ABSTRACT
Signed networks are ubiquitous in many real-world applications (e.g., social networks encoding trust/distrust relationships, correlation networks arising from time series data). While many signed networks are directed, there is a lack of survey papers and software packages on graph neural networks (GNNs) specially designed for directed networks. In this paper, we present PyTorch Geometric Signed Directed, a survey and software on GNNs for signed and directed networks. We review typical tasks, loss functions and evaluation metrics in the analysis of signed and directed networks, discuss data used in related experiments, and provide an overview of methods proposed. The deep learning framework consists of easy-to-use GNN models, synthetic and real-world data, as well as task-specific evaluation metrics and loss functions for signed and directed networks. The software is presented in a modular fashion, so that signed and directed networks can also be treated separately. As an extension library for PyTorch Geometric, our proposed software is maintained with open-source releases, detailed documentation, continuous integration, unit tests and code coverage checks. Our code is publicly available at https://github.com/SherylHYX/pytorch_geometric_signed_directed.

CCS CONCEPTS
• Software and its engineering → Designing software; • Information systems → Information retrieval; • Computing methodologies → Machine learning.

KEYWORDS
graph neural networks, open-source software, signed networks, directed networks, survey, machine learning, deep learning, PyTorch

1 INTRODUCTION
With an increasing number of applications where data are generated from non-Euclidean domains and are represented as graphs (e.g., social networks, citation networks, and biochemical graphs), graph data, which contains rich relation information, are related to many learning problems [75, 82]. To tackle network inference tasks, graph neural networks (GNNs) are a useful tool. Based on deep learning techniques such as network architecture, optimization approaches and parallel computation, GNNs can be trained just like standard neural networks. By leveraging the network structure, GNNs are able to retain information from the nodes’ neighborhood and incorporate long-range dependencies [82]. GNNs have a wide range of applications, such as social network analysis [28], traffic predictions [41], biological applications [38], recommender systems, computer vision, and natural language processing [75].

Although traditional network analysis usually focuses on a single fixed simple network, which could often be represented by a symmetric adjacency matrix with nonnegative entries, intricate network types are often more realistic. Signed networks, which have positive or negative edge weights, have been of interest in social network analysis [31, 58], with signs indicating positive or negative sentiments. Clustering time series is another task that has motivated the development of signed networks algorithms [3], including economic time series that capture macroeconomic variables [19], and financial time series [52, 83]. The empirical correlation matrix can be construed as a weighted signed network, with signs indicating correlations, potentially sparsified after thresholding on the p-value of individual entries, following statistical significance tests. Many signed networks are themselves directed, in that they have asymmetric sending and receiving patterns. Indeed, edge directionality should play an integral role in processing and learning from...
network data [45], and directed networks have many applications such as clustering time-series data with lead-lag relationships [4], detecting influential groups in social networks [7, 27, 79], ranking [26], as well as transportation and biological applications [45].

Survey papers are helpful in reviewing concepts and progress in a field. There are several survey papers on signed networks [20, 62] and directed networks [43, 45], but few are on signed or directed GNNs. The only such work we are aware of is [72], which is, however, restricted exclusively to financial applications. To the best of our knowledge, there has not been a survey paper thus far on GNNs that are specifically designed for directed networks (digraphs), and hence this paper is the first attempt to review GNNs designed for general digraphs, as well as the first survey that broadly reviews signed GNNs that are not restricted to financial applications.

On the other hand, deep learning frameworks [1, 10, 33, 51] have facilitated the emergence of open-source software on deep learning on graphs [15, 18, 22, 29, 57, 80]. The existence of such open-source libraries is essential to the practical success and large-scale deployment of graph machine learning systems [57]. However, despite the popularity of signed and directed networks, there does not exist an open-source library to include GNNs as well as related data processing methods on them. This motivates the design of our open-source software, called PyTorch Geometric Signed Directed, which is an extension library of the popular PyTorch Geometric [18], but is specially designed for signed and directed networks.

**Main contributions**. Our main contributions are as follows:

1. We present PyTorch Geometric Signed Directed, the first deep learning software for graph neural networks in signed and directed networks, which is equipped with open-source software and public releases, detailed documentation, code examples, continuous integration, unit tests and code coverage checks.

2. We provide easy-to-use GNN models, synthetic data generators and real-world data loaders, as well as task-specific evaluation metrics and loss functions for signed and directed networks in PyTorch Geometric Signed Directed. The software is presented in a modular fashion, so that signed and directed networks can also be treated separately.

3. We review typical tasks, loss functions and evaluation metrics in the analysis of signed and directed networks, discuss data used in related experiments, and provide an overview of methods proposed.

4. We evaluate task performance of GNNs available in PyTorch Geometric Signed Directed on synthetic and real-world data sets.

## 2 Typical Tasks, Loss Functions and Evaluation Metrics

In this section, we review the typical problem setup, discuss typical tasks and loss functions as well as evaluation metrics, in the field of signed and directed graph neural networks.

The setup in this paper is as follows. Let \( G = (V, E, w, X_V) \) denote a (possibly signed, directed and weighted) network with node attributes, where \( V \) is the set of nodes, \( E \) is the set of (directed) edges or links, and \( w \in (-\infty, \infty)^{|E|} \) is the set of weights of the edges. Here, \( G \) could have self-loops, but not multiple edges. The total number of nodes is \( n = |V| \), and \( X_V \in \mathbb{R}^{n \times d_{in}} \) is a matrix whose rows are node attributes (which could be generated from \( A \)), where \( d_{in} \) denotes the input dimension of the node attributes. This network can be represented by the attribute matrix \( X_V \) and adjacency matrix \( A = (A_{ij})_{i,j \in V} \), where \( A_{ij} = w_{ij} \), the edge weight, if there is an edge from node \( u \) to node \( v \); otherwise \( A_{ij} = 0 \).

Typical tasks in such a setup include: *(1) link prediction, (2) node clustering and node classification. These tasks can be carried out in a supervised, semi-supervised, or unsupervised setting. Here we do not include tasks which relate to groups of networks or to synthetic network generation.*

### 2.1 Link Prediction

*For signed networks,* a typical task is to predict the sign of an existing edge. A typical GNN pipeline would first learn node embeddings based on some loss function dependent on, for example, node embeddings and edge signs [16], or binary cross-entropy [31], and then apply a binary classifier to output the final predictions.

*For directed networks,* there are three types of link prediction tasks: *(1) Direction prediction: predict the edge direction of pairs of vertices \( u, v \) for which either \((u, v) \in E \) or \((v, u) \in E \).

(2) Existence prediction: predict if \((u, v) \in E \) by considering ordered pairs of vertices \((u, v) \).

