graphkit-learn Documentation

Release 1.0.0

Linlin Jia

Feb 28, 2024

CONTENTS

A Python package for graph kernels, graph edit distances and graph pre-image problem.

CHAPTER

ONE

REQUIREMENTS

- python>=3.6
- numpy>=1.16.2
- scipy>=1.1.0
- matplotlib>=3.1.0
- networkx>=2.2
- scikit-learn>=0.20.0
- tabulate>=0.8.2
- tqdm>=4.26.0
- control>=0.8.2 (for generalized random walk kernels only)
- slycot>=0.3.3 (for generalized random walk kernels only, which requires a fortran compiler (e.g., `gfortran`) and BLAS/LAPACK (e.g. `liblapack-dev`))
- Cython~=0.29.33 (for GEDLIB only)

CHAPTER

TWO

HOW TO USE?

2.1 Install the library

- Install stable version from PyPI (may not be up-to-date):

```
$ pip install graphkit-learn
```

- Install latest version from GitHub:

```
$ git clone https://github.com/jajupmochi/graphkit-learn.git
$ cd graphkit-learn/
$ python setup.py install
```

2.2 Run the test

A series of tests can be run to check if the library works correctly:

```
$ pip install -U pip pytest codecov coverage pytest-cov
$ pytest -v --cov-config=.coveragerc --cov-report term --cov=gklearn gklearn/tests/
```

2.3 Check examples

A series of demos of using the library can be found on Google Colab and in the ``*example*` <https://github.com/jajupmochi/graphkit-learn/tree/master/gklearn/examples>`_ folder.

2.4 Other demos

Check ``*notebooks*` <https://github.com/jajupmochi/graphkit-learn/tree/master/notebooks>`_ directory for more demos:

- ``*notebooks*` <https://github.com/jajupmochi/graphkit-learn/tree/master/notebooks>`_ directory includes test codes of graph kernels based on linear patterns;
- ``*notebooks/tests*` <https://github.com/jajupmochi/graphkit-learn/tree/master/notebooks/tests>`_ directory includes codes that test some libraries and functions;

- ``*notebooks/utils*` <https://github.com/jajupmochi/graphkit-learn/tree/master/notebooks/utils>`_ directory includes some useful tools, such as a Gram matrix checker and a function to get properties of datasets;
- ``*notebooks/else*` <https://github.com/jajupmochi/graphkit-learn/tree/master/notebooks/else>`_ directory includes other codes that we used for experiments.

2.5 Documentation

The docs of the library can be found here.

CHAPTER

THREE

MAIN CONTENTS

3.1 1 List of graph kernels

- Based on walks
  - The common walk kernel [1]
    - Exponential
    - Geometric
  - The marginalized kernel
    - With tottering [2]
    - Without tottering [7]
  - The generalized random walk kernel [3]
    - Sylvester equation
    - Conjugate gradient
    - Fixed-point iterations
    - Spectral decomposition
- Based on paths
  - The shortest path kernel [4]
  - The structural shortest path kernel [5]
  - The path kernel up to length h [6]
    - The Tanimoto kernel
    - The MinMax kernel
- Non-linear kernels
  - The treelet kernel [10]
  - Weisfeiler-Lehman kernel [11]
    - Subtree

A demo of computing graph kernels can be found on Google Colab and in the ``*examples*` <https://github.com/jajupmochi/graphkit-learn/blob/master/gklearn/examples/compute_graph_kernel.py>`_ folder.

3.2 2 Graph Edit Distances

3.3 3 Graph preimage methods

A demo of generating graph preimages can be found on Google Colab and in the ``*examples*` <https://github.com/jajupmochi/graphkit-learn/blob/master/gklearn/examples/median_preimege_generator.py>`_ folder.

3.4 4 Interface to `GEDLIB`

``*GEDLIB*` <https://github.com/dbblumenthal/gedlib>`_ is an easily extensible C++ library for (suboptimally) computing the graph edit distance between attributed graphs. A Python interface for `GEDLIB` is integrated in this library, based on ``*gedlibpy*` <https://github.com/Ryurin/gedlibpy>`_ library.

3.5 5 Computation optimization methods

- Python's `multiprocessing.Pool` module is applied to perform **parallelization** on the computations of all kernels as well as the model selection.
- **The Fast Computation of Shortest Path Kernel (FCSP) method** [8] is implemented in *the random walk kernel*, *the shortest path kernel*, as well as *the structural shortest path kernel* where FCSP is applied on both vertex and edge kernels.
- **The trie data structure** [9] is employed in *the path kernel up to length h* to store paths in graphs.

CHAPTER

FOUR

ISSUES

- This library uses `multiprocessing.Pool.imap_unordered` function to do the parallelization, which may not be able to run correctly under Windows system. For now, Windows users may need to comment the parallel codes and uncomment the codes below them which run serially. We will consider adding a parameter to control serial or parallel computations as needed.
- Some modules (such as `Numpy`, `Scipy`, `sklearn`) apply ``*OpenBLAS*` <https://www.openblas.net/>`_ to perform parallel computation by default, which causes conflicts with other parallelization modules such as `multiprossing.Pool`, highly increasing the computing time. By setting its thread to 1, `OpenBLAS` is forced to use a single thread/CPU, thus avoids the conflicts. For now, this procedure has to be done manually. Under Linux, type this command in terminal before running the code:

```
$ export OPENBLAS_NUM_THREADS=1
```

Or add `export OPENBLAS_NUM_THREADS=1` at the end of your `~/.bashrc` file, then run

```
$ source ~/.bashrc
```

to make this effective permanently.

CHAPTER

FIVE

RESULTS

Check this paper for detailed description of graph kernels and experimental results:

Linlin Jia, Benoit Gaüzère, and Paul Honeine. Graph Kernels Based on Linear Patterns: Theoretical and Experimental Comparisons. working paper or preprint, March 2019. URL https://hal-normandie-univ.archives-ouvertes.fr/hal-02053946.

A comparison of performances of graph kernels on benchmark datasets can be found here.

CHAPTER

SIX

HOW TO CONTRIBUTE

Fork the library and open a pull request! Make your own contribute to the community!

CHAPTER

SEVEN

AUTHORS

- Linlin Jia, LITIS, INSA Rouen Normandie
- Benoit Gaüzère, LITIS, INSA Rouen Normandie
- Paul Honeine, LITIS, Université de Rouen Normandie

CHAPTER

EIGHT

CITATION

If you have used `graphkit-learn` in your publication, please cite the the following paper:

```
@article{JIA2021,
    title = "graphkit-learn: A Python Library for Graph Kernels Based on Linear Patterns
↪",
    journal = "Pattern Recognition Letters",
    year = "2021",
    issn = "0167-8655",
    doi = "https://doi.org/10.1016/j.patrec.2021.01.003",
    url = "http://www.sciencedirect.com/science/article/pii/S0167865521000131",
    author = "Linlin Jia and Benoit Gaüzère and Paul Honeine",
    keywords = "Graph Kernels, Linear Patterns, Python Implementation",
    abstract = "This paper presents graphkit-learn, the first Python library for␣
↪efficient computation of graph kernels based on linear patterns, able to address␣
↪various types of graphs. Graph kernels based on linear patterns are thoroughly␣
↪implemented, each with specific computing methods, as well as two well-known graph␣
↪kernels based on non-linear patterns for comparative analysis. Since computational␣
↪complexity is an Achilles' heel of graph kernels, we provide several strategies to␣
↪address this critical issue, including parallelization, the trie data structure, and␣
↪the FCSP method that we extend to other kernels and edge comparison. All proposed␣
↪strategies save orders of magnitudes of computing time and memory usage. Moreover, all␣
↪the graph kernels can be simply computed with a single Python statement, thus are␣
↪appealing to researchers and practitioners. For the convenience of use, an advanced␣
↪model selection procedure is provided for both regression and classification problems.␣
↪Experiments on synthesized datasets and 11 real-world benchmark datasets show the␣
↪relevance of the proposed library."
}
```

CHAPTER

NINE

ACKNOWLEDGMENTS

This research was supported by CSC (China Scholarship Council) and the French national research agency (ANR) under the grant APi (ANR-18-CE23-0014). The authors would like to thank the CRIANN (Le Centre Régional Informatique et d'Applications Numériques de Normandie) for providing computational resources.

