srichaphan P.' 'Stakhovsky S.' 'Starace P.'
 'Stepanek R.' 'Tipsarevic J.' 'Tomic B.' 'Troicki V.' 'Tsonga J.W.'
 'Tursunov D.' 'Verdasco F.' 'Vliegen K.' 'Volandri F.' 'Wawrinka S.'
 'Young D.' 'Youzhny M.' 'Zverev M.']
```

3.33.5 Embedding

[23]:
```
spectral = Spectral(2, normalized=False)
```

[24]:
```
embedding = spectral.fit_transform(adjacency)
```

[25]:
```
index = np.argwhere(labels == algo.core_value_).ravel()
```

[26]:
```
SVG(svg_graph(position=embedding[index], names=names[index], scores=scores[index], node_
↪size=5, width=400, height=1000))
```

[26]:

3.34 Credits

The project started with the Master internship of Bertrand Charpentier and the PhD theses of Nathan de Lara and Quentin Lutz, under the supervision of Thomas Bonald at Télécom Paris, Institut Polytechnique de Paris.

3.34.1 Development Lead

- Thomas Bonald <thomas.bonald@telecom-paris.fr>
- Quentin Lutz <quentin.lutz@telecom-paris.fr>

3.34.2 Contributors

- Bertrand Charpentier
- Nathan de Lara
- Maximilien Danisch
- François Durand
- Alexandre Hollocou
- Fabien Mathieu
- Yohann Robert
- Julien Simonnet

- Alexis Barreaux
- Rémi Jaylet
- Victor Manach
- Pierre Pébereau
- Armand Boschin
- Tiphaine Viard
- Marc Jeanmougin
- Flávio Juvenal
- Wenzhuo Zhao
- Henry Carscadden

3.35 History

3.35.1 0.26.0 (2022-05-03)

- Add module on regression, by Thomas Bonald
- Add connected components for bipartite graphs, by Thomas Bonald
- Update functions for loading graphs, by Thomas Bonald
- Fix shuffling nodes in Louvain (issue #521), by Thomas Bonald
- Add radius and eccentricity metrics, by Henry Carscadden (#522)
- Add new use case (recommendation), by Thomas Bonald

3.35.2 0.25.0 (2022-03-15)

- Add use cases as notebooks, by Thomas Bonald
- Add list/dict of neighbors for building graphs, by Thomas Bonald
- Update Spectral embedding, by Thomas Bonald
- Update Block models, by Thomas Bonald (#507)
- Fix Tree sampling divergence, by Thomas Bonald (#505)
- Allow parsers to return weighted graphs, by Thomas Bonald
- Add Apple Silicon and Python 3.10 wheels, by Quentin Lutz (#503)

3.35.3 0.24.0 (2021-07-27)

- Merge Bi* algorithms (e.g., BiLouvain -> Louvain) by Thomas Bonald (#490)
- Transition from Travis to Github actions by Quentin Lutz (#488)
- Added sdist build for conda recipes
- Added name position for graph visualization
- Removed randomized algorithms

3.35.4 0.23.1 (2021-04-24)

- Updated NumPy and SciPy requirements

3.35.5 0.23.0 (2021-04-23)

- New push-based implementation of PageRank by Wenzhuo Zhao (#475)
- Fixed cut_balanced in hierarchy
- Dropped Python 3.6, wheels for Python 3.9 (switched to manylinux2014)

3.35.6 0.22.0 (2021-02-09)

- Added hierarchical Louvain embedding by Quentin Lutz (#468)
- Doc fixes and updates
- Requirements update

3.35.7 0.21.0 (2021-01-29)

- Added random projection embedding by Thomas Bonald (#461)
- Added PCA-based embedding by Thomas Bonald (#461)
- Added 64-bit support for Louvain by Flávio Juvenal (#450)
- Added verbosity options for dataset loaders
- Fixed Louvain embedding
- Various doc and tutorial updates

3.35.8 0.20.0 (2020-10-20)

- Added betweenness algorithm by Tiphaine Viard (#444)

3.35.9 0.19.3 (2020-09-17)

- Added Louvain-based embedding
- Fix documentation with new dataset website URLs

3.35.10 0.19.2 (2020-09-14)

- Fix documentation with new dataset website URLs.

3.35.11 0.19.1 (2020-09-09)

- Fix visualization features
- Fix documentation

3.35.12 0.19.0 (2020-09-02)

- Added link prediction module
- Added pie-node visualization of memberships
- Added Weisfeiler-Lehman graph coloring by Pierre Pebereau and Alexis Barreaux (#394)
- Added Force Atlas 2 graph layout by Victor Manach and Rémi Jaylet (#396)
- Added triangle listing algorithm for directed and undirected graph by Julien Simonnet and Yohann Robert (#376)
- Added k-core decomposition algorithm by Julien Simonnet and Yohann Robert (#377)
- Added k-clique listing algorithm by Julien Simonnet and Yohann Robert (#377)
- Added color map option in visualization module
- Updated NetSet URL

3.35.13 0.18.0 (2020-06-08)

- Added Katz centrality
- Refactor connectivity module into paths and topology
- Refactor Diffusion into Dirichlet
- Added parsers for adjacency list TSV and GraphML
- Added shortest paths and distances

3.35.14 0.17.0 (2020-05-07)

- Add clustering by label propagation
- Add models
- Add function to build graph from edge list
- Change a parameter in SVG visualization functions
- Minor corrections

3.35.15 0.16.0 (2020-04-30)

- Refactor basics module into connectivity
- Cython version for label propagation
- Minor corrections

3.35.16 0.15.2 (2020-04-24)

- Clarified requirements
- Minor corrections

3.35.17 0.15.1 (2020-04-21)

- Added OpenMP support for all platforms

3.35.18 0.15.0 (2020-04-20)

- Updated ranking module : new pagerank solver, new HITS params, post-processing
- Polynomes in linear algebra
- Added meta.name attribute for Bunch
- Minor corrections

3.35.19 0.14.0 (2020-04-17)

- Added spring layout in embedding
- Added label propagation in classification
- Added save / load functions in data
