Karate Club: An API Oriented Open-Source Python Framework for Unsupervised Learning on Graphs

Citation for published version:

Digital Object Identifier (DOI):
10.1145/3340531.3412757

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
Proceedings of the 29th ACM International Conference on Information & Knowledge Management

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.
Karate Club: An API Oriented Open-Source Python Framework for Unsupervised Learning on Graphs

Benedek Rozemberczki
The University of Edinburgh
Edinburgh, United Kingdom
benedek.rozemberczki@ed.ac.uk

Oliver Kiss
Central European University
Budapest, Hungary
kiss_oliver@phd.ceu.edu

Rik Sarkar
The University of Edinburgh
Edinburgh, United Kingdom
rsarkar@inf.ed.ac.uk

ABSTRACT
Graphs encode important structural properties of complex systems. Machine learning on graphs has therefore emerged as an important technique in research and applications. We present Karate Club – a Python framework combining more than 30 state-of-the-art graph mining algorithms. These unsupervised techniques make it easy to identify and represent common graph features. The primary goal of the package is to make community detection, node and whole graph embedding available to a wide audience of machine learning researchers and practitioners. Karate Club is designed with an emphasis on a consistent application interface, scalability, ease of use, sensible out of the box model behaviour, standardized dataset ingestion, and output generation. This paper discusses the design principles behind the framework with practical examples. We show Karate Club’s efficiency in learning performance on a wide range of real world clustering problems and classification tasks along with supporting evidence of its competitive speed.

KEYWORDS
network embedding, graph embedding, representation learning, network analytics, graph mining

1 INTRODUCTION
Techniques that extract features from graph data in an unsupervised manner [19, 24, 49] have seen an increasing success in the machine learning community. Features automatically extracted by these methods can serve as inputs for link prediction, node and graph classification, community detection and various other tasks [19, 24, 25, 30, 44] in a wide range of real world research and application scenarios. Graph mining tools such as [13, 17, 23] have contributed to enhancement and development of machine learning pipelines. The need for complicated custom feature engineering is reduced by unsupervised graph mining techniques. This approach produces features that are naturally reusable on multiple types of tasks. Recent research [24, 25, 38] has made such feature extraction highly efficient and parallelizable.

The democratization of machine learning for tabular data was led by frameworks which made fast paced development possible. Tools such as [1, 5, 21, 22, 32] are available in general purpose scripting languages with easy to use consistent interfaces. On the other hand, current graph based learning frameworks are less mature and of limited scope [13, 23]. For example, these packages host certain community detection algorithms, but none for whole graph or node embedding. In addition, some of these tools [17, 23] have significant barriers for the end users in terms of installing prerequisites and custom data structures used for representing graphs.

Present work. We propose Karate Club, an open source Python framework for unsupervised learning on graphs. We implemented Karate Club with consistent API oriented design principles in mind which makes the library end user friendly and modular. The name of the package is inspired by Zachary’s Karate Club [52] – a network commonly used to demonstrate network algorithms. The design of this machine learning toolbox was motivated by the principles used to create the widely used scikit-learn package [5].

The design entails a number of fundamental engineering patterns. Each algorithm has a sensible default hyperparameter setting which helps non expert practitioners. To further ease the use of our package, algorithms have a limited number of shared, publicly available methods (e.g. fit). Models ingest data structures from the scientific Python ecosystem [13, 41, 42] as input and the generated output also follows these formats. The inner model mechanics are always implemented as private methods using optimized back-end libraries [9, 29, 41, 42] for computing. These principles combined with the extensive documentation ensure that Karate Club is accessible to a wider audience than the currently available open-source graph mining frameworks.

Our empirical evaluation focuses on three common graph mining tasks: non-overlapping community detection, node and graph classification. We compare the learning performance of node and graph level algorithms implemented in our framework on various real world social, web and collaboration networks (collected from Deezer, Reddit, Facebook, Twitch, Wikipedia and GitHub). With respect to the runtime, models in Karate Club show reasonable scalability which we demonstrate by the use of synthetic data.
Our contributions. Specifically our work makes the following key contributions:

1. We release Karate Club, a Python graph mining framework which provides a wide range of easy to use community detection, node and whole graph embedding procedures.
2. We demonstrate with code the main ideas of the API oriented framework design: hyperparameter encapsulation and inspection, available public methods, dataset ingestion, output generation, and interfacing with downstream learning algorithms and evaluation methods.
3. We evaluate the learning performance of the framework on real world community detection, node and graph classification problems. We validate the scalability of the algorithms implemented in our framework.
4. We open sourced with the framework a detailed documentation and released multiple large graph classification datasets.