CHAPTER

TEN

REFERENCES

[1] Thomas Gärtner, Peter Flach, and Stefan Wrobel. On graph kernels: Hardness results and efficient alternatives. Learning Theory and Kernel Machines, pages 129–143, 2003.

[2] H. Kashima, K. Tsuda, and A. Inokuchi. Marginalized kernels between labeled graphs. In Proceedings of the 20th International Conference on Machine Learning, Washington, DC, United States, 2003.

[3] Vishwanathan, S.V.N., Schraudolph, N.N., Kondor, R., Borgwardt, K.M., 2010. Graph kernels. Journal of Machine Learning Research 11, 1201–1242.

[4] K. M. Borgwardt and H.-P. Kriegel. Shortest-path kernels on graphs. In Proceedings of the International Conference on Data Mining, pages 74-81, 2005.

[5] Liva Ralaivola, Sanjay J Swamidass, Hiroto Saigo, and Pierre Baldi. Graph kernels for chemical informatics. Neural networks, 18(8):1093–1110, 2005.

[6] Suard F, Rakotomamonjy A, Bensrhair A. Kernel on Bag of Paths For Measuring Similarity of Shapes. InESANN 2007 Apr 25 (pp. 355-360).

[7] Mahé, P., Ueda, N., Akutsu, T., Perret, J.L., Vert, J.P., 2004. Extensions of marginalized graph kernels, in: Proc. the twenty-first international conference on Machine learning, ACM. p. 70.

[8] Lifan Xu, Wei Wang, M Alvarez, John Cavazos, and Dongping Zhang. Parallelization of shortest path graph kernels on multi-core cpus and gpus. Proceedings of the Programmability Issues for Heterogeneous Multicores (MultiProg), Vienna, Austria, 2014.

[9] Edward Fredkin. Trie memory. Communications of the ACM, 3(9):490–499, 1960.

[10] Gaüzere, B., Brun, L., Villemin, D., 2012. Two new graphs kernels in chemoinformatics. Pattern Recognition Letters 33, 2038–2047.

[11] Shervashidze, N., Schweitzer, P., Leeuwen, E.J.v., Mehlhorn, K., Borgwardt, K.M., 2011. Weisfeiler-lehman graph kernels. Journal of Machine Learning Research 12, 2539–2561.

CHAPTER

ELEVEN

DOCUMENTATION

11.1 Modules

11.1.1 gklearn

gklearn

This package contains 4 sub packages :

- c_ext : binders to C++ code
- ged : allows to compute graph edit distance between networkX graphs
- kernels : computation of graph kernels, ie graph similarity measure compatible with SVM
- notebooks : examples of code using this library
- utils : Diverse computation on graphs

gklearn.kernels

gklearn - graph kernels module

gklearn.kernels.commonWalkKernel

@author: linlin

@references:

> [1] Thomas Gärtner, Peter Flach, and Stefan Wrobel. On graph kernels: Hardness results and efficient alternatives. Learning Theory and Kernel Machines, pages 129–143, 2003.

commonwalkkernel(**args*, *node_label='atom'*, *edge_label='bond_type'*, *weight=1*, *compute_method=None*, *n_jobs=None*, *chunksize=None*, *verbose=True*)

Compute common walk graph kernels between graphs.

Parameters

Gn
[List of NetworkX graph] List of graphs between which the kernels are computed.

G1, G2
[NetworkX graphs] Two graphs between which the kernel is computed.

node_label
[string] Node attribute used as symbolic label. The default node label is 'atom'.

edge_label
[string] Edge attribute used as symbolic label. The default edge label is 'bond_type'.

weight: integer
Weight coefficient of different lengths of walks, which represents beta in 'exp' method and gamma in 'geo'.

compute_method
[string] Method used to compute walk kernel. The Following choices are available:

'exp': method based on exponential serials applied on the direct product graph, as shown in reference [1]. The time complexity is O(n^6) for graphs with n vertices.

'geo': method based on geometric serials applied on the direct product graph, as shown in reference [1]. The time complexity is O(n^6) for graphs with n vertices.

n_jobs
[int] Number of jobs for parallelization.

Return

Kmatrix
[Numpy matrix] Kernel matrix, each element of which is a common walk kernel between 2 graphs.

`find_all_walks`(*G*, *length*)

Find all walks with a certain length in a graph. A recursive depth first search is applied.

Parameters

G
[NetworkX graphs] The graph in which walks are searched.

length
[integer] The length of walks.

Return

walk
[list of list] List of walks retrieved, where each walk is represented by a list of nodes.

`find_all_walks_until_length`(*G*, *length*, *node_label='atom'*, *edge_label='bond_type'*, *labeled=True*)

Find all walks with a certain maximum length in a graph. A recursive depth first search is applied.

Parameters

G
[NetworkX graphs] The graph in which walks are searched.

length
[integer] The maximum length of walks.

node_label
[string] node attribute used as label. The default node label is atom.

edge_label
[string] edge attribute used as label. The default edge label is bond_type.

labeled
[boolean] Whether the graphs are labeled. The default is True.

Return

walk
[list] List of walks retrieved, where for unlabeled graphs, each walk is represented by a list of nodes; while for labeled graphs, each walk is represented by a string consists of labels of nodes and edges on that walk.

`find_walks`(*G*, *source_node*, *length*)
Find all walks with a certain length those start from a source node. A recursive depth first search is applied.

Parameters

G
[NetworkX graphs] The graph in which walks are searched.

source_node
[integer] The number of the node from where all walks start.

length
[integer] The length of walks.

Return

walk
[list of list] List of walks retrieved, where each walk is represented by a list of nodes.

`wrapper_cw_exp`(*node_label*, *edge_label*, *beta*, *itr*)

`wrapper_cw_geo`(*node_label*, *edge_label*, *gama*, *itr*)

gklearn.kernels.marginalizedKernel

@author: linlin

@references:

[1] H. Kashima, K. Tsuda, and A. Inokuchi. Marginalized kernels between labeled graphs. In Proceedings of the 20th International Conference on Machine Learning, Washington, DC, United States, 2003.

