Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

pygrank: A Python package for graph node ranking

Emmanouil Krasanakis [a,b,*], Symeon Papadopoulos [a], Ioannis Kompatsiaris [a], Andreas L. Symeonidis [b]

[a] *Centre for Research and Technology Hellas, 57001 Thermi, Thessaloniki, Greece*
[b] *Aristotle University of Thessaloniki, 54124, Thessaloniki, Greece*

A R T I C L E I N F O

Article history:
Received 7 February 2022
Received in revised form 20 August 2022
Accepted 6 October 2022

Keywords:
Graph signal processing
Node ranking
Optimization

A B S T R A C T

We introduce *pygrank*, an open source Python package to define, run and evaluate node ranking algorithms. We provide object-oriented and extensively unit-tested algorithmic components, such as graph filters, post-processors, measures, benchmarks, and online tuning. Computations can be delegated to *numpy*, *tensorflow*, or *pytorch* backends and fit in back-propagation pipelines. Classes can be combined to define interoperable complex algorithms. Within the context of this paper, we compare the package with related alternatives, describe its architecture, demonstrate its flexibility and ease of use with code examples, and discuss its impact.

Code metadata

Current code version	*0.2.10*
Permanent link to code/repository used for this code version	https://github.com/MKLab-ITI/pygrank/releases/tag/0.2.10
Permanent link to reproducible capsule	
Legal code license	*Apache License, 2.0*
Code versioning system used	*git*
Software code language	*python*
Compilation requirements, operating environments and dependencies	*networkx, numpy, scipy, sklearn, wget, tensorflow, torch*
If available, link to developer documentation/manual	github.com/MKLab-ITI/pygrank
Support email for questions	maniospas@iti.gr

1. Motivation and significance

Representing data as graphs is popular in many domains; this involves designating data instances as nodes and linking the related instances through edges. Various graph analysis approaches have been developed, with node ranking algorithms being a powerful option when scoring nodes based on link structure and prior scores. Node ranking techniques include spectral graph filters [1], which can see use in community detection [2,3], link prediction [4,5], and graph neural networks [6,7]. However, most approaches employ algorithms and parameters with little to no ablation studies, when different choices could better match the structure of analyzed or – in case of deployed tools – new graphs.

To enable thorough investigation of node ranking algorithms and variations, we developed the *pygrank* Python package that implements a wide range of algorithmic components for machine learning and data mining pipelines. These include popular graph filters, objective enhancements (e.g., fairness-aware variations), supervised or unsupervised measures to evaluate algorithms on benchmark tasks, and online tuning of parameters.

Using the package is as simple as installing it by running the command line instruction *pip install --upgrade pygrank*, for example in a virtual environment, and importing it within source code. The imported package provides interfaces to define, run, and compare node ranking algorithms that comprise multiple components. By default, computations are delegated to efficient CPU-only processing with *numpy* for vector [8] and *scipy* [9] for sparse matrix operations. Developers can switch to different code execution backends that make use of in-built GPUs with either code instructions or by editing a local settings file.

* Corresponding author at: Centre for Research and Technology Hellas, 57001 Thermi, Thessaloniki, Greece.
E-mail addresses: maniospas@iti.gr (Emmanouil Krasanakis), papadop@iti.gr (Symeon Papadopoulos), ikom@iti.gr (Ioannis Kompatsiaris), symeonid@ece.auth.gr (Andreas L. Symeonidis).

https://doi.org/10.1016/j.softx.2022.101227

Table 1
Comparison of *pygrank* with alternatives for node ranking algorithms.

package	ad-hoc	backends	general	postpr.	tuning	backpr.	eval.
networkx	✓	numpy					✓
igraph	✓	custom C++					✓
pygsp	✓	numpy	✓				
dgl	✓	mxnet, pytorch, tensorflow				✓	
pyg	✓	pytorch				✓	
tfgnn	✓	tensorflow				✓	
spektral	✓	tensorflow				✓	
pygrank	✓	numpy, pytorch, tensorflow	✓	✓	✓	✓	✓

pygrank brings together a large number of node ranking literature practices (see Section 5). These can compare new and existing graph mining algorithms, construct algorithms from multiple components, and enable applications in different fields, such as social media or biological network analysis. The package has supported more than 7 publications on node ranking algorithms by the authors, whose results have been integrated in the code base and referenced in the documentation.