The remainder of this paper is structured as follows. In Section 2 we discuss the covered graph mining techniques. We overview the main principles behind Karate Club in Section 3 where we included detailed examples to explain these design ideas. The learning performance and scalability of the algorithms in the package are evaluated in Section 4. We review related data mining libraries in Section 5. The paper concludes with Section 6 where we discuss future directions. The source code of Karate Club can be downloaded from https://github.com/benedekrozemberczki/karateclub; the Python package can be installed from the Python Package Index. Extensive documentation is available at https://karateclub.readthedocs.io/en/latest/ with detailed examples.

2 GRAPH MINING PROCEDURES IN KARATE CLUB

In this section, we briefly discuss the various graph mining algorithms which are available in the 1.0. release of the Karate Club package.

2.1 Community detection

Community detection techniques cluster the vertices of the graph into densely connected groups of nodes. This grouping can be the final result or an input for a downstream learning algorithm (e.g. node classification or link prediction).

Karate Club currently contains several methods for overlapping and non-overlapping community detection. Non-overlapping community detection is analogous to clustering, and assumes that a node can only belong to a single group; see, for example, [18, 26, 28, 31]. While overlapping community detection is analogous to fuzzy clustering as nodes have an affiliation with multiple clusters; look for example [16, 36, 43, 49, 51].

2.2 Node embedding

Node embeddings map vertices of a graph into an Euclidean space in which those that are deemed to be similar according to a certain notion will be in close proximity. The Euclidean representation makes it easier to apply standard machine learning libraries for node classification, link prediction, clustering etc.

Neighbourhood preserving embedding maintains the proximity of nodes in the graph when an embedding is created. These methods implicitly [24, 25, 33] or explicitly [6, 15, 27, 37] decompose a proximity matrix (e.g. powers of the adjacency matrix or a sum of these matrices) to create the node embedding.

Structural embedding conserves the structural roles of nodes in the embedding space [2, 10, 14]. Nodes with similar embeddings have a similar distribution of descriptive statistics (e.g. centrality and clustering coefficient) in their vicinity. Embeddings are distilled by the decomposition of matrices representing structural features of nodes.

Attributed embedding retains the neighbourhood structure and generic feature similarity of nodes when the embedding is learned. This learning involves the joint factorization of the node-node and node-feature matrices with a direct [45, 48] or implicit matrix decomposition technique [30, 53].

Meta embedding combines information from neighbourhood preserving, structural and attributed embeddings in order to create higher representation quality embeddings [46].

2.3 Whole graph embedding and summarization

Whole graph embedding and summarization techniques create fixed size representations of entire graphs as points in a Euclidean space. Those graphs which are close in the embedding space share structural patterns such as subtrees. These representations are used for a range of graph level tasks – graph classification, regression and whole graph clustering.

Spectral fingerprints extract statistics from the eigenvectors and eigenvalues of the graph Laplacian [8, 38, 40]. Vectors of the descriptive statistics are used as the whole graph representation.

Implicit factorization techniques create a graph – structural feature matrix [7, 19] by enumerating string features in the graphs. This matrix is decomposed in order to create whole graph descriptors and feature embeddings jointly.

3 DESIGN PRINCIPLES

When we created Karate Club, we used an API oriented machine learning system design point of view [5, 22] in order to make an end-user friendly machine learning tool. This API oriented design principle entails a few simple ideas. In this section we discuss these ideas and their apparent advantages with appropriate illustrative examples in great detail.