[2] Pierre Mahé, Nobuhisa Ueda, Tatsuya Akutsu, Jean-Luc Perret, and Jean-Philippe Vert. Extensions of marginalized graph kernels. In Proceedings of the twenty-first international conference on Machine learning, page 70. ACM, 2004.

marginalizedkernel(**args*, *node_label='atom'*, *edge_label='bond_type'*, *p_quit=0.5*, *n_iteration=20*, *remove_totters=False*, *n_jobs=None*, *chunksize=None*, *verbose=True*)

Compute marginalized graph kernels between graphs.

Parameters

Gn
[List of NetworkX graph] List of graphs between which the kernels are computed.

G1, G2
[NetworkX graphs] Two graphs between which the kernel is computed.

node_label
[string] Node attribute used as symbolic label. The default node label is 'atom'.

edge_label
[string] Edge attribute used as symbolic label. The default edge label is 'bond_type'.

p_quit
[integer] The termination probability in the random walks generating step.

n_iteration
[integer] Time of iterations to compute R_inf.

remove_totters
[boolean] Whether to remove totterings by method introduced in [2]. The default value is False.

n_jobs
[int] Number of jobs for parallelization.

Return

Kmatrix
[Numpy matrix] Kernel matrix, each element of which is the marginalized kernel between 2 praphs.

wrapper_marg_do(*node_label*, *edge_label*, *p_quit*, *n_iteration*, *itr*)

wrapper_untotter(*Gn*, *node_label*, *edge_label*, *i*)

gklearn.kernels.randomWalkKernel

@author: linlin

@references:

[1] S Vichy N Vishwanathan, Nicol N Schraudolph, Risi Kondor, and Karsten M Borgwardt. Graph kernels. Journal of Machine Learning Research, 11(Apr):1201–1242, 2010.

computeVK(*g1*, *g2*, *ds_attrs*, *node_kernels*, *node_label*)

Compute vertex kernels between vertices of two graphs.

computeW(*g1*, *g2*, *vk_dict*, *ds_attrs*, *edge_kernels*, *edge_label*)

Compute the weight matrix of the direct product graph.

filterGramMatrix(*gmt*, *label_dict*, *label*, *directed*)

Compute (the transpose of) the Gram matrix filtered by a label.

func_fp(*x*, *p_times*, *lmda*, *w_times*)

getLabels(*Gn*, *node_label*, *edge_label*, *directed*)

Get symbolic labels of a graph dataset, where vertex labels are dealt with by concatenating them to the edge labels of adjacent edges.

randomwalkkernel(**args*, *compute_method=None*, *weight=1*, *p=None*, *q=None*, *edge_weight=None*, *node_kernels=None*, *edge_kernels=None*, *node_label='atom'*, *edge_label='bond_type'*, *sub_kernel=None*, *n_jobs=None*, *chunksize=None*, *verbose=True*)

Compute random walk graph kernels.

Parameters

Gn
[List of NetworkX graph] List of graphs between which the kernels are computed.

G1, G2
[NetworkX graphs] Two graphs between which the kernel is computed.

compute_method
[string] Method used to compute kernel. The Following choices are available:

'sylvester' - Sylvester equation method.

'conjugate' - conjugate gradient method.

'fp' - fixed-point iterations.

'spectral' - spectral decomposition.

weight
[float] A constant weight set for random walks of length h.

p
[None] Initial probability distribution on the unlabeled direct product graph of two graphs. It is set to be uniform over all vertices in the direct product graph.

q
[None] Stopping probability distribution on the unlabeled direct product graph of two graphs. It is set to be uniform over all vertices in the direct product graph.

edge_weight : float

Edge attribute name corresponding to the edge weight.

node_kernels: dict
A dictionary of kernel functions for nodes, including 3 items: 'symb' for symbolic node labels, 'nsymb' for non-symbolic node labels, 'mix' for both labels. The first 2 functions take two node labels as parameters, and the 'mix' function takes 4 parameters, a symbolic and a non-symbolic label for each the two nodes. Each label is in form of 2-D dimension array (n_samples, n_features). Each function returns a number as the kernel value. Ignored when nodes are unlabeled. This argument is designated to conjugate gradient method and fixed-point iterations.

edge_kernels: dict
A dictionary of kernel functions for edges, including 3 items: 'symb' for symbolic edge labels, 'nsymb' for non-symbolic edge labels, 'mix' for both labels. The first 2 functions take two edge labels as parameters, and the 'mix' function takes 4 parameters, a symbolic and a non-symbolic label for each the two edges. Each label is in form of 2-D dimension array (n_samples, n_features). Each function returns a number as the kernel value. Ignored when edges are unlabeled. This argument is designated to conjugate gradient method and fixed-point iterations.

node_label: string
Node attribute used as label. The default node label is atom. This argument is designated to conjugate gradient method and fixed-point iterations.

edge_label
[string] Edge attribute used as label. The default edge label is bond_type. This argument is designated to conjugate gradient method and fixed-point iterations.

sub_kernel: string
Method used to compute walk kernel. The Following choices are available: 'exp' : method based on exponential serials. 'geo' : method based on geometric serials.

n_jobs: int
Number of jobs for parallelization.

Return

Kmatrix
[Numpy matrix] Kernel matrix, each element of which is the path kernel up to d between 2 praphs.

`wrapper_cg_labeled_do`(*ds_attrs*, *node_kernels*, *node_label*, *edge_kernels*, *edge_label*, *lmda*, *itr*)

`wrapper_cg_unlabled_do`(*lmda*, *itr*)

`wrapper_fp_labeled_do`(*ds_attrs*, *node_kernels*, *node_label*, *edge_kernels*, *edge_label*, *lmda*, *itr*)

`wrapper_sd_do`(*weight*, *sub_kernel*, *itr*)

`wrapper_se_do`(*lmda*, *itr*)

gklearn.kernels.spKernel

@author: linlin

@references:

[1] Borgwardt KM, Kriegel HP. Shortest-path kernels on graphs. InData Mining, Fifth IEEE International Conference on 2005 Nov 27 (pp. 8-pp). IEEE.

spkernel(**args*, *node_label='atom'*, *edge_weight=None*, *node_kernels=None*, *parallel='imap_unordered'*, *n_jobs=None*, *chunksize=None*, *verbose=True*)

Compute shortest-path kernels between graphs.

Parameters

Gn
[List of NetworkX graph] List of graphs between which the kernels are computed.

G1, G2
[NetworkX graphs] Two graphs between which the kernel is computed.

node_label
[string] Node attribute used as label. The default node label is atom.

edge_weight
[string] Edge attribute name corresponding to the edge weight.

node_kernels
[dict] A dictionary of kernel functions for nodes, including 3 items: 'symb' for symbolic node labels, 'nsymb' for non-symbolic node labels, 'mix' for both labels. The first 2 functions take two node labels as parameters, and the 'mix' function takes 4 parameters, a symbolic and a non-symbolic label for each the two nodes. Each label is in form of 2-D dimension array (n_samples, n_features). Each function returns an number as the kernel value. Ignored when nodes are unlabeled.

n_jobs
[int] Number of jobs for parallelization.

Return

Kmatrix
[Numpy matrix] Kernel matrix, each element of which is the sp kernel between 2 praphs.

spkernel_do(*g1*, *g2*, *ds_attrs*, *node_label*, *node_kernels*)

wrapper_getSPGraph(*weight*, *itr_item*)

wrapper_sp_do(*ds_attrs*, *node_label*, *node_kernels*, *itr*)

gklearn.kernels.structuralspKernel

Created on Thu Sep 27 10:56:23 2018

@author: linlin

@references:

[1] Suard F, Rakotomamonjy A, Bensrhair A. Kernel on Bag of Paths For Measuring Similarity of Shapes. InESANN 2007 Apr 25 (pp. 355-360).