2. Scientific and technological context

Typically, node ranking algorithms build on the notion of graph signals, that is, maps between graph nodes and corresponding scores (real numbers). Signal priors hold "personalized" node information, such as probabilities that nodes are members of metadata communities [10]. Information can be propagated through edges with graph shift operations and graph filters aggregate multi-hop shifts based on either ad-hoc assumptions in the node space of how propagation behaves (e.g., as a stochastic random walk with restart [11]) or the desired impact on the adjacency matrix's eigenvalue spectrum [1]. Non-personalized variations are obtained when all nodes hold the same prior scores.

Most node ranking algorithms consist of base filters and postprocessing to augment their outcomes. Researchers and data scientists select which algorithms to use based on either ad-hoc criteria or on evaluation of competing alternatives on test graphs with supervised or unsupervised measures. In practice, popular ad-hoc algorithms are implemented in graph management packages, such as *networkx* [12] and *igraph* [13], or deep graph learning packages, such as *DGL* [14], *pyg* [15], *tfgnn* [16], and *spektral* [17]. Other implementations are parts of scientific tools, such as *textrank* [18]. Finally, *pygsp* [19] provides many types of graph filters, but does not support non-spectral analysis and does not fit into machine learning pipelines.

Table 1 compares *pygrank* to other packages that could run node ranking algorithms in terms of (a) provision of *ad-hoc* graph filters, (b) supported *backends*, (c) ability to define *general*-purpose filters with no additional coding, (d) *postpr*ocessing to improve outcomes, (e) online *tuning*, (f) *backpr*opagation support, and (g) *eval*uation measures of algorithm quality to enable both supervised and unsupervised evaluation depending on business needs. We consider only base capabilities that pertain to node ranking and are usable by non-experts without additional development. For example, external autoML packages, such as *autogluon* [20], require more coding to use in deep graph learning setups.

Overall, *pygrank* introduces new functionality and combines advantages of other packages; the latter span broader scopes but do not adequately explore node ranking. Hence, when node ranking is needed, our implementations can be plugged in graph mining and machine learning applications to easily assemble algorithms from novel or existing graph filters and postprocessors. The package further simplifies writing benchmark experiments for the selection of best algorithms and can automatically tune parameters to fit on graphs encountered at runtime.

3. Software description

3.1. Software architecture

Fig. 1 illustrates the package's functional components, modules and dependencies. For ease of use, all components are accessible from the top level import. The *core* manages programming interface abstractions of backends, defines graph signals, and provides additional utilities, such as graph preprocessors to be used by measures or algorithms. *Measures* defines supervised and unsupervised measures that either compare posterior graph signals to ground truth ones or assess quantitative characteristics in relation to graph dynamics. *Algorithms* comprises graph filters, convergence, early stopping schemes, postprocessors, and online tuning components, which can be combined to define a variety of interoperable node ranking algorithms. *Benchmarks* provides helper methods to quickly design node ranking experiments for collections of graphs, algorithms and training-test splits under evaluation measures. *Fastgraph* provides a graph data structure that re-implements *networkx*'s programming interfaces while avoiding data constructs not used by our package (e.g. node neighbor indexes), therefore fitting larger graphs in memory and constructing them with fewer computations.

All components are curated so that, for connected undirected graphs and often for directed ones, combinations run in log-linear amortized times $O(E \log E)$ with respect to the number of edges E. In particular, the shift operator of graph filters repeats sparse matrix–vector multiplications that only traverse the non-zero elements of graph adjacency matrices. At worst, to reach small numerical tolerances, repetitions grow logarithmically with the number of edges for connected undirected graphs and are generally upper-bounded by most filters. Graph signal element-by-element operations (e.g. addition, scalar multiplication) conclude in time proportional to the number of nodes, which is similar or less than the number of edges. Preprocessing only operates on the diagonal and non-zero elements of graph adjacency matrices, which correspond to the number of nodes and edges respectively. Measures involve only element-by-element graph signal operations, sampling fixed numbers of non-neighbors for each node, and sorting node scores; these take up linear or log-linear time with respect to the number of nodes - and hence edges. Finally, postprocessors and automatic tuning employ simple graph signal operations and – sometimes – bounded numbers of base algorithm and evaluation repetitions. Thus, running times of component combinations are at worst log-linear with respect to the number of edges. Large multiplicative terms could apply, but these are scale-free and the package remains applicable to graphs that fit in memory (e.g. with tens of millions of edges).