- Added display edges parameter in svg graph exports
- Corrected typos in documentation

3.35.20 0.13.3 (2020-04-13)

- Minor bug

3.35.21 0.13.2 (2020-04-13)

- Added wheels for multiple platforms (OSX, Windows (32 & 64 bits) and many Linux) and Python version (3.6/3.7/3.8)
- Documentation update (SVG dendrograms, tutorial updates)

3.35.22 0.13.1a (2020-04-09)

- Minor bug

3.35.23 0.13.0a (2020-04-09)

- Changed from Numba to Cython for better performance
- Added visualization module
- Added k-nearest neighbors classifier
- Added Louvain hierarchy
- Added predict method in embedding
- Added soft clustering to clustering algorithms
- Added soft classification to classification algorithms
- Added graphs in data module
- Various API change

3.35.24 0.12.1 (2020-01-20)

- Added heat kernel based node classifier
- Updated loaders for WikiLinks
- Fixed file-related issues for Windows

3.35.25 0.12.0 (2019-12-10)

- Added VerboseMixin for verbosity features
- Added Loaders for WikiLinks & Konect databases

3.35.26 0.11.0 (2019-11-28)

- sknetwork: new API for bipartite graphs
- new module: Soft node classification
- new module: Node classification
- new module: data (merge toy graphs + loader)
- clustering: Spectral Clustering
- ranking: new algorithms
- utils: K-neighbors
- hierarchy: Spectral WardDense
- data: loader (Vital Wikipedia)

3.35.27 0.10.1 (2019-08-26)

- Minor bug

3.35.28 0.10.0 (2019-08-26)

- Clustering (and related metrics) for directed and bipartite graphs
- Hierarchical clustering (and related metrics) for directed and bipartite graphs
- Fix bugs on embedding algorithms

3.35.29 0.9.0 (2019-07-24)

- Change parser output
- Fix bugs in ranking algorithms (zero-degree nodes)
- Add notebooks
- Import algorithms from scipy (shortest path, connected components, bfs/dfs)
- Change SVD embedding (now in decreasing order of singular values)

3.35.30 0.8.2 (2019-07-19)

- Minor bug

3.35.31 0.8.1 (2019-07-18)

- Added diffusion ranking
- Minor fixes
- Minor doc tweaking

3.35.32 0.8.0 (2019-07-17)

- Changed Louvain, BiLouvain, Paris and PageRank APIs
- Changed PageRank method
- Documentation overhaul
- Improved Jupyter tutorials

3.35.33 0.7.1 (2019-07-04)

- Added Algorithm class for nicer repr of some classes
- Added Jupyter notebooks as tutorials in the docs
- Minor fixes

3.35.34 0.7.0 (2019-06-24)

- Updated PageRank
- Added tests for Numba versioning

3.35.35 0.6.1 (2019-06-19)

- Minor bug

3.35.36 0.6.0 (2019-06-19)

- Largest connected component
- Simplex projection
- Sparse Low Rank Decomposition
- Numba support for Paris
- Various fixes and updates

3.35.37 0.5.0 (2019-04-18)

- Unified Louvain.

3.35.38 0.4.0 (2019-04-03)

- Added Louvain for directed graphs and ComboLouvain for bipartite graphs.

3.35.39 0.3.0 (2019-03-29)

- Updated clustering module and documentation.

3.35.40 0.2.0 (2019-03-21)

- First real release on PyPI.

3.35.41 0.1.1 (2018-05-29)

- First release on PyPI.

3.36 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

3.36.1 Types of Contributions

Report Bugs

Report bugs at https://github.com/sknetwork-team/sknetwork/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

Implement Features

Look through the GitHub projects. Anything listed in the projects is a feature to be implemented.

You can also look through GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

Write Documentation

scikit-network could always use more documentation, whether as part of the official scikit-network docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/sknetwork-team/sknetwork/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

3.36.2 Get Started!

Ready to contribute? Here's how to set up *sknetwork* for local development.

1. Fork the *sknetwork* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/sknetwork.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv sknetwork
$ cd sknetwork/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 sknetwork tests
$ python setup.py test or py.test
$ tox
```

To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

3.36.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 3.6, 3.7 and 3.8. Check https://travis-ci.org/sharpenb/sknetwork/pull_requests and make sure that the tests pass for all supported Python versions.

A more complete guide for writing code for the package can be found under "Contributing guide" in the Wiki.

3.36.4 Tips

To run a subset of tests:

```
$ py.test tests.test_sknetwork
```

Do not hesitate to check the Wiki.

3.36.5 Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in HISTORY.rst). Then run:

```
$ bumpversion patch # possible: major / minor / patch
$ git push
$ git push --tags
```

Travis will then deploy to PyPI if tests pass.

3.37 Index

3.38 Glossary

adjacency Square matrix whose entries indicate edges between nodes of a graph, usually denoted by A.

biadjacency Rectangular matrix whose entries indicate edges between nodes of a bipartite graph, usually denoted by B.

embedding Mapping of the nodes of a graph to points in a vector space.

INDEX

G

H

I

J

K

L

M

N

P

R

S

T

W