`getAllEdgeKernels`(*g1*, *g2*, *edge_kernels*, *edge_label*, *ds_attrs*)

`getAllNodeKernels`(*g1*, *g2*, *node_kernels*, *node_label*, *ds_attrs*)

`get_shortest_paths`(*G*, *weight*, *directed*)

Get all shortest paths of a graph.

Parameters

G
[NetworkX graphs] The graphs whose paths are computed.

weight
[string/None] edge attribute used as weight to compute the shortest path.

directed: boolean
Whether graph is directed.

Return

sp
[list of list] List of shortest paths of the graph, where each path is represented by a list of nodes.

`get_sps_as_trie`(*G*, *weight*, *directed*)

Get all shortest paths of a graph and insert them into a trie.

Parameters

G
[NetworkX graphs] The graphs whose paths are computed.

weight
[string/None] edge attribute used as weight to compute the shortest path.

directed: boolean
Whether graph is directed.

Return

sp
[list of list] List of shortest paths of the graph, where each path is represented by a list of nodes.

`ssp_do_trie`(*g1*, *g2*, *trie1*, *trie2*, *ds_attrs*, *node_label*, *edge_label*, *node_kernels*, *edge_kernels*)

`structuralspkernel`(**args*, *node_label='atom'*, *edge_weight=None*, *edge_label='bond_type'*, *node_kernels=None*, *edge_kernels=None*, *compute_method='naive'*, *parallel='imap_unordered'*, *n_jobs=None*, *chunksize=None*, *verbose=True*)

Compute mean average structural shortest path kernels between graphs.

Parameters

Gn
[List of NetworkX graph] List of graphs between which the kernels are computed.

G1, G2
[NetworkX graphs] Two graphs between which the kernel is computed.

node_label
[string] Node attribute used as label. The default node label is atom.

edge_weight
[string] Edge attribute name corresponding to the edge weight. Applied for the computation of the shortest paths.

edge_label
[string] Edge attribute used as label. The default edge label is bond_type.

node_kernels
[dict] A dictionary of kernel functions for nodes, including 3 items: 'symb' for symbolic node labels, 'nsymb' for non-symbolic node labels, 'mix' for both labels. The first 2 functions take two node labels as parameters, and the 'mix' function takes 4 parameters, a symbolic and a non-symbolic label for each the two nodes. Each label is in form of 2-D dimension array (n_samples, n_features). Each function returns a number as the kernel value. Ignored when nodes are unlabeled.

edge_kernels
[dict] A dictionary of kernel functions for edges, including 3 items: 'symb' for symbolic edge labels, 'nsymb' for non-symbolic edge labels, 'mix' for both labels. The first 2 functions take two edge labels as parameters, and the 'mix' function takes 4 parameters, a symbolic and a non-symbolic label for each the two edges. Each label is in form of 2-D dimension array (n_samples, n_features). Each function returns a number as the kernel value. Ignored when edges are unlabeled.

compute_method
[string] Computation method to store the shortest paths and compute the graph kernel. The Following choices are available:

'trie': store paths as tries.

'naive': store paths to lists.

n_jobs
[int] Number of jobs for parallelization.

Return

Kmatrix
[Numpy matrix] Kernel matrix, each element of which is the mean average structural shortest path kernel between 2 praphs.

structuralspkernel_do(*g1*, *g2*, *spl1*, *spl2*, *ds_attrs*, *node_label*, *edge_label*, *node_kernels*, *edge_kernels*)

traverseBothTriee(*root*, *trie2*, *kernel*, *vk_dict*, *ek_dict*, *pcurrent=[]*)

traverseBothTriem(*root*, *trie2*, *kernel*, *vk_dict*, *ek_dict*, *pcurrent=[]*)

traverseBothTrieu(*root*, *trie2*, *kernel*, *vk_dict*, *ek_dict*, *pcurrent=[]*)

traverseBothTriev(*root*, *trie2*, *kernel*, *vk_dict*, *ek_dict*, *pcurrent=[]*)

traverseTrie2e(*root*, *p1*, *kernel*, *vk_dict*, *ek_dict*, *pcurrent=[]*)

traverseTrie2m(*root*, *p1*, *kernel*, *vk_dict*, *ek_dict*, *pcurrent=[]*)

traverseTrie2u(*root*, *p1*, *kernel*, *vk_dict*, *ek_dict*, *pcurrent=[]*)

traverseTrie2v(*root*, *p1*, *kernel*, *vk_dict*, *ek_dict*, *pcurrent=[]*)

wrapper_getSP_naive(*weight*, *directed*, *itr_item*)

wrapper_getSP_trie(*weight*, *directed*, *itr_item*)

wrapper_ssp_do(*ds_attrs*, *node_label*, *edge_label*, *node_kernels*, *edge_kernels*, *itr*)

wrapper_ssp_do_trie(*ds_attrs*, *node_label*, *edge_label*, *node_kernels*, *edge_kernels*, *itr*)

gklearn.kernels.treeletKernel

@author: linlin

@references:

[1] Gaüzère B, Brun L, Villemin D. Two new graphs kernels in chemoinformatics. Pattern Recognition Letters. 2012 Nov 1;33(15):2038-47.

find_all_paths(*G*, *length*, *is_directed*)
Find all paths with a certain length in a graph. A recursive depth first search is applied.

Parameters

G
[NetworkX graphs] The graph in which paths are searched.

length
[integer] The length of paths.

Return

path
[list of list] List of paths retrieved, where each path is represented by a list of nodes.

`find_paths`(*G*, *source_node*, *length*)
Find all paths with a certain length those start from a source node. A recursive depth first search is applied.

Parameters

G
[NetworkX graphs] The graph in which paths are searched.

source_node
[integer] The number of the node from where all paths start.

length
[integer] The length of paths.

Return

path
[list of list] List of paths retrieved, where each path is represented by a list of nodes.

`get_canonkeys`(*G*, *node_label*, *edge_label*, *labeled*, *is_directed*)
Generate canonical keys of all treelets in a graph.

Parameters

G
[NetworkX graphs] The graph in which keys are generated.

node_label
[string] node attribute used as label. The default node label is atom.

edge_label
[string] edge attribute used as label. The default edge label is bond_type.

labeled
[boolean] Whether the graphs are labeled. The default is True.

Return

canonkey/canonkey_l
[dict] For unlabeled graphs, canonkey is a dictionary which records amount of every tree pattern. For labeled graphs, canonkey_l is one which keeps track of amount of every treelet.

`treeletkernel`(**args*, *sub_kernel*, *node_label='atom'*, *edge_label='bond_type'*, *parallel='imap_unordered'*, *n_jobs=None*, *chunksize=None*, *verbose=True*)
Compute treelet graph kernels between graphs.

Parameters

Gn
[List of NetworkX graph] List of graphs between which the kernels are computed.

G1, G2
[NetworkX graphs] Two graphs between which the kernel is computed.

sub_kernel
[function] The sub-kernel between 2 real number vectors. Each vector counts the numbers of isomorphic treelets in a graph.

node_label
[string] Node attribute used as label. The default node label is atom.

edge_label
[string] Edge attribute used as label. The default edge label is bond_type.

parallel
[string/None] Which parallelizton method is applied to compute the kernel. The Following choices are available:

'imap_unordered': use Python's multiprocessing.Pool.imap_unordered method.