3.1 Encapsulated model hyperparameters and inspection

An unsupervised Karate Club model instance is created by using the constructor of the appropriate Python object. This constructor has a default hyperparameter setting which allows for sensible out-of-the-box model usage. In simple terms this means that the end user does not need to understand the inner model mechanics in great detail to use the methods implemented in our framework. We set these default hyperparameters to provide a reasonable learning and runtime performance. If needed, these model hyperparameters can be modified at the model instance creation time with the appropriate
re-parametrisation of the constructor. The hyperparameters are stored as public attributes to allow the inspection of model settings. We demonstrate the encapsulation of hyperparameters by the code snippet in Figure 1. First, we want to create an embedding for a NetworkX generated Erdos-Renyi graph (line 4) with the standard hyperparameter settings. When the model is constructed and fitted (lines 6-7) we do not change default hyperparameters and we can print the standard setting of the dimensions hyperparameter (line 8). Second, we decided to set a different number of dimensions, so we created and fitted a new model (lines 10-11) and we print the new value of the dimensions hyperparameter (line 12).

```
import networkx as nx
from karateclub import DeepWalk

graph = nx.gnm_random_graph(100, 1000)
model = DeepWalk()
model.fit(graph)
print(model.dimensions)

model = DeepWalk(dimensions=64)
model.fit(graph)
print(model.dimensions)
```

Figure 1: Creating a synthetic graph, using a DeepWalk model with standard and modified hyperparameter settings.

### 3.2 API Consistency and non-proliferation of classes

Each unsupervised machine learning model in Karate Club is implemented as a separate class which inherits from the Estimator class. Algorithms implemented in our framework have a limited number of public methods as we do not assume that the end user is particularly interested in the algorithmic details related to a specific technique. All models are trained by the use of the fit method which takes the inputs (graph, node features) and calls the appropriate private methods to learn an embedding or clustering. Node and graph embeddings are returned by the get_embedding public method and cluster memberships are retrieved by calling get_memberships.

```
import networkx as nx
from karateclub import DeepWalk

graph = nx.gnm_random_graph(100, 1000)
model = DeepWalk()
model.fit(graph)
embedding = model.get_embedding()
```

Figure 2: Creating a synthetic graph, using the DeepWalk constructor, fitting the embedding and returning it.

We avoided the proliferation of classes with two specific strategies. First, the inputs used by our framework and the outputs generated do not rely on custom data classes. This helps to prevent the unnecessary growth of the number of classes and also helps with interfacing with downstream applications. Second, algorithms which use the same data pre-processing or algorithmic step (e.g. truncated random walk, Weisfeiler-Lehman hashing) were built on shared blocks.

In Figure 2 we create a random graph (line 4), and a DeepWalk model with the default hyperparameters (line 6), we fit this model (line 7) using the public fit method (line 7) and return the embedding by calling the public get_embedding method (line 8).

```
import networkx as nx
from karateclub import Walklets

graph = nx.gnm_random_graph(100, 1000)
model = Walklets()
model.fit(graph)
embedding = model.get_embedding()
```

Figure 3: Creating a synthetic graph, using the Walklets constructor, fitting the embedding and returning it.

The example in Figure 2 can be modified to create a Walklets embedding with minimal effort by changing the model import (line 2) and the constructor (line 6) - these modifications result in the snippet of Figure 3.

Looking at these two snippets the advantage of the API driven design is evident as we only needed to do a few modifications. First, we had to change the import of the embedding model. Second, we needed to modify the model construction and the default hyperparameters were already set. Third, the public methods provided by the DeepWalk and Walklets classes behave the same way. An embedding is learned with fit and it is returned by get_embedding. This allows for quick and minimal changes to the code when an upstream unsupervised model used for feature extraction performs poorly.