(3) Three-class classification: classify an edge \((u, v) \in E \), \((v, u) \in E \), or \((u, v), (v, u) \notin E \). When treated as a classification problem, the cross-entropy loss function can be employed [79].

Classification results are typically evaluated with accuracy, Area Under Curve (AUC) [8] and F1 score [60].

### 2.2 Node Clustering/Classification

A clustering/classification into \( K \) clusters is a partition of the node set into disjoint sets \( V = C_0 \cup C_1 \cup \cdots \cup C_{K-1} \). Intuitively, nodes within a cluster should be similar to each other, while nodes across clusters should be dissimilar.

In a semi-supervised setting, for each of the \( K \) clusters, a fraction of training nodes are selected as seed nodes, for which the cluster membership labels are known before training. The set of seed nodes is denoted as \( \mathcal{V}^{seed} \subseteq \mathcal{V}^{train} \subseteq \mathcal{V} \), where \( \mathcal{V}^{train} \) is the set of all training nodes. For this task, the goal is to use the embedding for assigning each node \( v \in \mathcal{V} \) to a cluster containing known seed nodes. When no seeds are given, we are in a self-supervised setting, where only the number of clusters, \( K \), is given. For a node classification problem, seed nodes are usually the same as training nodes, so we do not mention seeds in the rest of the paper when we are not in a node clustering setting.

The quality of a clustering partition is often assessed through a modularity objective function [67] which compares the partition to that expected under a null model for the network with the assumption that nodes within a cluster are relatively more densely connected than nodes across clusters. However, depending on the task at hand, *similarity* could have different meanings [25]. In a signed network with positive and negative edges, similarity may relate to the neighborhood of a node such as the proportion of shared friends or enemies. In a digraph, nodes could also be clustered depending on their position within a directed flow on the network, see [27].

A GNN’s objective plays an essential role in guiding the GNN to learn. In a node clustering/classification task, when seed nodes with known cluster labels are available, the cross entropy loss function is usually applied. In [28], a contrastive loss function based on triplets...
of the nodes is used to push embeddings of nodes within clusters/classes to be closer to each other than those across clusters/classes. For unsupervised tasks, however, objectives independent of known labels are required. To train a GNN model, differentiable loss functions are used. A probabilistic version of the balanced normalized cut [11] loss is proposed in [28], while a probabilistic version of flow imbalance loss is introduced in [27].

Node classification performance can be evaluated via accuracy. For a node clustering problem, as cluster indices could be permuted, accuracy would not be an appropriate evaluation measure. Instead, the Adjusted Rand Index (ARI) [32] is often chosen, which is invariant to label permutation. Depending on the downstream task, other measures could be used, such as the unhappy ratio by [28], and the imbalance scores by [27].

3 DATA

3.1 Synthetic Data

3.1.1 Signed Stochastic Block Model (SSBM). A Signed Stochastic Block Model (SSBM) [28] for a network on $n$ nodes with $K$ blocks (clusters), is constructed as follows: *(1) Assign block sizes $n_i \leq n_1 \leq \cdots \leq n_{K-1}$ with size ratio $\rho \geq 1$, as follows. If $\rho = 1$, then the first $K-1$ blocks have the same size $\lceil n/K \rceil$, and the last block has size $n - (K-1) \lceil n/K \rceil$. If $\rho > 1$, we set $\rho_0 = \rho^{1 + \frac{1}{K}}$. Then, $\rho_0 n_0 = n$, and taking integer value gives $n_0 = \lceil n(1 - \rho_0)/(1 - \rho_0^K) \rceil$. Further, set $n_i = \lceil \rho n_{i-1} \rceil$, for $i = 1, \dotsc, K-2$ if $K \geq 3$, and $n_{K-1} = n - \sum_{i=0}^{K-2} n_i$. Then, the ratio of the size of the largest to the smallest block is approximately $\rho_0^{K-1} = \rho$. *(2) Assign each node to one of $K$ blocks, so that each block has the allocated size. *(3) For each pair of nodes in the same block, with probability $p_{\text{in}}$, create an edge with $+1$ as weight between them, independently of the other potential edges. *(4) For each pair of nodes in different blocks, with probability $p_{\text{out}}$, create an edge with $-1$ as weight between them, independently of the other potential edges. *(5) Flip the sign of the across-cluster edges from the previous stage with sign flip probability $\eta_{\text{out}}$, and $\eta_{\text{out}}$ for edges within and across clusters, respectively.

3.1.2 Polarized SSBMs. In a polarized SSBM model [28], SSBMs are planted in an ambient network; each edge of each SSBM is a cluster, and the nodes not assigned to any SSBM form an ambient cluster. The polarized SSBM model that creates communities of SSBMs, is constructed as follows: *(1) Generate an Erdős–Rényi graph with $n$ nodes and edge probability $p$, whose sign is set to $\pm 1$ with equal probability $0.5$. *(2) Fix $n_c$ as the number of SSBM communities, and calculate community sizes $n_1 \leq n_2 \leq \cdots \leq n_r$, for each of the $r$ communities as in Sec. 3.1.1, such that the ratio of the largest block size to the smallest block size is approximately $\rho$, and the total number of nodes in these SSBMs is $N \times n_c$. *(3) Generate $r$ SSBM models, each with $K_i = 2, i = 1, \dotsc, r$ blocks, number of nodes according to its community size, with the same edge probability $p$, size ratio $\rho$, and flip probability $\eta$. *(4) Place the SSBM models on disjoint subsets of the whole network; the remaining nodes not part of any SSBM are dubbed as ambient nodes. The resulting polarized SSBM model is denoted as Pol-SSBM ($n, r, \rho, p, \eta, N$).