None: no parallelization is applied.

n_jobs
[int] Number of jobs for parallelization. The default is to use all computational cores. This argument is only valid when one of the parallelization method is applied.

Return

Kmatrix
[Numpy matrix] Kernel matrix, each element of which is the treelet kernel between 2 praphs.

wrapper_get_canonkeys(*node_label*, *edge_label*, *labeled*, *is_directed*, *itr_item*)

wrapper_treeletkernel_do(*sub_kernel*, *itr*)

gklearn.kernels.untilHPathKernel

@author: linlin

@references:

[1] Liva Ralaivola, Sanjay J Swamidass, Hiroto Saigo, and Pierre Baldi. Graph kernels for chemical informatics. Neural networks, 18(8):1093–1110, 2005.

find_all_path_as_trie(*G*, *length*, *ds_attrs*, *node_label='atom'*, *edge_label='bond_type'*)

find_all_paths_until_length(*G*, *length*, *ds_attrs*, *node_label='atom'*, *edge_label='bond_type'*, *tolabelseqs=True*)

Find all paths no longer than a certain maximum length in a graph. A recursive depth first search is applied.

Parameters

G
[NetworkX graphs] The graph in which paths are searched.

length
[integer] The maximum length of paths.

ds_attrs: dict
Dataset attributes.

node_label
[string] Node attribute used as label. The default node label is atom.

edge_label
[string] Edge attribute used as label. The default edge label is bond_type.

Return

path
[list] List of paths retrieved, where for unlabeled graphs, each path is represented by a list of nodes; while for labeled graphs, each path is represented by a list of strings consists of labels of nodes and/or edges on that path.

`paths2labelseqs`(*plist*, *G*, *ds_attrs*, *node_label*, *edge_label*)

`untilhpathkernel`(**args*, *node_label='atom'*, *edge_label='bond_type'*, *depth=10*, *k_func='MinMax'*, *compute_method='trie'*, *parallel='imap_unordered'*, *n_jobs=None*, *chunksize=None*, *verbose=True*)

Compute path graph kernels up to depth/hight h between graphs.

Parameters

Gn
[List of NetworkX graph] List of graphs between which the kernels are computed.

G1, G2
[NetworkX graphs] Two graphs between which the kernel is computed.

node_label
[string] Node attribute used as label. The default node label is atom.

edge_label
[string] Edge attribute used as label. The default edge label is bond_type.

depth
[integer] Depth of search. Longest length of paths.

k_func
[function] A kernel function applied using different notions of fingerprint similarity, defining the type of feature map and normalization method applied for the graph kernel. The Following choices are available:

'MinMax': use the MiniMax kernel and counting feature map.

'tanimoto': use the Tanimoto kernel and binary feature map.

None: no sub-kernel is used, the kernel is computed directly.

compute_method
[string] Computation method to store paths and compute the graph kernel. The Following choices are available:

'trie': store paths as tries.

'naive': store paths to lists.

n_jobs
[int] Number of jobs for parallelization.

Return

Kmatrix
[Numpy matrix] Kernel matrix, each element of which is the path kernel up to h between 2 praphs.

`wrapper_find_all_path_as_trie`(*length*, *ds_attrs*, *node_label*, *edge_label*, *itr_item*)

`wrapper_find_all_paths_until_length`(*length*, *ds_attrs*, *node_label*, *edge_label*, *tolabelseqs*, *itr_item*)

`wrapper_uhpath_do_kernelless`(*k_func*, *itr*)

`wrapper_uhpath_do_naive`(*k_func*, *itr*)

`wrapper_uhpath_do_trie`(*k_func*, *itr*)

gklearn.kernels.weisfeilerLehmanKernel

@author: linlin

@references:

[1] Shervashidze N, Schweitzer P, Leeuwen EJ, Mehlhorn K, Borgwardt KM. Weisfeiler-lehman graph kernels. Journal of Machine Learning Research. 2011;12(Sep):2539-61.

`compute_kernel_matrix`(*Kmatrix*, *all_num_of_each_label*, *Gn*, *parallel*, *n_jobs*, *chunksize*, *verbose*)
Compute kernel matrix using the base kernel.

`compute_subtree_kernel`(*num_of_each_label1*, *num_of_each_label2*, *kernel*)
Compute the subtree kernel.

`weisfeilerlehmankernel`(**args*, *node_label='atom'*, *edge_label='bond_type'*, *height=0*, *base_kernel='subtree'*, *parallel=None*, *n_jobs=None*, *chunksize=None*, *verbose=True*)
Compute Weisfeiler-Lehman kernels between graphs.

Parameters

Gn
[List of NetworkX graph] List of graphs between which the kernels are computed.

G1, G2
[NetworkX graphs] Two graphs between which the kernel is computed.

node_label
[string] Node attribute used as label. The default node label is atom.

edge_label
[string] Edge attribute used as label. The default edge label is bond_type.

height
[int] Subtree height.

base_kernel
[string] Base kernel used in each iteration of WL kernel. Only default 'subtree' kernel can be applied for now.

parallel
[None] Which paralleliztion method is applied to compute the kernel. No parallelization can be applied for now.

n_jobs
[int] Number of jobs for parallelization. The default is to use all computational cores. This argument is only valid when one of the parallelization method is applied and can be ignored for now.

Return

Kmatrix
[Numpy matrix] Kernel matrix, each element of which is the Weisfeiler-Lehman kernel between 2 praphs.

Notes

This function now supports WL subtree kernel only.

`wl_iteration`(*G*, *node_label*)

`wrapper_compute_subtree_kernel`(*Kmatrix*, *itr*)

`wrapper_wl_iteration`(*node_label*, *itr_item*)

gklearn.utils

gklearn - utils module

Implement some methods to manage graphs
graphfiles.py : load .gxl and .ct files utils.py : compute some properties on networkX graphs

gklearn.utils.graphdataset

Obtain all kinds of attributes of a graph dataset.

This file is for old version of graphkit-learn.

`get_dataset_attributes`(*Gn*, *target=None*, *attr_names=[]*, *node_label=None*, *edge_label=None*)
Returns the structure and property information of the graph dataset Gn.

Parameters

Gn
[List of NetworkX graph] List of graphs whose information will be returned.

target
[list] The list of classification targets corresponding to Gn. Only works for classification problems.

attr_names
[list] List of strings which indicate which informations will be returned. The possible choices includes:

'substructures': sub-structures Gn contains, including 'linear', 'non

linear' and 'cyclic'.

'node_labeled': whether vertices have symbolic labels.

'edge_labeled': whether egdes have symbolic labels.

'is_directed': whether graphs in Gn are directed.

'dataset_size': number of graphs in Gn.

'ave_node_num': average number of vertices of graphs in Gn.

'min_node_num': minimum number of vertices of graphs in Gn.

'max_node_num': maximum number of vertices of graphs in Gn.

'ave_edge_num': average number of edges of graphs in Gn.

'min_edge_num': minimum number of edges of graphs in Gn.

'max_edge_num': maximum number of edges of graphs in Gn.

'ave_node_degree': average vertex degree of graphs in Gn.