### 3.3 Standardized dataset ingestion

We designed Karate Club to use standardized dataset ingestion when a model is fitted. Practically this means that algorithms which have the same purpose use the same data types for model training. In detail:

- Neighbourhood based and structural node embedding techniques use a single NetworkX graph as input for the fit method.
- Attributed node embedding procedures take a NetworkX graph as input and the features are represented as a NumPy array or as a SciPy sparse matrix. In these matrices rows correspond to nodes and columns to features.
- Graph level embedding methods and statistical graph fingerprints take a list of NetworkX graphs as an input.
- Community detection methods use a NetworkX graph as an input.
3.4 High performance model mechanics

The underlying mechanics of the graph mining algorithms were implemented using widely available Python libraries which are not operation system dependent and do not require the presence of other external libraries like TensorFlow or PyTorch does [1, 21]. The internal graph representations in Karate Club use NetworkX. Dense linear algebra operations are done with NumPy and their sparse counterparts use SciPy. Implicit matrix factorization techniques [2, 24, 25, 30, 53] utilize the GenSim [29] package and methods which rely on graph signal processing use PyGSP [9].

```python
import community
import networkx as nx
from karateclub import LabelPropagation, SCD

graph = nx.gnm_random_graph(100, 1000)
model = SCD()
model.fit(graph)
lp_memberships = model.get_memberships()
print(community.modularity(lp_memberships, graph))

model = LabelPropagation()
model.fit(graph)
scd_memberships = model.get_memberships()
print(community.modularity(scd_memberships, graph))
```

Figure 4: Creating a synthetic graph, clustering with two community detection techniques and using an external library to evaluate the modularity of clusterings.

3.5 Standardized output generation and downstream interfacing

The standardized output generation of Karate Club ensures that unsupervised learning algorithms which serve the same purpose always return the same type of output with a consistent data point ordering. There is a very important consequence of this design principle. When a certain type of algorithm is replaced with the same type of algorithm, the downstream code which uses the output of the upstream unsupervised model does not have to be changed. Specifically the outputs generated with our framework use the following data structures:

- **Node embedding algorithms** (neighbourhood preserving, attributed and structural) always return a NumPy float array when the `get_embedding` method is called. The number of rows in the array is the number of vertices and the row index always corresponds to the vertex index. Furthermore, the number of columns is the number of embedding dimensions.
- **Whole graph embedding methods** (spectral fingerprints, implicit matrix factorization techniques) return a NumPy float array when the `get_embedding` method is called. The row index corresponds to the position of a single graph in the list of graphs inputted. In the same way, columns represent the embedding dimensions.
- **Community detection procedures** return a dictionary when the `get_memberships` method is called. Node indices are keys and the values corresponding to the keys are the community memberships of vertices. Certain graph clustering techniques create a node embedding in order to find vertex clusters. These return a NumPy float array when the `get_embedding` method is called. This array is structured like the ones returned by node embedding algorithms.

We demonstrate the standardized output generation and interfacing by the code fragment in Figure 4. We create clusterings of a random graph and return dictionaries containing the cluster memberships. Using the external community library we can calculate the modularity of these clusterings (lines 15-16). This shows that the standardized output generation makes interfacing with external graph mining and machine learning libraries easy.

3.6 Limitations

The current design of Karate Club has certain limitations and we make strong assumptions about the input. We assume that the NetworkX graph is undirected and consists of a single strongly connected component. All algorithms assume that nodes are indexed with integers consecutively and the starting node index is 0. Moreover, we assume that the graph is not multipartite, nodes are homogeneous and edges are unweighted (each edge has a unit weight).

In case of the whole graph embedding algorithms [7, 8, 12, 19, 38, 40] all graphs in the set of graphs must amend the previously listed requirements with respect to the input. The Weisfeiler-Lehman feature based embedding techniques [7, 19] allow nodes to have a single string feature which can be accessed with the `feature` key. Without the presence of this key these algorithms default to the use of degree centrality as a node feature.

4 EXPERIMENTAL EVALUATION

In the experimental evaluation of Karate Club we will demonstrate two things. First, we will show that the implemented algorithms have a good performance with respect to embedding and extracted community quality on a variety of machine learning problems. Second, we support evidence that those algorithms which in theory scale linearly with the input size (number of nodes or number of graphs) scale linearly using our framework in practice. Throughout these experiments we will always use the standard hyperparameter settings of the 1.0. release of our package.

4.1 Learning performance

The evaluation of the representation quality focuses on three types of machine learning tasks. These are: community detection with ground truth communities, node classification with the node embeddings, and whole graph classification with graph level embeddings.