In addition to the *pygrank* package, which can be found in the namesake sub-directory of our code base, we also implement software engineering practices that ensure high code quality, maintainability, and extensibility. These include unit testing of components to assert that algorithms perform as advertised, at least on sample graphs, continuous integration that ensures

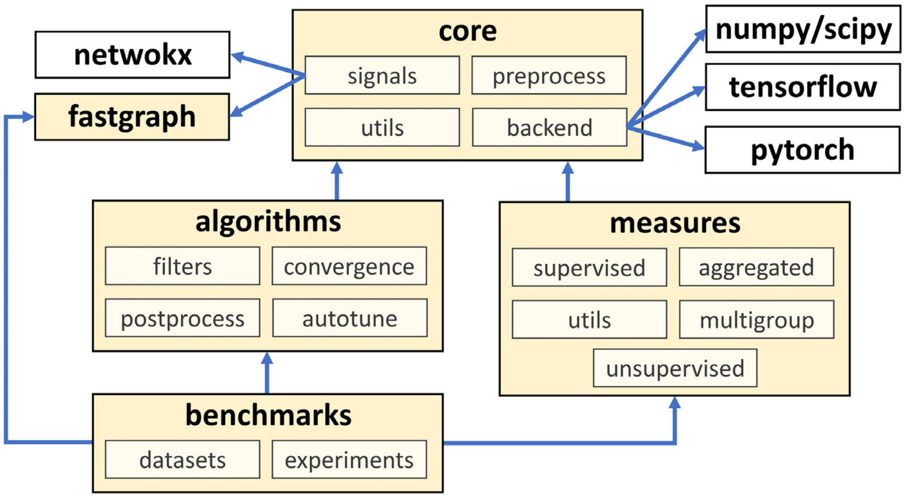

Fig. 1. Functional components of *pygrank*, their modules and dependencies.

that tests pass and achieve near-100% code coverage, *git* hook scripts to automatically document newly-integrated components, extensive documentation, and comprehensive code examples.

3.2. Software functionality

In *pygrank*, graph signals passed to code components are compatible with multiple practical data structures, such as dictionaries between nodes and scores, or lists of nodes to assign scores of 1 to. In both cases, omitted nodes obtain scores of 0. The package defines its own *GraphSignal* class to hold node ranking outputs. Its instances can also be manually generated to be used as inputs and behave as Python dictionaries between nodes and scores. Internally, signals hold (retrievable) backend primitives (e.g. *numpy* arrays, *tensorflow* tensors, *pytorch* tensors) to keep track of scores and are seamlessly exchanged between code components. This speeds up computations and makes most algorithms backpropagate-able on GPU backends.

Graph filters are initialized based on their types and parameters, and independently from data on which they run. Parameters include symmetric vs. column-based graph adjacency matrix normalization, employing the renormalization trick [21], caching repeated computations, and following the Lanczos method [22] for fast approximation of results. Filters or more complex algorithms are used as callables that take as inputs combinations of *networkx* graphs and graph signals. *GraphSignal* instances are tied to specific graphs and, when passed to algorithms, the graph argument can be omitted. Postprocessors are initialized with base algorithms to be wrapped (e.g., filter instances) and additional parameters. Multiple postprocessors can wrap each other to create complex algorithms, such as organization of nodes into overlapping or partitioned communities and rank-based link prediction. Corresponding code examples are provided in the package's documentation.