'min_node_degree': minimum vertex degree of graphs in Gn.

'max_node_degree': maximum vertex degree of graphs in Gn.

'ave_fill_factor': average fill factor (number_of_edges /

(number_of_nodes ** 2)) of graphs in Gn.

'min_fill_factor': minimum fill factor of graphs in Gn.

'max_fill_factor': maximum fill factor of graphs in Gn.

'node_label_num': number of symbolic vertex labels.

'edge_label_num': number of symbolic edge labels.

'node_attr_dim': number of dimensions of non-symbolic vertex labels.

Extracted from the 'attributes' attribute of graph nodes.

'edge_attr_dim': number of dimensions of non-symbolic edge labels.

Extracted from the 'attributes' attribute of graph edges.

'class_number': number of classes. Only available for classification problems.

node_label
[string] Node attribute used as label. The default node label is atom. Mandatory when 'node_labeled' or 'node_label_num' is required.

edge_label
[string] Edge attribute used as label. The default edge label is bond_type. Mandatory when 'edge_labeled' or 'edge_label_num' is required.

Return

attrs
[dict] Value for each property.

`load_predefined_dataset`(*ds_name*)

gklearn.utils.graphfiles

Utilities function to manage graph files

`loadCT`(*filename*)
load data from a Chemical Table (.ct) file.

Notes

a typical example of data in .ct is like this:

3 2 <- number of nodes and edges

0.0000 0.0000 0.0000 C <- each line describes a node (x,y,z + label)

0.0000 0.0000 0.0000 C

0.0000 0.0000 0.0000 O

1 3 1 1 <- each line describes an edge : to, from, bond type, bond stereo

2 3 1 1

Check CTFile Formats file for detailed format discription.

`loadDataset`(*filename*, *filename_y=None*, *extra_params=None*)
Read graph data from filename and load them as NetworkX graphs.

Parameters

filename
[string] The name of the file from where the dataset is read.

filename_y
[string] The name of file of the targets corresponding to graphs.

extra_params
[dict] Extra parameters only designated to '.mat' format.

Return

data : List of NetworkX graph.

y : List

Targets corresponding to graphs.

Notes

This function supports following graph dataset formats:

'ds': load data from .ds file. See comments of function loadFromDS for a example.

'cxl': load data from Graph eXchange Language file (.cxl file). See here for detail.

'sdf': load data from structured data file (.sdf file). See here for details.

'mat': Load graph data from a MATLAB (up to version 7.1) .mat file. See README in downloadable file for details.

'txt': Load graph data from a special .txt file. See here for details. Note here filename is the name of either .txt file in the dataset directory.

loadFromDS(*filename*, *filename_y*)

Load data from .ds file.

Possible graph formats include:

'.ct': see function loadCT for detail.

'.gxl': see dunction loadGXL for detail.

Note these graph formats are checked automatically by the extensions of graph files.

loadFromXML(*filename*, *extra_params*)

loadGXL(*filename*)

loadMAT(*filename*, *extra_params*)

Load graph data from a MATLAB (up to version 7.1) .mat file.

Notes

A MAT file contains a struct array containing graphs, and a column vector lx containing a class label for each graph. Check README in downloadable file for detailed structure.

loadSDF(*filename*)

load data from structured data file (.sdf file).

Notes

A SDF file contains a group of molecules, represented in the similar way as in MOL format. Check here for detailed structure.

loadTXT(*filename*)

Load graph data from a .txt file.

Notes

The graph data is loaded from separate files. Check README in downloadable file, 2018 for detailed structure.

saveDataset(*Gn*, *y*, *gformat='gxl'*, *group=None*, *filename='gfile'*, *xparams=None*)

Save list of graphs.

saveGXL(*graph*, *filename*, *method='default'*, *node_labels=[]*, *edge_labels=[]*, *node_attrs=[]*, *edge_attrs=[]*)

gklearn.utils.kernels

Those who are not graph kernels. We can be kernels for nodes or edges! These kernels are defined between pairs of vectors.

chi2_kernel(*x*, *y*, *gamma=1.0*)

cosine_kernel(*x*, *y*)

delta_kernel(*x*, *y*)

deltakernel(*x*, *y*)

exponential_kernel(*x*, *y*, *gamma=None*)

gaussian_kernel(*x*, *y*, *gamma=None*)

Gaussian kernel. Compute the rbf (gaussian) kernel between x and y:

K(x, y) = exp(-gamma ||x-y||^2).

Read more in the User Guide of scikit-learn library.

Parameters

x, y : array

gamma

[float, default None] If None, defaults to 1.0 / n_features

Returns

kernel : float

gaussiankernel(*x*, *y*, *gamma=None*)

highest_polynomial_kernel(*x*, *y*, *d=1*, *c=0*)

Polynomial kernel. Compute the polynomial kernel between x and y:

K(x, y) = <x, y> ^d + c.

Parameters

x, y : array

d : integer, default 1

c : float, default 0

Returns

kernel : float

intersection_kernel(*x*, *y*)

inverse_multiquadratic_kernel(*x*, *y*, *c=0*)

kernelproduct(*k1*, *k2*, *d11*, *d12*, *d21=None*, *d22=None*, *lamda=1*)

Product of a pair of kernels.

k = lamda * k1(d11, d12) * k2(d21, d22)

Parameters

k1, k2
: [function] A pair of kernel functions.

d11, d12:
: Inputs of k1. If d21 or d22 is None, apply d11, d12 to both k1 and k2.

d21, d22:
: Inputs of k2.

lamda: float
: Coefficient of the product.

Return

kernel : integer

kernelsum(*k1*, *k2*, *d11*, *d12*, *d21=None*, *d22=None*, *lamda1=1*, *lamda2=1*)

Sum of a pair of kernels.

k = lamda1 * k1(d11, d12) + lamda2 * k2(d21, d22)

Parameters

k1, k2
[function] A pair of kernel functions.

d11, d12:
Inputs of k1. If d21 or d22 is None, apply d11, d12 to both k1 and k2.

d21, d22:
Inputs of k2.

lamda1, lamda2: float
Coefficients of the product.

Return

kernel : integer

kronecker_delta_kernel(*x*, *y*)

Delta kernel. Return 1 if x == y, 0 otherwise.

Parameters

x, y
[any] Two parts to compare.

Return

kernel
[integer] Delta kernel.

References

[1] H. Kashima, K. Tsuda, and A. Inokuchi. Marginalized kernels between labeled graphs. In Proceedings of the 20th International Conference on Machine Learning, Washington, DC, United States, 2003.

laplacian_kernel(*x*, *y*, *gamma=None*)

linear_kernel(*x*, *y*)

Polynomial kernel. Compute the polynomial kernel between x and y:

K(x, y) = <x, y>.