3.1.3 Directed SSBM. A standard directed stochastic blockmodel (DSBM) is often used to represent a network cluster structure, see for example [43]. A DSBM model relies on a meta-graph adjacency matrix $F = (F_{i,j})_{i,j=0,\ldots,K-1}$ and a filled version of it, $\tilde{F} = (\tilde{F}_{i,j})_{i,j=0,\ldots,K-1}$, and on a noise level parameter $\eta \leq 0.5$. The meta-graph adjacency matrix $F$ is generated from the given meta-graph structure, called $M$. To include an ambient background, the filled meta-graph adjacency matrix $\tilde{F}$ replaces every zero in $F$ that is not part of the imbalance structure by $0.5$. The filled meta-graph thus creates a number of ambient nodes which correspond to entries which are not part of $M$ and thus are not part of a meaningful cluster; this set of ambient nodes is also called the ambient cluster. First, we provide examples of structures of $F$ without any ambient nodes, where $\eta$ denotes the indicator function: *(1) "cycle": $F_{k,l} = (1 - \eta)1(l = ((k + 1) \mod K)) + \eta1(l = ((k - 1) \mod K)) + \frac{1}{2}1(l = k)$. *(2) "path": $F_{k,l} = (1 - \eta)1(l = k + 1) + \eta1(l = k - 1) + \frac{1}{2}1(l = k)$. *(3) "complete": assign diagonal entries $\frac{1}{4}$. For each pair $(k, l)$ with $k < l$, let $F_{k,l}$ be $\eta$ and $1 - \eta$ with equal probability, then assign $F_{k,l} = 1 - F_{k,l}$. *(4) "star", following [17]: select the center node as $\omega = \frac{K - 1}{2}$ and set $F_{k,l} = (1 - \eta)1(l = \omega, 1) + \eta1(k = \omega, 1) + \frac{1}{2}1(l = \omega, 1 \text{ even}) + (1 - \eta)1(l = \omega, k \text{ odd}) + \frac{1}{2}1(l = \omega, k \text{ even})$. When ambient nodes are present, the construction involves two steps, with the first step the same as the above, but with the following changes: For "cycle" meta-graph structure, $F_{k,l} = (1 - \eta)1(l = ((k + 1) \mod (K - 1)) + \eta1(l = ((k - 1) \mod (K - 1)) + 0.51(l = k)$. The second step is to assign $0 (0.5, \text{resp.})$ to the last row and the last column of $F$ (resp.).

A DSBM model, denoted by DSBM ($M, \eta$ (ambient), $n, K, p, \rho, \eta$), is built as follows: *(1-2) Same as in (1-2) in Sec. 3.1.1. *(3) For nodes $v_i, v_j \in C_k$, independently sample an edge from $v_i$ to $v_j$ with probability $p \cdot \tilde{F}_{k,k}$. *(4) For each pair of different clusters $C_k, C_l$ with $k \neq l$, for each node $v_i \in C_k$, and each node $v_j \in C_l$, independently sample an edge from $v_i$ to $v_j$ with probability $p \cdot \tilde{F}_{k,l}$.

3.2 Real-world Data

Signed networks. For signed networks, we provide data loaders for the following data sets. *(The Sampson monastery data [58] cover four social relationships, each of which could be positive or negative. We combine these relationships into a network of 25 nodes. This data set is the only one of the above for which we use a node attribute, namely whether or not they attended the minor seminary of "Cloisterville". As ground truth we take Sampson’s division of the novices into four groups: Young Turks, Loyal Opposition, Outcasts, and an interstitial group. *(Rainfall [5] contains Australian rainfalls pairwise correlations. *(FinYNet [76] consists of yearly correlation matrices for $n = 451$ stocks for 2000-2020 (21 distinct networks), using so-called market excess returns; that is, we compute each correlation matrix from overnight (previous close to open) and intraday (open-to-close) price daily returns, from which we subtract the market return of the S&P500 index. *(S&P500 [76] considers daily prices for $n = 1,193$ stocks, in the S&P 1500 Index, between 2003 and 2015, and builds correlation matrices also from market excess returns. The result is a fully-connected weighted network, with stocks as nodes and correlations as edge weights. *(PPI [74] is a signed protein-protein interaction network. *(Wiki-Rfa [71] is a signed network describing voting information for electing Wikipedia managers. Positive edges represent supporting votes, while negative edges...
Table 1: Summary statistics for the real-world networks that can be loaded by PyTorch Geometric Signed Directed. Here \( n \) is the number of nodes, \( |E^+| \) and \( |E^-| \) denote the number of positive and negative edges, respectively. For an unsigned network, \( |E^-| = 0 \). For a weighted graph, we mean that the set of edge weights only contains one value if we disregard signs. We also report whether the network is directed or weighted in the last two columns. Note that Fin-YNet and Lead-Lag are averaged over 21 and 19 financial networks, respectively.

| Data set     | \( n \) | \(|E^+|\) | \(|E^-|\) | Directed | Weighted |
|--------------|--------|--------|--------|----------|----------|
| Sampson      | 25     | 129    | 126    | ✗        | ✔        |
| Cornell      | 183    | 295    | 0      | ✔        | ✗        |
| Texas        | 183    | 309    | 0      | ✔        | ✗        |
| Wisconsin    | 251    | 499    | 0      | ✔        | ✗        |
| Telegram     | 245    | 8,912  | 0      | ✔        | ✗        |
| Lead-Lag     | 269    | 29,159 | 0      | ✔        | ✗        |
| Rainfall     | 306    | 64,408 | 29,228 | ✗        | ✔        |
| Fin-YNet     | 451    | 14,853 | 5,431  | ✔        | ✗        |
| Email        | 1,005  | 25,571 | 0      | ✔        | ✗        |
| Blog         | 1,222  | 19,024 | 0      | ✔        | ✗        |
| S&P 1500     | 1,193  | 1,069,319 | 353,930 | ✗        | ✔        |
| Chameleon    | 2,277  | 36,101 | 0      | ✔        | ✗        |
| Migration    | 3,075  | 72,143 | 0      | ✔        | ✗        |
| Cora-ML      | 2,995  | 8,416  | 0      | ✔        | ✗        |
| PPI          | 3,058  | 7,996  | 3,864  | ✔        | ✗        |
| CiteSeer     | 3,312  | 4,715  | 0      | ✔        | ✗        |
| BitCoin-Alpha| 3,783  | 22,650 | 1,536  | ✔        | ✗        |
| Squirrel     | 5,201  | 217,073| 0      | ✔        | ✗        |
| BitCoin-OTC  | 5,881  | 32,029 | 3,563  | ✔        | ✗        |
| Wiki-Rfa     | 7,634  | 136,961| 38,826 | ✔        | ✗        |
| WikiCS       | 11,701 | 216,123| 0      | ✔        | ✗        |
| Slashdot     | 82,140 | 425,072| 124,130| ✔        | ✗        |
| Epinions     | 131,828| 717,667| 123,705| ✔        | ✗        |
| WikiTalk     | 2,388,953| 5,018,445| 0      | ✔        | ✗        |

As real-world directed, unsigned networks, we also include the following data sets. *Blog* [2] records \(|E| = 19,024\) directed edges between \( n = 1,212 \) political blogs from the 2004 US presidential election. *Migration* [54] reports the number of people that migrated between pairs of counties in the US during 1995-2000. It involves \( n = 3,075 \) countries and \(|E| = 721,432\) directed edges after obtaining the largest weakly connected component. Since the original digraph has entries extremely large, to cope with these outliers, we preprocess the input network by \( A_{ij} = \frac{\max(A_{ij})}{\sum(A_{ij} > 0)}, \), \( i, j \in \{1, \ldots, n\} \), which follows the preprocessing of [14]. *WikiTalk* [37] contains all users and discussion from the inception of Wikipedia until Jan. 2008. The \( n = 2,388,953 \) nodes in the network represent Wikipedia users and a directed edge from node \( u_i \) to node \( u_j \) denotes that user \( i \) edited at least once a talk page of user \( j \). We extract the largest weakly connected component. *Telegram* [7] is a pairwise influence network between 245 Telegram channels with 8,912 links. Labels are generated from the method discussed in [7], with a total of four classes. *Cora-ML and CiteSeer* are popular citation networks with node labels corresponding to scientific subareas [6]. *Texas, Wisconsin, and Cornell* are WebKB datasets modeling links between websites at different universities [53]. *Chameleon and Squirrel* [56] represent links between Wikipedia pages related to chameleons and squirrels.