4.1.1 Datasets. In order to evaluate the performance of vertex level algorithms (node embedding and community detection) we used attributed web, collaboration and social networks which are publicly available on SNAP [17, 30]. We decided to use attributed networks because a large number of algorithms in Karate Club can exploit the presence of node features. These datasets are the following:

- MultiQ: A dataset of multi-relational attributed networks where each node has a set of attributes and each edge has a set of relations.
- WebKB: A dataset of web pages where each node has a set of attributes.
- Cora: A dataset of citation networks where each node has a set of attributes.
- Citeseer: A dataset of citation networks where each node has a set of attributes.
- Pubmed: A dataset of citation networks where each node has a set of attributes.
- **Wikipedia Crocodiles**: In this graph nodes represent Wikipedia pages and edges are mutual links. The vertex features describe the presence of nouns in the article and the binary target variable indicates the volume of traffic on the site.
- **GitHub Developers**: Vertices in this network are developers who use GitHub and edges represent mutual follower relationships between the users. Features are derived based on location, biography and other metadata, the binary target variable is whether someone is a machine learning or web developer.
- **Twitch England**: Nodes of this graph are Twitch users from England and edges are mutual friendships between them. Node features were extracted based on the streaming history of the users while the binary node class describes whether the user creates explicit content.
- **Facebook Page-Page**: A network of verified Facebook pages where nodes are pages and the links between nodes are mutual likes. Features are distilled from the page descriptions and the target is the category of the Facebook page (Politicians, Governments, Companies, TV Shows).

The descriptive statistics of these node level datasets are summarized in Table 1. As one can see these networks have a large variety of size, level of clustering and diameter.

### Table 1: The social networks used for node level algorithms.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Nodes</th>
<th>Density</th>
<th>Transitivity</th>
<th>Diameter</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wikipedia Crocodiles</td>
<td>11,631</td>
<td>0.003</td>
<td>0.026</td>
<td>11</td>
<td>13,183</td>
</tr>
<tr>
<td>GitHub Developers</td>
<td>37,700</td>
<td>0.001</td>
<td>0.013</td>
<td>7</td>
<td>4,005</td>
</tr>
<tr>
<td>Twitch England</td>
<td>7,126</td>
<td>0.002</td>
<td>0.042</td>
<td>3</td>
<td>2,545</td>
</tr>
<tr>
<td>Facebook Page-Page</td>
<td>22,470</td>
<td>0.001</td>
<td>0.232</td>
<td>15</td>
<td>4,714</td>
</tr>
</tbody>
</table>

Graph level embedding algorithms were evaluated on a variety of web and social graph datasets which we collected specifically for this paper. We made these graph collections publicly available.\(^1\) The graph collections used for predictive performance evaluation are the following:

- **Reddit Threads**: Discussion and non-discussion based threads from Reddit which we collected in May 2018. The task is to predict whether a thread is discussion based.
- **Twitch Egos**: The ego-nets of Twitch users who participated in the partnership program in April 2018. The binary classification task is to predict using the ego-net whether the central gamer plays a single or multiple games.
- **GitHub StarsGazers**: The social networks of developers who starred popular machine learning and web development repositories until 2019 August. The task is to decide whether a social network belongs to a web or machine learning repository.
- **Deezer Egos**: The ego-nets of Eastern European users collected from the music streaming service Deezer in February 2020. The related task is the prediction of gender for the ego node in the graph.

We listed the size of these datasets with the respective descriptive statistics in Table 2. It is worth noting that the Reddit Threads and Twitch Egos both have at least 10 fold more graphs than the social graph datasets which are widely used for graph classification evaluation [44]. We would also like to emphasize that the use of graph kernels would not be feasible on graph datasets which are this numerous.