Parameters

x, y : array

d : integer, default 1

c : float, default 0

Returns

kernel : float

linearkernel(*x*, *y*)

multiquadratic_kernel(*x*, *y*, *c=0*)

polynomial_kernel(*x*, *y*, *gamma=1*, *coef0=0*, *d=1*)

polynomialkernel(*x*, *y*, *d=1*, *c=0*)

sigmoid_kernel(*x*, *y*, *gamma=None*, *coef0=1*)

tanimoto_kernel(*x*, *y*)

gklearn.utils.model_selection_precomputed

compute_gram_matrices(*dataset*, *y*, *estimator*, *param_list_precomputed*, *output_dir*, *ds_name*, *n_jobs=1*, *str_fw=''*, *verbose=True*)

model_selection_for_precomputed_kernel(*datafile*, *estimator*, *param_grid_precomputed*, *param_grid*, *model_type*, *NUM_TRIALS=30*, *datafile_y=None*, *extra_params=None*, *ds_name='ds-unknown'*, *output_dir='outputs/'*, *n_jobs=1*, *read_gm_from_file=False*, *verbose=True*)

Perform model selection, fitting and testing for precomputed kernels using nested CV. Print out neccessary data during the process then finally the results.

Parameters

datafile
[string] Path of dataset file.

estimator
[function] kernel function used to estimate. This function needs to return a gram matrix.

param_grid_precomputed
[dictionary] Dictionary with names (string) of parameters used to calculate gram matrices as keys and lists of parameter settings to try as values. This enables searching over any sequence of parameter settings. Params with length 1 will be omitted.

param_grid
[dictionary] Dictionary with names (string) of parameters used as penelties as keys and lists of parameter settings to try as values. This enables searching over any sequence of parameter settings. Params with length 1 will be omitted.

model_type
[string] Type of the problem, can be 'regression' or 'classification'.

NUM_TRIALS
[integer] Number of random trials of the outer CV loop. The default is 30.

datafile_y
[string] Path of file storing y data. This parameter is optional depending on the given dataset file.

extra_params
[dict] Extra parameters for loading dataset. See function gklearn.utils. graphfiles.loadDataset for detail.

ds_name
[string] Name of the dataset.

n_jobs
[int] Number of jobs for parallelization.

read_gm_from_file
[boolean] Whether gram matrices are loaded from a file.

Examples

```
>>> import numpy as np
>>> from gklearn.utils.model_selection_precomputed import model_selection_for_
↪precomputed_kernel
>>> from gklearn.kernels.untilHPathKernel import untilhpathkernel
>>>
>>> datafile = '../datasets/MUTAG/MUTAG_A.txt'
>>> estimator = untilhpathkernel
>>> param_grid_precomputed = {'depth':  np.linspace(1, 10, 10), 'k_func':
                ['MinMax', 'tanimoto'], 'compute_method':  ['trie']}
>>> # 'C' for classification problems and 'alpha' for regression problems.
>>> param_grid = [{'C': np.logspace(-10, 10, num=41, base=10)}, {'alpha':
                np.logspace(-10, 10, num=41, base=10)}]
>>>
>>> model_selection_for_precomputed_kernel(datafile, estimator,
                param_grid_precomputed, param_grid[0], 'classification', ds_
↪name='MUTAG')
```

parallel_trial_do(*param_list_pre_revised*, *param_list*, *y*, *model_type*, *trial*)

printResultsInTable(*param_list*, *param_list_pre_revised*, *average_val_scores*, *std_val_scores*, *average_perf_scores*, *std_perf_scores*, *average_train_scores*, *std_train_scores*, *gram_matrix_time*, *model_type*, *verbose*)

read_gram_matrices_from_file(*output_dir*, *ds_name*)

trial_do(*param_list_pre_revised*, *param_list*, *gram_matrices*, *y*, *model_type*, *trial*)

gklearn.utils.parallel

Created on Tue Dec 11 11:39:46 2018 Parallel aid functions. @author: ljia

parallel_ged_mat(*func*, *ged_mat*, *Gn*, *init_worker=None*, *glbv=None*, *method='imap_unordered'*, *n_jobs=None*, *chunksize=None*, *verbose=True*)

Parallel computing graph edit distance matrix.

Notes

This is equivalent to the function *parallel_gm*.

parallel_gm(*func*, *Kmatrix*, *Gn*, *init_worker=None*, *glbv=None*, *method='imap_unordered'*, *n_jobs=None*, *chunksize=None*, *verbose=True*)

parallel_me(*func*, *func_assign*, *var_to_assign*, *itr*, *len_itr=None*, *init_worker=None*, *glbv=None*, *method=None*, *n_jobs=None*, *chunksize=None*, *itr_desc=''*, *verbose=True*)

gklearn.utils.trie

Created on Wed Jan 30 10:48:49 2019

Trie (prefix tree) @author: ljia @references: NLP: Build a Trie Data structure from scratch with python, 2019.1

class Trie

Bases: `object`

deleteWord(*word*)

getNode()

insertWord(*word*)

load_from_json(*file_name*)

load_from_pickle(*file_name*)

save_to_json(*file_name*)

save_to_pickle(*file_name*)

searchWord(*word*)

searchWordPrefix(*word*)

to_json()

gklearn.utils.utils

class SpecialLabel(*value*)

Bases: Enum

can be used to define special labels.

DUMMY = 1

check_json_serializable(*obj*, *deep: bool = False*) → bool

Check if an object is JSON serializable.

Parameters

obj
[object] The object to be checked.

deep
[bool, optional] Whether to check the object recursively when *obj* is iterable. The default is False.

Returns

bool
True if the object is JSON serializable, False otherwise.

compute_distance_matrix(*gram_matrix*)

compute_gram_matrices_by_class(*ds_name*, *kernel_options*, *save_results=True*, *dir_save=''*, *irrelevant_labels=None*, *edge_required=False*)

compute_vertex_kernels(*g1*, *g2*, *node_kernels*, *node_labels=[]*, *node_attrs=[]*)

Compute kernels between each pair of vertices in two graphs.

Parameters

g1, g2
[NetworkX graph] The kernels bewteen pairs of vertices in these two graphs are computed.

node_kernels
[dict] A dictionary of kernel functions for nodes, including 3 items: 'symb' for symbolic node labels, 'nsymb' for non-symbolic node labels, 'mix' for both labels. The first 2 functions take two node labels as parameters, and the 'mix' function takes 4 parameters, a symbolic and a non-symbolic label for each the two nodes. Each label is in form of 2-D dimension array (n_samples, n_features). Each function returns a number as the kernel value. Ignored when nodes are unlabeled. This argument is designated to conjugate gradient method and fixed-point iterations.

node_labels
[list, optional] The list of the name strings of the node labels. The default is [].

node_attrs
[list, optional] The list of the name strings of the node attributes. The default is [].

Returns

vk_dict
[dict] Vertex kernels keyed by vertices.

Notes

This function is used by ``gklearn.kernels.FixedPoint'' and ``gklearn.kernels.StructuralSP''. The method is borrowed from FCSP [1].

References

Parallelization of shortest path graph kernels on multi-core cpus and gpus. Proceedings of the Programmability Issues for Heterogeneous Multicores (MultiProg), Vienna, Austria, 2014.

`direct_product`(*G1*, *G2*, *node_label*, *edge_label*)
Return the direct/tensor product of directed graphs G1 and G2.

Parameters

G1, G2
[NetworkX graph] The original graphs.

node_label
[string] node attribute used as label. The default node label is 'atom'.

edge_label
[string] edge attribute used as label. The default edge label is 'bond_type'.

Return

gt
[NetworkX graph] The direct product graph of G1 and G2.

Notes

This method differs from networkx.tensor_product in that this method only adds nodes and edges in G1 and G2 that have the same labels to the direct product graph.

References

`direct_product_graph`(*G1*, *G2*, *node_labels*, *edge_labels*)
Return the direct/tensor product of directed graphs G1 and G2.