*WikiCS* [48] is a collection of Computer Science articles.

*Lead-Lag* [4] contains yearly lead-lag matrices from 269 stocks from 2001 to 2019. The lead-lag matrices are built from time series of daily price log returns, as detailed in [4]. The lead-lag metric for entry \((i, j)\) in the network encodes a measure of the extent to which stock \( i \) leads stock \( j \), and is obtained by applying a functional that computes the signed normalized area under the curve (auc) of the standard cross-correlation function (ccf). The resulting matrix is skew-symmetric, and entry \((i, j)\) quantifies the extent to which stock \( i \) leads or lags stocks \( j \), thus leading to a directed network interpretation. Starting from the skew-symmetric matrix, we further convert negative entries to zero, so that the resulting directed graph (digraph) can be directly fed into other methods; note that this step does not throw away any information, and is pursued only to render the representation of the digraph consistent with the format expected by all methods compared.

4 METHODS

In this section, we discuss relevant background for methods in signed and directed networks, and review GNNs currently implemented in PyTorch Geometric Signed Directed. Tab. 2 lists whether the implemented GNNs can deal with signed edges or directed edges, together with the tasks they are concerned with.

4.1 Signed

Within the last decade, the landscape of signed network analysis has been mainly dominated by non-GNN methods (in particular spectral methods), such as those based on the (potentially normalized) Signed Laplacian matrix [36] and its variations [47, 81]. Balanced...
Normalized Cut and Balanced Ratio Cut [11], and many others [13, 73, 78]. More recently, GNNs for signed networks emerged.

SGCN [16] proposes an information aggregation and propagation mechanism with the mean-pooling strategy for signed undirected networks based on social balance theory [24]. SiGAT [30] utilizes GAT [70] in embedding learning for signed directed networks and devises a motif-based GNN architecture based on balance theory and status theory. SNEA [39] proposes another graph attentional layer, which uses a masked self-attention mechanism, and designs an objective function for both framework optimization and node representation learning. SDGNN [31], though also using graph attention, is more efficient than SiGAT, and its objective functions can model not only the edge sign, but in addition other vital features such as edge directions and triangles. SSSNET [28] conducts the semi-supervised node clustering task in signed networks, with a novel aggregation scheme that is not based on the popular social balance theory [24] that many previous signed GNNs rely on.

5.2 Directed

Research on directed network analysis has been mainly spectral in nature, and centered around symmetrization-based techniques [42, 50, 59], but edge directionality itself can contain important information [14, 45]. Imbalanced flows in digraphs have been uncovered via Hermitian clustering [14] and motif-based techniques [69]. Recently, researchers have started applying GNNs to extract essential information from directed edges.

DGCN [66] uses first and second order proximity, constructs three Laplacians, but the method is space and speed-inefficient. DiGCN [65] simplifies DGCN, builds a directed Laplacian based on PageRank, and aggregates information dependent on higher-order proximity. We denote the variant with the so-called “inception blocks” as DiGCNNB, and the variant without “inception blocks” as DiGCN. MagNet [79] constructs a Hermitian matrix that encodes undirected geometric structure in the magnitude of its entries, and directional information in their phase. DiGCL [64] introduces a digraph data augmentation method called Laplacian perturbation and conducts digraph contrastive learning. DIGRAC [27] conducts digraph clustering based on flow imbalance measures, with novel imbalance objectives and evaluation metrics.

5 SOFTWARE DESIGN

In this section, we present an overview of our library PyTorch Geometric Signed Directed, discuss relevant design choices, provide detailed case study examples, and end the section with long-term maintenance and viability of the software.

5.1 Neural Network Layers and Methods

PyTorch Geometric Signed Directed is built on the existing high-level neural network layer classes from the PyTorch and PyTorch Geometric ecosystems. We construct neural network layers for signed and directed networks from more than ten GNN methods. The constructors of these layers use type hinting to enable the set-up of the hyperparameters. Users can build upon these layers and construct their own model, depending on the task at hand.

Building upon the layers, we also implement the full methods from various research papers, so that end-users can directly utilize full models. For example, users can either employ the MagNetConv layer and construct their own full architecture, or they can directly call MagNet_node_classification or MagNet_link_prediction if they are interested in applying MagNet [79] to the node classification or the link prediction task, respectively.

5.2 Data Structures

5.2.1 Network Generators. PyTorch Geometric Signed Directed includes synthetic network generators for models described in Sec. 3.1, i.e., SSBMs and Pol-SSBMs from [28], as well as DSBMs from [27].

5.2.2 Data Classes. To accommodate the distinct features of signed and directed networks, based on the Data class from PyTorch Geometric [18], our software introduces two data classes for signed and directed networks, respectively. These classes (called SignedData and DirectedData, respectively) provide properties for checking whether a network is signed (or directed, respectively), can be initialized to obtain custom data objects with either edge_index only, edge_index together with edge_weight, or the sparse adjacency matrix. Such a data object can also inherit attributes from another data object, where edge attributes are stored in COO format.

We also provide class functions to construct node features based on the edge information. For instance, we can set the node feature matrix as the stacked leading eigenvectors of the regularized adjacency matrix for a signed network as in [28], or as the stacked real and imaginary parts of top eigenvectors for a Hermitian matrix [14] constructed from the digraph adjacency matrix as in [27].

5.2.3 Data Loaders and Splitters. Our software provides easy-to-use data loaders for real-world data sets for signed and directed networks. The loaded data set is then transformed into a custom data object designed in this software.

To facilitate downstream tasks, we also provide splitters on the data objects. Our node splitter generates masks for training, validation, test and seed nodes, where seed nodes are a portion of the training set; such masks can be useful in e.g., the semi-supervised setting of [28]. Our link split utility function divides edges into training, validation and test groups. The splitters can either be called separately, or called as a data class function.

5.3 Task-Specific Evaluations and Utilities

Tasks on signed and directed networks may have custom loss functions and evaluation metrics, such as the probabilistic normalized cut loss introduced in [28]. Therefore, PyTorch Geometric Signed Directed implements these custom loss functions and evaluation metrics to enable researchers to utilize these functionalities easily. Our library also provides some utility functions introduced in our implemented research papers, so that end-users are able to reproduce the whole process of a paper including data preprocessing, training, testing and so on.