A variety of evaluation measures are provided to assess filter outcomes. There are two types of measures: (a) supervised ones that are instantiated with ground truth graph signals, and (b) unsupervised measures that are instantiated with a *fastgraph* or *networkx* graph they are applied on. Providing the graph to unsupervised measures is optional, as long as *GraphSignal* instances (e.g., the outcomes of algorithms) are evaluated later on to retrieve the graph from those. The package also implements measures that address multiple objectives, such as trading-off between measures or assessing the outcome of more than one community detection tasks.

The package also offers online tuning interfaces used as normal node ranking algorithm callables while wrapping either sets of competing algorithms or parameterized algorithm definitions. These respectively select node ranking algorithms and parameters at runtime by optimizing any of provided measures. In case of supervised measures, tuning is guided by a train-validation split of graph signal inputs. Whether measures are optimized for high or low values is automatically determined.

Finally, the package supports large-scale experimentation. To this end, it includes helper methods that load graph datasets following the SNAP format of defining edges through node pairs and node communities as lists [23], download and parse certain datasets from online sources, perform training-test splits on community data, and even automate graph neural network training for node classification. Loaded graphs are by default generated with the *fastgraph* module, but they can also be constructed as *networkx* graphs. Methods can apply postprocessors and create more variations of large sets of node ranking alternatives. A benchmarking interface can compare multiple algorithms across many datasets with many training-test split ratios and under designated measures.

4. Illustrative examples

Algorithm definition. The following code defines a HeatKernel filter [24] with custom parameters and creates a seed oversampling variation [25] to improve metadata community recommendations. Then, a helper method creates two more variations wrapping the two algorithms with the sweep ratio [26] while adding a *"+sweep"* suffix to their names. Finally, all algorithms are wrapped with a normalization step and run on an example graph dataset to score how much nodes relate to a community of interest. Score-based conductance [27] assesses all outcomes (lower is better) and we show the console output, which verifies that there can be merit to combining multiple components.

```
import pygrank as pg

algorithm = pg.HeatKernel(t=5,  # the number of hops
    to place maximal importance on
                        normalization="symmetric",
                            renormalize=True)
algorithms = {"hk5": algorithm, "hk5+oversampling": pg
    .SeedOversampling(algorithm)}
algorithms = algorithms | pg.create_variations(
    algorithms, {"+sweep": pg.Sweep})
algorithms = pg.create_variations(algorithms, pg.
    Normalize)
```

```
_, graph, community = next(pg.
    load_datasets_one_community(["EUCore"]))
personalization = {node: 1. for node in community}  #
    missing scores considered zero
measure = pg.Conductance()  # smaller means tightly-
    knit stochastic community
for algorithm_name, algorithm in algorithms.items():
    scores = algorithm(graph, personalization)  #
        returns a dict-like pg.GraphSignal
    pg.benchmark_print_line(algorithm_name, measure(
        scores), tabs=[20, 5])  # pretty
```

```
hk5                      9.53
hk5+oversampling         9.11
hk5+sweep                9.18
hk5+oversampling+sweep   8.84
```

Benchmarks. The following code demonstrates benchmark experiments on two datasets with community node labels to recommend new members by using half of the nodes for training and the rest for testing. Assuming that graphs are immutable, experiments are sped up by caching the outcomes of a shared graph preprocessor. Console outputs by running the code (could be formatted as latex too) on a random split assert the usefulness of online tuning. This tends to exhibit similar or better AUC than individual algorithms, where best algorithms could be different for different graphs.

```
import pygrank as pg

datasets = ["CiteSeer", "EUCore"]
pre = pg.preprocessor(assume_immutability=True,
    normalization="symmetric")  # shared
algs = {"ppr.85": pg.PageRank(.85, preprocessor=pre),
        "ppr.99": pg.PageRank(.99, preprocessor=pre,
            max_iters=1000),
        "hk3": pg.HeatKernel(3, preprocessor=pre),
        "hk5": pg.HeatKernel(5, preprocessor=pre),
        "tuned": pg.ParameterTuner(preprocessor=pre)}
loader = pg.load_datasets_one_community(datasets)
pg.benchmark_print(pg.benchmark(algs, loader, pg.AUC,
    fraction_of_training=.5))
```

```
                 ppr.85    ppr.99     hk3    hk5    tuned
CiteSeer         .87       .87        .87    .87    .87
EUCore           .83       .48        .89    .87    .91
```