### Table 3: Mean NMI values with standard errors on the node level datasets calculated from 100 runs.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Wikipedia Crocodiles</th>
<th>GitHub Developers</th>
<th>Twitch England</th>
<th>Facebook Page-Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>BANMF [51]</td>
<td>0.65 ± 0.01</td>
<td>0.03 ± 0.01</td>
<td>0.07 ± 0.01</td>
<td>0.16 ± 0.01</td>
</tr>
<tr>
<td>M-NMF [43]</td>
<td>0.63 ± 0.01</td>
<td>0.04 ± 0.01</td>
<td>0.04 ± 0.01</td>
<td>0.06 ± 0.01</td>
</tr>
<tr>
<td>NNSED [36]</td>
<td>0.63 ± 0.01</td>
<td>0.03 ± 0.01</td>
<td>0.04 ± 0.01</td>
<td>0.07 ± 0.01</td>
</tr>
<tr>
<td>SymmNMF [16]</td>
<td>0.62 ± 0.01</td>
<td>0.07 ± 0.01</td>
<td>0.07 ± 0.01</td>
<td>0.20 ± 0.01</td>
</tr>
<tr>
<td>Ego-Splitting [11]</td>
<td>0.17 ± 0.01</td>
<td>0.20 ± 0.01</td>
<td>0.23 ± 0.01</td>
<td>0.34 ± 0.01</td>
</tr>
<tr>
<td>EdMot [18]</td>
<td>0.85 ± 0.01</td>
<td>0.18 ± 0.01</td>
<td>0.08 ± 0.01</td>
<td>0.27 ± 0.01</td>
</tr>
<tr>
<td>LabelProp [28]</td>
<td>0.12 ± 0.01</td>
<td>0.09 ± 0.02</td>
<td>0.03 ± 0.01</td>
<td>0.32 ± 0.01</td>
</tr>
<tr>
<td>SCD [26]</td>
<td>0.18 ± 0.01</td>
<td>0.19 ± 0.01</td>
<td>0.16 ± 0.01</td>
<td>0.36 ± 0.01</td>
</tr>
<tr>
<td>GEMSEC [31]</td>
<td>0.10 ± 0.01</td>
<td>0.12 ± 0.01</td>
<td>0.08 ± 0.01</td>
<td>0.24 ± 0.01</td>
</tr>
</tbody>
</table>

4.1.2 Community Detection. We evaluate the community detection performance by running the clustering algorithms on the node level datasets. In case of overlapping community detection algorithms [11, 16, 36, 43, 49, 51] we assigned each node to the cluster that has the strongest affiliation score with the node (ties were broken randomly). The metric used for the clustering performance measurement is the average normalized mutual information (henceforth NMI) score calculated between the cluster membership vector and the factual class memberships. We report in Table 3 the NMI averages with the standard errors calculated from 100 experimental runs.

Looking at Table 3 first we notice that the non-overlapping community detection techniques [11, 18, 26, 28, 31] materially outperform the overlapping models which create latent spaces [16, 36, 43, 49, 51] on every dataset in terms of NMI. Second, those algorithms which create clusters based on the presence of closed triangles (SCD [26], Ego-Splitting [11]) have a general strong performance. Finally, on problems where it can be assumed that the class membership vector is associated with structural properties (e.g. Wikipedia Crocodiles), the overlapping latent space creating community detection methods perform poorly in terms of NMI.

\(^1\)https://snap.stanford.edu/data/
4.1.3 Graph classification. In each dataset we created representations for the graphs and used those as predictors for the downstream classification task. We repeated the feature distillation and supervised model training 100 times, used 80% of graphs for training and 20% for testing with seeded splits. Using the graph class vectors of the test set and class probabilities outputted by the logistic regression classifier we calculated mean area under the curve (henceforth AUC) values which are presented in Table 4 along with their standard errors.

Table 4: Mean AUC values with standard errors on the graph level datasets calculated from 100 seed train-test splits.