5.4 Case Study on Signed Networks

In this subsection, we overview a simple end-to-end machine learning pipeline designed with PyTorch Geometric Signed Directed for signed networks. These code snippets solve a signed clustering problem on a Signed Stochastic Block Model – clustering the nodes in the signed network into five groups. The pipeline consists of
from sklearn.metrics import adjusted_rand_score
import scipy.sparse as sp
from torch_geometric_signed_directed.nn import SSSNET_node_clustering as SSSNET
from torch_geometric_signed_directed.data import SignedData, SSBM
from torch_geometric_signed_directed.utils import node_split, extract_network, triplet_loss_node_classification

model.train()
Z, log_prob, _, prob = model(edge_index_p, edge_weight_p, edge_index_n, edge_weight_n, features)
loss_pbnc = loss_func_pbnc(prob[mask])
loss_triplet = triplet_loss_node_classification(y=y[mask], z=Z[mask], n_sample=500, thres=0.1)
loss_ce = loss_func_ce(log_prob[mask], y[mask])
loss = 50*(loss_ce + 0.1*loss_triplet) + loss_pbnc
optimizer.zero_grad()
optimizer.step()
train_ari = adjusted_rand_score(y[mask].cpu(), (torch.argmax(prob, dim=1)).cpu()[mask])
return loss.detach().item(), train_ari

def test(features, edge_index_p, edge_weight_p, edge_index_n, edge_weight_n, mask, y):
    model.eval()
    with torch.no_grad():
        prob = model(edge_index_p, edge_weight_p, edge_index_n, edge_weight_n, features)
        test_ari = adjusted_rand_score(y[mask].cpu(), (torch.argmax(prob, dim=1)).cpu()[mask])
    return test_ari

Listings 1: Preparation, data loading, and node splitting with PyTorch Geometric Signed Directed.

5.4.1 Preparation, Data Loading and Splitting. In Listings 1, as a first step, we import the essentials (lines 1-10). We then define the device to be used (lines 12-13). After that, we define default values to be used in the network generation process, generate the synthetic network and extract the largest connected component (lines 15-20), and use the SignedData class (line 21). As no node features are available initially, we use the set_signed_Laplacian_features() method to set up the node feature matrix (line 22). We then create a train-validation-test seed split of the node set by using the node splitting function and calculate separated positive and negative parts of the signed network to be stored inside the data object (lines 23-26). Finally, we move the data object to the device (line 27).

Listings 2: Defining the model and initialize one part of the supervised loss function.

5.4.2 Model Definition and Loss Initialization. In Listings 2, we initialize the cross-entropy loss function (line 1), construct the GNN model and map it to the device (lines 3-4).

5.4.3 Defining functions for training and evaluation. In Listings 3, we define the training and evaluation functions. Setting the model to be trainable (line 4), we obtain the node embedding matrix Z and cluster assignment probabilities prob and its logarithm log_prob with a forward pass of the model instance (lines 5-6). We then obtain the probabilistic balanced normalized cut loss (line 7), triplet loss (lines 8-9), and cross entropy loss (line 10) values. The weighted sum of the three losses serves as the training loss value (line 11). We then backpropagate and update the model parameters (lines 12-14). After that, we calculate the ARI of the training nodes (lines 15-16). Finally, we return the loss value and training ARI (line 17).

For evaluation (function test()), we do not set the model to be trainable (lines 21-22). With a forward pass (lines 23-24), we obtain the probability assignment matrix. Taking argmax for the probabilities, we obtain test ARI (lines 25-26) and return the result (line 27).
5.5.4 Running the experiment. We run the actual experiments in Listings 4. For each of the data splits (line 1), we first initialize the Adam optimizer [34] (lines 2-3). We then obtain the data split indices (lines 4-7), initialize the self-supervised loss function (lines 8-10), and start the training process (line 11). For each epoch, we apply the training function to obtain training loss and ARI score (lines 12-16), then evaluate with the test() function on validation nodes (lines 17-19). We then print the training and validation results (lines 20-23). After training, we obtain the test performance (lines 25-27) and print some logs (line 28). Finally, we reset the model parameters (line 29) and iterate to the next data split loop.

5.5.5 Case Study on Directed Networks

In this subsection, we overview a simple end-to-end machine learning pipeline designed with PyTorch Geometric Signed Directed for directed networks. These code snippets solve a link direction prediction problem on a real-world data set. The pipeline consists of data preparation, model definition, training, and evaluation phases.

```python
from sklearn.metrics import accuracy_score
import torch

from torch_geometric_signed_directed.utils import 
  link_class_split, in_out_degree
from torch_geometric_signed_directed.nn.directed import 
  MagNet_link_prediction
from torch_geometric_signed_directed.data import 
  load_directed_real_data

device = torch.device('cuda' if 
  torch.cuda.is_available() else 'cpu')
data = load_directed_real_data(dataset='webkb',
  root=path, name='cornell').to(device)
link_data = link_class_split(data, prob_val=0.15,
  prob_test=0.05, task = 'direction', device=device)
```

Listings 5: Preparation, data loading, and node splitting with PyTorch Geometric Signed Directed.

5.5.5.1 Preparation, Data Loading and Splitting. In Listings 5, after importing and defining the device (lines 1-12), we load the Directed-Data object for the selected data set and map it to the device (lines 14-15). We then create a train-validation-test split of the edge set by using the directed link splitting function (lines 16-17).

```python
model = MagNet_link_prediction(q=0.25, K=1, num_features=2,
  hidden=16, label_dim=2).to(device)
criterion = torch.nn.NLLLoss()
def train(X_real, X_img, y, edge_index, edge_weight, query_edges):
  optimizer = torch.optim.Adam(model.parameters(), lr=0.01,
    weight_decay=0.0005)
  edge_index = link_data[split]["graph"]
  edge_weight = link_data[split]["weights"]
  query_edges = link_data[split]["train"]['edges']
  y = link_data[split]["train"]['label']
  X_real = in_out_degree(edge_index, 
    size=len(data.x)).to(device)
  X_img = X_real.clone()
  query_val_edges = link_data[split]["val"]['edges']
```

Listings 6: Defining the model and initializing one part of the supervised loss function.

5.5.2 Model Definition and Loss Initialization. In Listings 6, we first construct the model instance (lines 1-2), then initialize the cross-entropy loss function (line 3).

```python
def train(X_real, X_img, y, edge_index, 
  edge_weight, query_edges):
  optimizer = torch.optim.Adam(model.parameters(), lr=0.01,
    weight_decay=0.0005)
  edge_index = link_data[split]["graph"]
  edge_weight = link_data[split]["weights"]
  query_edges = link_data[split]["train"]['edges']
  y = link_data[split]["train"]['label']
  X_real = in_out_degree(edge_index, 
    size=len(data.x)).to(device)
  X_img = X_real.clone()
  query_val_edges = link_data[split]["val"]['edges']
```

Listings 7: Defining the training and evaluation functions.