Defining graph neural networks. pygrank can help set up graph neural network architectures employing node ranking algorithms. For example, the following code implements the predict-then-propagate architecture [6] for node classification; a multilayer perceptron is defined with the *keras* package [28], *pygrank* propagates perceptron prediction columns through the graph with a namesake method, and outcomes pass through a *softmax* activation. We start from a graph with node features, create a seeded 60-20-20 training-validation-test split of nodes and train the model with a helper method *gnn_train*. The *tensorflow* backend (similar implementations can be achieved for *pytorch*) lets backpropagation pass through node ranking. Training would be the same if test labels were masked to zeros. Notably, graph filter propagation provided by the package takes up only two lines of code to define and run.

```
import pygrank as pg
import tensorflow as tf
from tensorflow.keras.layers import Dropout, Dense
from tensorflow.keras.regularizers import L2

class APPNP(tf.keras.Sequential):
    def __init__(self, num_inputs, num_outputs, hidden
        =64):
        super().__init__([
            Dropout(0.5, input_shape=(num_inputs,)),
            Dense(hidden, activation="relu",
                kernel_regularizer=L2(0.005)),
            Dropout(0.5),
            Dense(num_outputs)])
        self.ranker = pg.PageRank(0.9, renormalize=
            True, assume_immutability=True,
            use_quotient=False, error_type="iters",
                max_iters=10)  # 10 iterations

    def call(self, features, graph, training=False):
        # can call with tensor graph
        predict = super().call(features, training=
            training)
        propagate = self.ranker.propagate(graph,
            predict, graph_dropout=0.5*training)
        return tf.nn.softmax(propagate, axis=1)

graph, features, labels = pg.load_feature_dataset("
    citeseer")
training, test = pg.split(list(range(len(graph))),
    0.8, seed=5)  # seeded split
training, validation = pg.split(training, 1 - 0.2 /
    0.8)
model = APPNP(features.shape[1], labels.shape[1])
with pg.Backend("tensorflow"):  # pygrank with
    tensorflow backend
    pg.gnn_train(model, features, graph, labels,
        training, validation,
              optimizer=tf.optimizers.Adam(
                  learning_rate=0.01), verbose=
                  True)
    print("Accuracy", pg.gnn_accuracy(labels, model(
        features, graph), test))
```

Online tuning also fits into machine learning pipelines. For example, the following code lets APPNP's ranker automatically determine the diffusion parameter ($par[0] \in [0.5, 1]$) per propagated dimension of a "manual" PageRank definition. Tuning each propagation improves the test accuracy score averaged across code runs for $seed = 0, 1, \ldots, 9$ from 76.2% to 77.5%. Parameter selection is made to run on the *numpy* backend; as of writing, this is several times faster than GPU backends (*tensorflow*, *torch*) for graph signal shifts. The package spends most of its time on this operation, due to frequent repetition and a running time proportional to the number of graph edges. Other backend operations do not create bottlenecks, as they are applied either infrequently or element-by-element on node scores. After selecting parameters, computations return to the *tensorflow* backend, which is backpropagate-able and could run further computations in the GPU.

```
pre = pg.preprocessor(renormalize=True,
    assume_immutability=True)
self.ranker = pg.ParameterTuner(
            lambda par: pg.GenericGraphFilter([par[0]
                ** i for i in range(int(10))],
               preprocessor=pre, error_type="iters",
                   max_iters=10),
            max_vals=[1], min_vals=[0.5], verbose=
                False,
            measure=pg.Mabs, deviation_tol=0.01,
                tuning_backend="numpy")
```

5. Impact

pygrank brings together a variety of node ranking algorithm tools. First, it has supported and integrates approaches validated on many algorithms and variations. These range from works on postprocessors (e.g. [25]) to early stopping criteria, and unsupervised evaluation of ranking quality [27]. Second, it implements third-party research, including but not limited to PageRank [29], HeatKernel [24] and absorbing random walk [30] graph filters, custom adjacency matrix normalizations, including potential renormalization [21], sweep postprocessors for community detection [26], caching and approximate Arnoldi space [31] speed-ups, and tuning in the non-convex space of graph filter parameters with a mixture of divided rectangles [32] and coordinate descent optimization [33]. Usage of these features is simplified thanks to interoperable interfaces.