<table>
<thead>
<tr>
<th>Method</th>
<th>Reddit Threads</th>
<th>Twitch Egos</th>
<th>Github StarGazers</th>
<th>Deezer Egos</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL2Vec [7]</td>
<td>0.73 ± 0.002</td>
<td>0.64 ± 0.002</td>
<td>0.55 ± 0.001</td>
<td>0.50 ± 0.001</td>
</tr>
<tr>
<td>Graph2Vec [19]</td>
<td>0.80 ± 0.002</td>
<td>0.70 ± 0.003</td>
<td>0.58 ± 0.001</td>
<td>0.52 ± 0.001</td>
</tr>
<tr>
<td>SF [8]</td>
<td>0.84 ± 0.002</td>
<td>0.68 ± 0.003</td>
<td>0.58 ± 0.001</td>
<td>0.50 ± 0.001</td>
</tr>
<tr>
<td>NetLSD [38]</td>
<td>0.87 ± 0.001</td>
<td>0.63 ± 0.002</td>
<td>0.63 ± 0.001</td>
<td>0.52 ± 0.001</td>
</tr>
<tr>
<td>FGSD [40]</td>
<td>0.82 ± 0.002</td>
<td>0.70 ± 0.003</td>
<td>0.66 ± 0.001</td>
<td>0.52 ± 0.001</td>
</tr>
<tr>
<td>GeoScattering [12]</td>
<td>0.80 ± 0.001</td>
<td>0.67 ± 0.001</td>
<td>0.54 ± 0.003</td>
<td>0.52 ± 0.003</td>
</tr>
<tr>
<td>FEATHER [34]</td>
<td>0.83 ± 0.002</td>
<td>0.72 ± 0.003</td>
<td>0.74 ± 0.002</td>
<td>0.54 ± 0.001</td>
</tr>
</tbody>
</table>

The results presented in Table 4 show that the representations created by implicit factorization [7, 19] and spectral finger printing [8, 34, 38, 40] techniques are predictive on most problems. In addition, we see evidence that algorithms from the latter group create somewhat higher quality representations.

4.1.4 Node classification. In this series of experiments we evaluated the node classification performance on the node level datasets. For each graph we learned a node embedding and used the features of this node embedding as predictors for a downstream logistic regression model. We repeated the embedding and supervised model training 100 times, used 80% of the nodes for training and 20% for testing with seeded splits. Using the target vectors of the test set and the class probabilities outputted by the downstream model we calculated mean AUC scores. These average AUC values are reported in Table 5 with standard errors. The results in Table 5 generally demonstrate that the included neighbourhood based [4, 6, 15, 20, 24, 25, 27, 33, 37], structural role preserving [2, 10], and attributed [3, 30, 45, 48, 50, 53] node embedding techniques all generate reasonable quality representations for this classification task. There are additional conclusions; (i) multi-scale node embeddings such as GraRep [6], Walklets, [25], and MUSAE [30] create high quality node features, (ii) combining neighbourhood and attribute information results in the best representations [30, 53], (iii) there is not a single model which is generally superior.

4.2 Scalability

We perform scalability tests for all three types of algorithms (community detection, node and whole graph embedding). For each of these categories we investigate the scalability of 4 chosen algorithms. We use Erdos-Renyi graphs where the input size and graph density can be manipulated directly.

Table 5: Mean AUC values with standard errors on the node level datasets calculated from 100 seed train-test splits.