5.5.5.3 Defining functions for training and evaluation. In Listings 7, we define the training and evaluation functions. Setting the model to be trainable (line 3), we obtain edge class assignment probabilities with a forward pass of the model instance (lines 4-6). We then obtain the training loss value (line 7). After that, we backpropagate and update the model parameters (lines 8-10). Then, we calculate the accuracy of the training samples (lines 11-12). Finally, we return the loss value as well as the training accuracy (line 13).

For the evaluation function (function test()), we do not set the model to be trainable (lines 17-18). With a forward pass (lines 19-21), we obtain the probability assignment matrix. We then obtain test accuracy (line 22) and return the result (line 23).

```python
for split in list(link_data.keys()):
  optimizer = torch.optim.Adam(model.parameters(), lr=0.01,
    weight_decay=0.0005)
  edge_index = link_data[split]["graph"]
  edge_weight = link_data[split]["weights"]
  query_edges = link_data[split]["train"]['edges']
  y = link_data[split]["train"]['label']
  X_real = in_out_degree(edge_index, 
    size=len(data.x)).to(device)
  X_img = X_real.clone()
  query_val_edges = link_data[split]["val"]['edges']
```
We run the actual experiments in Listings 8. For each data split (line 1), we first initialize the optimizer (lines 2-3), then prepare data objects to be used (lines 4-12), before starting the training process (line 13). For each epoch, we apply the train() function to obtain training loss and accuracy (lines 14-15), then evaluate with the test() function on validation nodes (lines 16-17). We then print the training and validation results (lines 18-20). After training, we prepare test data (lines 22-23), obtain the test performance (lines 24-25), and print some logs (line 26). Finally, we reset the model parameters (line 27) before the next iteration.

5.6 Maintaining the Software

We maintain our software via open-source code, public releases, automatically updated documentation, code examples, continuous integration, and unit tests with almost 100% test coverage.

5.6.1 Open-Source Software and Public Releases. The source code of our software is publicly available on GitHub under the MIT license at https://github.com/SherylHYX/pytorch_geometric_signed_directed. The code repository provides contributing guidelines, issue templates and test instructions, which enables the public to contribute to the software and report any problems. The public releases of the library are also made available on the Python Package Index (PyPI) at https://pypi.org/project/torch-geometric-signed-directed/. Installation is thus possible via the pip command using the terminal.

5.6.2 Documentation. We equip our software with a publicly available documentation of the library at https://pytorch-geometric-signed-directed.readthedocs.io/en/latest/, which is automatically built and updated as we modify the main branch of the public GitHub repository. Our documentation covers signed and directed GNN layers as well as the full methods in a number of research papers, data classes specifically designed for signed and directed networks, synthetic data generators, data loaders for real-world datasets, node-level and edge-level splitters, as well as task-specific evaluation metrics, loss functions and utilities functions. The documentation also includes an in-depth installation guide and a tour to external resources.

5.6.3 Code Examples. In our GitHub repository, we present code examples for a number of papers whose methods have been implemented in our software. Our documentation also introduces examples on data structures, data loaders, and splitters.

5.6.4 Continuous Integration. We use the freely available GitHub Actions to provide continuous integration for PyTorch Geometric Signed Directed. When the code is updated on the main branch of the GitHub repository or there is a pull request to the main branch, the build process is triggered and the library is deployed on Linux, Windows, and macOS virtual machines, for different Python versions: 3.6, 3.7, 3.8 and 3.9.

5.6.5 Unit Tests and Code Coverage. We provide unit tests for all of the signed and directed graph neural network layers, task-specific loss functions, evaluation metrics and utility functions, as well as data classes and data loaders. These unit tests can either be executed locally using the source code, or executed through GitHub Actions when the continuous integration process is trigger. When unit tests are executed, a coverage report by CodeCov is also generated. We maintain high test coverage rate to almost 100%.

6 COMPARISON WITH EXISTING SOFTWARE

Similar to existing deep learning software on graphs, which are usually based on machine learning frameworks such as TensorFlow (TF) [1], PyTorch (TF) [51], MxNet (MX) [10] and JAX [33], PyTorch Geometric Signed Directed is built on the PyTorch ecosystem. We summarize the characteristics of related deep learning libraries on graphs in Table 3, comparing frameworks based on the backend, presence of methods designed for signed networks, presence of methods designed specially for general digraphs, and GPU support. Note that the digraph method [63] contained in CogDL [9] requires the digraph to be acyclic, which is not generally true.

| Library          | Backend | Signed | Directed | GPU |
|------------------|---------|--------|----------|-----|
| OpenNE [68]      | Custom  | ✗      | ✗        | ✗   |
| CogDL [9]        | PT      | ✗      | ✗        | ✗   |
| Spektral [22]    | TF      | ✗      | ✗        | ✗   |
| TF Geometric [29]| TF      | ✗      | ✗        | ✓   |
| StellarGraph [15]| TF      | ✗      | ✗        | ✗   |
| DGL [80]         | TF/PT/MX| ✗      | ✗        | ✓   |
| DIG [40]         | PT      | ✗      | ✗        | ✓   |
| Jraph [21]       | JAX     | ✗      | ✗        | ✓   |
| Graph-Learn [77] | TF/PT   | ✗      | ✗        | ✓   |
| PT Geometric Temporal [37] | PT | ✗      | ✗        | ✓   |
| PT Geometric [18] | PT      | ✗      | ✗        | ✓   |
| **Our Work**     | PT      | ✓      | ✓        | ✓   |

We are also aware of open-source libraries for signed/directed networks that do not contain GNN methods. Python-igraph [12] is a collection of network analysis tools with the emphasis on efficiency, portability and ease of use; NetworkX [23] is a Python package for
the creation, manipulation, and study of the structure, dynamics, and functions of complex networks; NetworkKit [61] is a growing open-source toolkit for large-scale network analysis; SigNet [13] contains spectral analysis methods on signed networks; CDLIB [55] is a Python software package that allows to extract, compare and evaluate communities from complex networks; EvalINE [44] is a Python library designed for assessing and comparing the performance of network embedding methods on link prediction, sign prediction, network reconstruction, node classification and visualization downstream tasks.

To the best of our knowledge, the proposed software is the first deep learning library which consists of GNN methods specifically designed for general digraphs with GPU acceleration.