We recognize four prospective applications of the package, each achieved with only a few lines of code: (i) defining and running node ranking algorithms based on ad-hoc criteria, (ii) defining multiple promising algorithms and selecting the best with experiments on domain-related graph datasets, (iii) adjusting algorithm parameters at runtime with online tuning, and (iv) integrating algorithms in machine learning pipelines, such as graph neural networks. These applications can help researchers answer questions pertaining to which algorithms to use for different graph domains, for instance by comparing multiple promising algorithms on benchmark graph mining tasks and selecting the best.

Moreover, *pygrank* enables principled exploration of new node ranking algorithmic components or evaluation practices by comparing them to a variety of existing alternatives under scientifically rigorous settings. For multi-component algorithms, the same process also simplifies ablation studies. Overall, the package reduces the development load of re-implementing code scattered throughout the literature and promotes faster and on-the-point research. These features have aided the authors in their work, and could therefore also prove useful to other researchers working on node ranking.

Finally, high-level programming interface abstractions define and run algorithms with few lines of code and encourage use of state-of-the-art practices, even by non-experts. In practice, algorithm implementations are both scalable and come alongside popular speed-ups that let them run fast, even on large graphs with millions of edges. These characteristics make the package ideal for integrating technologically-ready solutions with minimal engineering costs.

6. Conclusions

We have developed a Python package, called *pygrank*, that provides comprehensive and interoperable implementations of graph node ranking algorithmic components, as well as tuning, evaluation and benchmarking capabilities. The current version comprises 7 types of highly parameterized base graph filters, 17 types of postprocessors (each featuring multiple variations), three types of online parameter tuning strategies, and 30 algorithm evaluation measures. In addition to parsing programming inputs, it can automatically download and parse 15 web resources to conduct out-of-the-box experiments. Algorithms have been optimized in terms of computational complexity, allocated memory and speed-up techniques, and support backpropagation by GPU backends.

In the future, we plan to integrate even more existing and novel literature components in the code base. Researchers are also encouraged to contribute implementations of their work.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Emmanouil Krasanakis reports financial support was provided by European Commission. Emmanouil Krasanakis reports a relationship with Centre for Research and Technology-Hellas that includes: employment.

Data availability

Data are automatically downloaded when running the code

Acknowledgments

Development of the package was partially funded by the European Commission under contract numbers H2020-951911 AI4Media and H2020-825585 HELIOS.