<table>
<thead>
<tr>
<th>Method</th>
<th>Wikipedia Crocodiles</th>
<th>GitHub Developers</th>
<th>Twitch England</th>
<th>Facebook Page-Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>BoosINE [15]</td>
<td>.685 ± .001</td>
<td>.845 ± .001</td>
<td>.596 ± .001</td>
<td>.722 ± .001</td>
</tr>
<tr>
<td>NodeSketch [47]</td>
<td>.722 ± .001</td>
<td>.631 ± .001</td>
<td>.520 ± .001</td>
<td>.579 ± .001</td>
</tr>
<tr>
<td>Diff2Vec [33]</td>
<td>.832 ± .001</td>
<td>.858 ± .001</td>
<td>.589 ± .001</td>
<td>.873 ± .001</td>
</tr>
<tr>
<td>NetMF [27]</td>
<td>.866 ± .001</td>
<td>.867 ± .001</td>
<td>.629 ± .002</td>
<td>.946 ± .001</td>
</tr>
<tr>
<td>Walklets [25]</td>
<td>.875 ± .001</td>
<td>.899 ± .002</td>
<td>.622 ± .001</td>
<td>.973 ± .001</td>
</tr>
<tr>
<td>HOPE [26]</td>
<td>.870 ± .001</td>
<td>.844 ± .001</td>
<td>.612 ± .002</td>
<td>.909 ± .001</td>
</tr>
<tr>
<td>NMF-ADMM [37]</td>
<td>.747 ± .001</td>
<td>.784 ± .001</td>
<td>.619 ± .001</td>
<td>.937 ± .001</td>
</tr>
<tr>
<td>LAP [4]</td>
<td>.784 ± .001</td>
<td>.529 ± .001</td>
<td>.511 ± .001</td>
<td>.501 ± .001</td>
</tr>
<tr>
<td>GraphWave [10]</td>
<td>.517 ± .001</td>
<td>.620 ± .001</td>
<td>.583 ± .001</td>
<td>.613 ± .001</td>
</tr>
<tr>
<td>BANE [48]</td>
<td>.866 ± .002</td>
<td>.570 ± .001</td>
<td>.551 ± .001</td>
<td>.970 ± .002</td>
</tr>
<tr>
<td>TENE [50]</td>
<td>.907 ± .001</td>
<td>.874 ± .001</td>
<td>.615 ± .001</td>
<td>.886 ± .001</td>
</tr>
<tr>
<td>TADW [45]</td>
<td>.896 ± .001</td>
<td>.817 ± .001</td>
<td>.612 ± .002</td>
<td>.871 ± .001</td>
</tr>
<tr>
<td>FSCNMF [3]</td>
<td>.912 ± .001</td>
<td>.856 ± .002</td>
<td>.621 ± .001</td>
<td>.891 ± .001</td>
</tr>
<tr>
<td>SINE [53]</td>
<td>.904 ± .001</td>
<td>.910 ± .002</td>
<td>.646 ± .001</td>
<td>.979 ± .001</td>
</tr>
<tr>
<td>MUSAE [30]</td>
<td>.931 ± .001</td>
<td>.903 ± .001</td>
<td>.628 ± .001</td>
<td>.981 ± .001</td>
</tr>
</tbody>
</table>

Figure 5 plots runtime against size and density of the clustered while the average number of edges is fixed to 10. In the densification scenario we clustered a graph with $2^12$ nodes. Non-overlapping community detection techniques show a remarkable scalability with respect to graph size increase, and we also see that the densification of the graph results in longer runtimes.

Figure 5: Scalability of the community detection procedures in Karate Club. We vary the number of nodes and the density of an Erdos-Renyi graph.

We measured the same way how the average runtime of node embedding varies with input size changes and densification and plotted these in Figure 6. These results show that under no preferential attachment all of the included methods scale linearly with input size changes. Moreover, implicit factorization runtimes are unaffected by the densification of the graph.

In case of the whole graph representation we plotted the average runtime as a function of the number of graphs and their size on
The libraries because of lightweight prerequisites and wide coverage of the learning techniques. First, the SNAP and GraphTool packages both have C++ prerequisites which have to be pre-compiled and installed. Our framework only has Python dependencies and builds on top of the NetworkX project. Second, the SNAP [17] library only covers specific methods which were created by the authors of the framework. The NetworkX [13] and GraphTool [23] libraries only provide community detection tools. Node and whole graph embedding is not supported by these frameworks.

6 CONCLUSION AND FUTURE DIRECTIONS

In this work we described Karate Club a Python framework built on the open source packages NetworkX [13], PyGSP [9], Gensim [29], NumPy [42], and SciPy Sparse [41] which performs unsupervised learning on graph data. Specifically, it supports community detection, node embedding, and whole graph embedding techniques.

We discussed in detail the design principles which we followed when we created Karate Club, standard hyperparameter encapsulation, the assumptions about the format of input data and generated output, and available public methods. In order to demonstrate these principles we included illustrative examples of code. In a series of experiments on real world datasets we validated that the machine learning models in Karate Club produce high quality clusters and embeddings. We also demonstrated on synthetic data that the linear runtime algorithms scale well with increasing input size.

As discussed, Karate Club has certain limitations with regards to the types of graphs that it can handle. In the future we plan to extend it to operate on directed and weighted graphs. Another aim is to provide a general framework for unsupervised learning algorithms on heterogeneous, multiplex, temporal graphs and procedures for the hyperbolic embedding of nodes [35, 39].

ACKNOWLEDGEMENTS

Benedek Rozemberczki was supported by the Centre for Doctoral Training in Data Science, funded by EPSRC (grant EP/L016427/1).

REFERENCES