7 EXPERIMENTAL EVALUATION

To demonstrate that the implemented methods can reproduce original papers’ performance, we adopt the settings in [31] for link sign prediction on SGCN, SiGAT, SNEA and SDGNN, settings in [28] and [27] respectively for the node clustering task, settings in [79] for MagNet, DGCN, DiGCN, and DiGCNIB, for digraph node classification and link-level tasks, and report the results in this section. For DiGCL results, we use the settings in [64] for Cora_ML and CiteSeer for node classification, and search parameters for the other data sets. Tab. 4 reproduces link sign prediction performance for four signed GNNs, Fig. 2a illustrates test ARI for SSSNET on a polarized SSBM model, Fig. 2b shows test ARI for DIGRAC on a DSBM model, while Tab. 5 presents results on digraph node classification and three link prediction tasks, where DiGCNIB is the “inception block” version of DiGCN introduced in [65]. Slight differences in the reported results and the results in the original papers are possible due to differences in data splits and random seed employed.

Table 4: Link sign prediction results (AUC, Macro-F1) for signed networks (%).

| BitCoin-Alpha | BitCoin-OTC | Slashdot | Epinions |
|---------------|-------------|----------|----------|
| SGCN (87.26, 65.05) (88.16, 73.99) (89.93, 72.14) (86.85, 75.47) | SGCN (86.26, 69.72) (88.17, 73.88) (85.12, 72.11) (89.26, 76.73) | SGCN (86.33, 66.76) (89.58, 73.92) (87.14, 75.10) (90.73, 78.47) | SGCN (88.75, 74.17) (89.68, 79.52) (86.89, 76.55) (90.41, 81.22) |

8 CONCLUSION AND OUTLOOK

We present PyTorch Geometric Signed Directed, an easy-to-use and easy-to-adapt end-to-end software which addresses three main tasks in the analysis of such networks: node classification, clustering, and edge prediction. In network science, alternative methods are available for these tasks; the NetworkX [21] suite in Python provides a range of tools. In this paper, however, we concentrate on GNNs to survey and implement recent developments in this area, thus providing an easy entry point for users.

As an extension library to PyTorch Geometric, PyTorch Geometric Signed Directed is open source and comes with an invitation to contribute new methods, data sets, and case studies. In the future we envisage contributions in the area of scalability and metrics for assessing specific tasks. Our software is related to PyTorch Geometric Temporal, another extension library to PyTorch Geometric developed for the analysis of temporal networks, and in future these two software packages are likely to grow closer together.

REFERENCES

[1] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A System for Large-Scale Machine Learning. In 12th (I$\Sigma$NIX) Symposium on Operating Systems Design and Implementation ((OSDI) 16). 265–283.
[2] Lada A Adamic and Natalie Glance. 2005. The political blogosphere and the 2004 US election: divided they blog. In Proceedings of the 3rd International Workshop on Link Discovery. Ieee, 97–98.

[3] S. Aghabozorgi et al. 2015. Time-series clustering—A decade review. Information Systems 53 (2015), 16–38.

[4] Stefanos Bennett, Mihai Cucuringu, and Gesine Reinert. 2021. Detection and clustering of lead-lag networks for multivariate time series with an application to financial markets. 7th SIAGDD Workshop on Mining and Learning from Time Series (MiLTS) (2021).

[5] Michael Bertolacci, Edward Cripps, Ori Rosen, John W Lau, and Sally Cripps. 2019. Climate inference on daily rainfall across the Australian continent. 1876–2015. The Annals of Applied Statistics 13, 2 (2019), 683–712.

[6] Aleksandar Bojchevski and Stephan Günnemann. 2017. Deep Gaussian embedding of graphs: Unsupervised inductive learning via ranking. In ICLR Workshop on Representation Learning on Graphs and Manifolds.

[7] Alexandre Bovet and Peter Grindrod. 2020. The Activity of the Far Right on Telegram. https://www.researchgate.net/publication/346968575_The_Activity_of_the_Far_Right_on_Telegram_v21.

[8] Andrew F Bradley. 1997. The use of the area under the ROC curve in the evaluation of machine learning algorithms. Pattern Recognition 30, 7 (1997), 1145–1159.

[9] Yukuo Cen, Zhenyu Hou, Yan Wang, Qibin Chen, Yizhen Luo, Xingcheng Yao, Anhan Zeng, Shiqiang Guo, Peng Zhang, Guohao Dai, et al. 2021. CogDL: An Extensive Toolkit for Deep Learning on Graphs. arXiv preprint arXiv:2103.09599 (2021).

[10] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. arXiv preprint arXiv:1512.01274 (2015).

[11] Kai-Yang Chiang, Joyce Joyoung Whang, and Inderjit S Dhillon. 2012. Scalable clustering of signed networks using balance normalized cut. In Proceedings of the 21st ACM International Conference on Information and Knowledge Management. 615–624.

[12] Gabor Csardi, Tamas Nepusz, et al. 2006. The igraph software package for complex network research. InterJournal, Complex Systems 2006, 5 (2006), 1–16.

[13] Mihai Cucuringu, Peter Davies, Aldo Glielmo, and Hemant Tyagi. 2019. SPONGE: Graph Signal Processing, Graph Neural Network and Graph Learning on Biological Data. A Systematic Review. IEEE Reviews in Biomedical Engineering (2021).

[14] Yu Li, Xin Yuan, Mohsen Radfar, Peter Marendy, Wei Ni, Terence J O’Brien, and Pablo M Casillas-Expinosa. 2021. Graph Signal Processing, Graph Neural Network and Graph Learning on Biological Data. A Systematic Review. IEEE Reviews in Biomedical Engineering (2021).

[15] Yi Ma, Jianye Hao, Yaodong Yang, Han Li, Junqi Jin, and Guangyong Chen. 2019. Spectral-based graph convolutional network for directed graphs. arXiv preprint arXiv:1907.08990 (2019).

[16] Corona, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2021. EvaluNE: A framework for Network embedding via graph attention. In Proceedings of the AAAI Conference on Artificial Intelligence, Vol. 35. 4772–4779.

[17] Yi Ma, Jianye Hao, Yaodong Yang, Han Li, Junqi Jin, and Guangyong Chen. 2019. Spectral-based graph convolutional network for directed graphs. arXiv preprint arXiv:1907.08990 (2019).

[18] Fragkiskos D Malliaros and Michalis Vazirgiannis. 2013. Clustering and community detection in directed networks: A survey. Journal of Machine Learning Research 14 (2013), 1–90.

[19] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. arXiv preprint arXiv:1512.01274 (2015).

[20] Aohan Zeng, Shiguang Guo, Peng Zhang, Guohao Dai, et al. 2021. CogDL: An Extensive Toolkit for Deep Learning on Graphs. arXiv preprint arXiv:2103.09599 (2021).