Parameters

G1, G2
[NetworkX graph] The original graphs.

node_labels
[list] A list of node attributes used as labels.

edge_labels
[list] A list of edge attributes used as labels.

Return

gt
[NetworkX graph] The direct product graph of G1 and G2.

Notes

This method differs from networkx.tensor_product in that this method only adds nodes and edges in G1 and G2 that have the same labels to the direct product graph.

References

dummy_edge()

/*!

- @brief Returns a dummy edge.
- @return ID of dummy edge.

*/

dummy_node()

/*!

- @brief Returns a dummy node.
- @return ID of dummy node.

*/

find_all_paths(*G*, *length*, *is_directed*)
Find all paths with a certain length in a graph. A recursive depth first search is applied.

Parameters

G
[NetworkX graphs] The graph in which paths are searched.

length
[integer] The length of paths.

Return

path
[list of list] List of paths retrieved, where each path is represented by a list of nodes.

`find_paths`(*G*, *source_node*, *length*)
Find all paths with a certain length those start from a source node. A recursive depth first search is applied.

Parameters

G
[NetworkX graphs] The graph in which paths are searched.

source_node
[integer] The number of the node from where all paths start.

length
[integer] The length of paths.

Return

path
[list of list] List of paths retrieved, where each path is represented by a list of nodes.

`floydTransformation`(*G*, *edge_weight=None*)
Transform graph G to its corresponding shortest-paths graph using Floyd-transformation.

Parameters

G
[NetworkX graph] The graph to be tramsformed.

edge_weight
[string] edge attribute corresponding to the edge weight. The default edge weight is bond_type.

Return

S
[NetworkX graph] The shortest-paths graph corresponding to G.

References

getSPGraph(*G*, *edge_weight=None*)

Transform graph G to its corresponding shortest-paths graph.

Parameters

G
[NetworkX graph] The graph to be transformed.

edge_weight
[string] edge attribute corresponding to the edge weight.

Return

S
[NetworkX graph] The shortest-paths graph corresponding to G.

Notes

For an input graph G, its corresponding shortest-paths graph S contains the same set of nodes as G, while there exists an edge between all nodes in S which are connected by a walk in G. Every edge in S between two nodes is labeled by the shortest distance between these two nodes.

References

getSPLengths(*G1*)

get_edge_labels(*Gn*, *edge_label*)

Get edge labels of dataset Gn.

get_graph_kernel_by_name(*name*, *node_labels=None*, *edge_labels=None*, *node_attrs=None*, *edge_attrs=None*, *ds_infos=None*, *kernel_options={}*, ***kwargs*)

get_mlti_dim_edge_attrs(*G*, *attr_names*)

get_mlti_dim_node_attrs(*G*, *attr_names*)

get_node_labels(*Gn*, *node_label*)

Get node labels of dataset Gn.

get_shortest_paths(*G*, *weight*, *directed*)

Get all shortest paths of a graph.

Parameters

G
[NetworkX graphs] The graphs whose paths are calculated.

weight
[string/None] edge attribute used as weight to calculate the shortest path.

directed: boolean
Whether graph is directed.

Return

sp
[list of list] List of shortest paths of the graph, where each path is represented by a list of nodes.

`get_sp_graph`(*G*, *edge_weight=None*)

Transform graph G to its corresponding shortest-paths graph.

Parameters

G
[NetworkX graph] The graph to be transformed.

edge_weight
[string] edge attribute corresponding to the edge weight.

Return

S
[NetworkX graph] The shortest-paths graph corresponding to G.

Notes

For an input graph G, its corresponding shortest-paths graph S contains the same set of nodes as G, while there exists an edge between all nodes in S which are connected by a walk in G. Every edge in S between two nodes is labeled by the shortest distance between these two nodes.

References

`graph_deepcopy`(*G*)

Deep copy a graph, including deep copy of all nodes, edges and attributes of the graph, nodes and edges.

Note

- It is the same as the NetworkX function graph.copy(), as far as I know.
- This function only supports Networkx.Graph and Networkx.DiGraph.

graph_isIdentical(*G1*, *G2*)

Check if two graphs are identical, including: same nodes, edges, node labels/attributes, edge labels/attributes.

Notes

1. The type of graphs has to be the same.
2. Global/Graph attributes are neglected as they may contain names for graphs.

is_basic_python_type(*obj*, *type_list: list | None = None*, *deep: bool = False*) → bool

Check if an object is a basic type in Python.

Parameters

obj
[object] The object to be checked.

type_list
[list, optional] The list of basic types in Python. The default is None, which means the default basic types are used. The default basic types include *int*, *float*, *complex*, *str*, *bool*, *NoneType*, *list*, *tuple*, *dict*, *set*, *frozenset*, *range*, *slice*.

deep
[bool, optional] Whether to check the object recursively when *obj* is iterable. The default is False.

Returns

bool
True if the object is a basic type in Python, False otherwise.

normalize_gram_matrix(*gram_matrix*)

nx_permute_nodes(*G*, *random_state=None*)

Permute node indices in a NetworkX graph.

Parameters

G
[TYPE] DESCRIPTION.

random_state
[TYPE, optional] DESCRIPTION. The default is None.

Returns

G_new
[TYPE] DESCRIPTION.

Notes

- This function only supports Networkx.Graph and Networkx.DiGraph.

`undefined_node`()

/*!

- @brief Returns an undefined node.
- @return ID of undefined node.

*/

`untotterTransformation`(*G*, *node_label*, *edge_label*)

Transform graph G according to Mahé et al.'s work to filter out tottering patterns of marginalized kernel and tree pattern kernel.

Parameters

G
[NetworkX graph] The graph to be tramsformed.

node_label
[string] node attribute used as label. The default node label is 'atom'.

edge_label
[string] edge attribute used as label. The default edge label is 'bond_type'.

Return

gt
[NetworkX graph] The transformed graph corresponding to G.

References

11.2 Experiments

To exhibit the effectiveness and practicability of *graphkit-learn* library, we tested it on several benchmark datasets. See (Kersting et al., 2016) for details on these datasets.

A two-layer nested cross-validation (CV) is applied to select and evaluate models, where outer CV randomly splits the dataset into 10 folds with 9 as validation set, and inner CV then randomly splits validation set to 10 folds with 9 as training set. The whole procedure is performed 30 times, and the average performance is computed over these trails. Possible parameters of a graph kernel are also tuned during this procedure.

The machine used to execute the experiments is a cluster with 28 CPU cores of Intel(R) Xeon(R) E5-2680 v4 @ 2.40GHz, 252GB memory, and 64-bit operating system CentOS Linux release 7.3.1611. All results were run with Python 3.5.2.

The figure below exhibits accuracies achieved by graph kernels implemented in *graphkit-learn* library, in terms of regression error (the upper table) and classification rate (the lower table). Red color indicates the worse results and dark green the best ones. Gray cells with the "inf" marker indicate that the computation of the graph kernel on the dataset is omitted due to much higher consumption of computational resources than other kernels.

The figure below displays computational time consumed to compute Gram matrices of each graph kernels (in $log10$ of seconds) on each dataset. Color legends have the same meaning as in the figure above.

CHAPTER

TWELVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

g

INDEX

H

I

K

L

M

N

P

R

S

T

U

W