References

[1] Ortega A, Frossard P, Kovačević J, Moura JM, Vandergheynst P. Graph signal processing: Overview, challenges, and applications. Proc IEEE 2018;106(5):808–28.
[2] Zhang Y, Xia X, Xu X, Yu F, Wu H, Yu Y, et al. Robust hierarchical overlapping community detection with personalized pagerank. IEEE Access 2020;8:102867–82.
[3] Gao Y, Yu X, Zhang H. Overlapping community detection by constrained personalized PageRank. Expert Syst Appl 2021;173:114682.
[4] Nassar H, Benson AR, Gleich DF. Pairwise link prediction. In: 2019 IEEE/ACM international conference on advances in social networks analysis and mining. IEEE; 2019, p. 386–93.
[5] Wu X, Wu J, Li Y, Zhang Q. Link prediction of time-evolving network based on node ranking. Knowl-Based Syst 2020;195:105740.
[6] Klicpera J, Bojchevski A, Günnemann S. Predict then propagate: Graph neural networks meet personalized pagerank. 2018, arXiv preprint arXiv:1810.05997.
[7] Huang Q, He H, Singh A, Lim S-N, Benson AR. Combining label propagation and simple models out-performs graph neural networks. 2020, arXiv preprint arXiv:2010.13993.
[8] Harris CR, Millman KJ, van der Walt SJ, Gommers R, Virtanen P, Cournapeau D, et al. Array programming with NumPy. Nature 2020;585(7825):357–62.
[9] Virtanen P, Gommers R, Oliphant TE, Haberland M, Reddy T, Cournapeau D, et al. SciPy 1.0: fundamental algorithms for scientific computing in Python. Nature Methods 2020;17(3):261–72.
[10] Peel L, Larremore DB, Clauset A. The ground truth about metadata and community detection in networks. Sci Adv 2017;3(5):e1602548.
[11] Tong H, Faloutsos C, Pan J-Y. Fast random walk with restart and its applications. In: Sixth international conference on data mining. IEEE; 2006, p. 613–22.
[12] Hagberg A, Swart P, Chult DS. Exploring network structure, dynamics, and function using NetworkX. Tech. rep., Los Alamos, NM (United States): Los Alamos National Lab.(LANL); 2008.
[13] Csardi G, Nepusz T, et al. The igraph software package for complex network research. InterJ Complex Syst 2006;1695(5):1–9.
[14] Wang M, Yu L, Zheng D, Gan Q, Gai Y, Ye Z, et al. Deep graph library: towards efficient and scalable deep learning on graphs. 2019.
[15] Fey M, Lenssen JE. Fast graph representation learning with PyTorch Geometric. 2019, arXiv preprint arXiv:1903.02428.
[16] Ferludin O, Eigenwillig A, Blais M, Zelle D, Pfeifer J, Sanchez-Gonzalez A, et al. TF-GNN: graph neural networks in TensorFlow. 2022, CoRR, vol. abs/2207.03522.
[17] Grattarola D, Alippi C. Graph neural networks in tensorflow and keras with spektral. 2020, arXiv preprint arXiv:2006.12138.
[18] Mihalcea R, Tarau P. Textrank: Bringing order into text. In: Proceedings of the 2004 conference on empirical methods in natural language processing. 2004, p. 404–11.
[19] Defferrard M, Martin L, Pena R, Perraudin N. PyGSP: graph signal processing in python.
[20] Erickson N, Mueller J, Shirkov A, Zhang H, Larroy P, Li M, et al. Autogluon-tabular: Robust and accurate automl for structured data. 2020, arXiv preprint arXiv:2003.06505.
[21] Kipf TN, Welling M. Semi-supervised classification with graph convolutional networks. 2016, arXiv preprint arXiv:1609.02907.
[22] Susnjara A, Perraudin N, Kressner D, Vandergheynst P. Accelerated filtering on graphs using lanczos method. 2015, arXiv preprint arXiv:1509.04537.
[23] Leskovec J, Krevl A. SNAP datasets: stanford large network dataset collection. 2014, http://snap.stanford.edu/data.
[24] Chung F. The heat kernel as the pagerank of a graph. Proc Natl Acad Sci 2007;104(50):19735–40.
[25] Krasanakis E, Schinas E, Papadopoulos S, Kompatsiaris Y, Symeonidis A. Boosted seed oversampling for local community ranking. Inf Process Manage 2020;57(2):102053.

[26] Andersen R, Chung F, Lang K. Local partitioning for directed graphs using pagerank. Internet Math 2008;5(1–2):3–22.

[27] Krasanakis E, Papadopoulos S, Kompatsiaris Y. Unsupervised evaluation of multiple node ranks by reconstructing local structures. Appl Netw Sci 2020;5(1):1–32.

[28] Gulli A, Pal S. Deep learning with keras. Packt Publishing Ltd; 2017.

[29] Page L, Brin S, Motwani R, Winograd T. The PageRank citation ranking: Bringing order to the web. Tech. rep., Stanford InfoLab; 1999.

[30] Wu X-M, Li Z, So AM-C, Wright J, Chang S-F. Learning with partially absorbing random walks. In: NIPS. vol. 25, 2012, p. 3077–85.

[31] Golub GH, Greif C. An arnoldi-type algorithm for computing page rank. BIT Numer Math 2006;46(4):759–71.

[32] Finkel DE, Kelley C. Additive scaling and the DIRECT algorithm. J Global Optim 2006;36(4):597–608.

[33] Lyu H. Convergence of block coordinate descent with diminishing radius for nonconvex optimization. 2020, arXiv preprint arXiv:2012.03503.