[21] Alex Verba, Charles P Hobbs, and Chris Leary and Dougal Maclaurin and George Necula and Adam Paszke. 2018. Composable Transformations of Python+NumPy Programs. http://github.com/google/jax.

[22] Diederik Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In Proceedings of the 3rd International Conference on Learning Representations.

[23] Yu Li, Xin Yuan, Mohsen Radfar, Peter Marendy, Wei Ni, Terence J O’Brien, and Pablo M Casillas-Expinosa. 2021. Graph Signal Processing, Graph Neural Network and Graph Learning on Biological Data. A Systematic Review. IEEE Reviews in Biomedical Engineering (2021).

[24] Rafael Albuquerque, Mihai Cucuringu, and Gesine Reinert. 2021. Signed network comparison approaches. In Proceedings of the AAAI Conference on Artificial Intelligence, Vol. 35. 1361–1370.

[25] Lu Li, Yuan Tian, Jiaowei Zhang, and Yi Chi. 2020. Learning signed network embedding via graph attention. In Proceedings of the AAAI Conference on Artificial Intelligence, Vol. 34. 4772–4779.

[26] Junjie Huang, Huawei Shen, Liang Hou, and Xueqi Cheng. 2019. Signed graph clustering of lead-lag networks for multivariate time series with an application to financial markets. IEEE Transactions on Knowledge and Data Engineering 31, 7 (2019), 1378–1388. https://doi.org/10.1109/TKDE.2017.2802925.

[27] Aohan Zeng, Shiguang Guo, Peng Zhang, Guohao Dai, et al. 2021. CogDL: An Extensive Toolkit for Deep Learning on Graphs. arXiv preprint arXiv:2103.09599 (2021).

[28] Juhi Hu, Shengsheng Qian, Quan Fang, Youze Wang, Quan Zhao, Huawen Zhang, and Changsheng Xu. 2021. Efficient Graph Deep Learning in TensorFlow with TF2-GNN. arXiv preprint arXiv:2105.11552 (2021).

[29] Junjie Huang, Huawei Shen, Liang Hou, and Xueqi Cheng. 2019. Signed graph attention networks. In International Conference on Artificial Neural Networks. Springer, 566–577.

[30] Aohan Zeng, Shiguang Guo, Peng Zhang, Guohao Dai, et al. 2021. CogDL: An Extensive Toolkit for Deep Learning on Graphs. arXiv preprint arXiv:2103.09599 (2021).

[31] Erhard Rahm and Giancarlo Pedone. 2005. Conceptual model for binary relations. In Proceedings of the 13th International Conference on Data Engineering. IEEE, 209–220.

[32] Jiyan Zeng, Xiongwei Yu, Xun Ge, and Jingwu Jiang. 2019. Adversarial training for clustering with local consistency. In ICLR Workshop on Optimization for Deep Learning Tasks.

[33] Alex Verba, Charles P Hobbs, and Chris Leary and Dougal Maclaurin and George Necula and Adam Paszke. 2018. Composable Transformations of Python+NumPy Programs. http://github.com/google/jax.

[34] Fragkiskos D Malliaros and Michalis Vazirgiannis. 2013. Clustering and community detection in directed networks: A survey. Journal of Machine Learning Research 14 (2013), 1–90.

[35] Diederik Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In Proceedings of the 3rd International Conference on Learning Representations.

[36] Mihai Cucuringu, Peter Davies, Aldo Glielmo, and Hemant Tyagi. 2019. SPONGE: Graph Signal Processing, Graph Neural Network and Graph Learning on Biological Data. A Systematic Review. IEEE Reviews in Biomedical Engineering (2021).

[37] Lu Li, Yuan Tian, Jiaowei Zhang, and Yi Chi. 2020. Learning signed network embedding via graph attention. In Proceedings of the AAAI Conference on Artificial Intelligence, Vol. 34. 4772–4779.

[38] Junjie Huang, Huawei Shen, Liang Hou, and Xueqi Cheng. 2019. Signed graph clustering of lead-lag networks for multivariate time series with an application to financial markets. IEEE Transactions on Knowledge and Data Engineering 31, 7 (2019), 1378–1388. https://doi.org/10.1109/TKDE.2017.2802925.

[39] Yu Li, Xin Yuan, Mohsen Radfar, Peter Marendy, Wei Ni, Terence J O’Brien, and Pablo M Casillas-Expinosa. 2021. Graph Signal Processing, Graph Neural Network and Graph Learning on Biological Data. A Systematic Review. IEEE Reviews in Biomedical Engineering (2021).

[40] Meng Liu, Youzhi Luo, Limei Wang, Xiaocheng Zhang, Zhiwei Ren, and Shuigeng Zhang. 2021. GNNRank: Learning Global Rankings from Pairwise Comparisons. arXiv preprint arXiv:2202.05287 (2021).
[54] Marc J Perry. 2003. *State-to-state Migration Flows, 1995 to 2000*. US Department of Commerce, Economics and Statistics Administration.

[55] Giulio Rossetti, Letizia Milli, and Rémy Canabet. 2019. CDLIB: a python library to extract, compare and evaluate communities from complex networks. *Applied Network Science* 4, 1 (2019), 1–26.

[56] Benedek Rozemberczki, Carl Allen, and Rik Sarkar. 2019. Multi-scale Attributed Node Embedding. *arXiv preprint arXiv:1909.13882* (2019).

[57] Benedek Rozemberczki, Paul Scherer, Yixuan He, George Panagopoulos, Alexander Riedel, Maria Asteфанoaei, Oliver Kiss, Ferenc Beres, Guzmán López, Nicolas Collignon, et al. 2021. Pytorch geometric temporal: Spatiotemporal signal processing with neural machine learning models. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*. 4564–4573.

[58] Samuel Franklin Sampson. 1968. *A novitiate in a period of change: An experimental and case study of social relationships*. Cornell University, Ithaca, NY 14850, USA.

[59] Venu Satuluri and Srinivasan Parthasarathy. 2011. Symmetrizations for clustering directed graphs. In *Proceedings of the 14th International Conference on Extending Database Technology*. 343–354.

[60] Marina Sokolova and Guy Lapalme. 2009. A systematic analysis of performance measures for classification tasks. *Information Processing & Management* 45, 4 (2009), 427–437.

[61] Christian L Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. 2016. NetworKit: A tool suite for large-scale complex network analysis. *Network Science* 4, 4 (2016), 508–530.

[62] Jiliang Tang, Yi Chang, Charu Aggarwal, and Huan Liu. 2016. A survey of signed network mining in social media. *ACM Computing Surveys (CSUR)* 49, 3 (2016), 1–37.

[63] Veronika Thost and Jie Chen. 2021. Directed Acyclic Graph Neural Networks. In *International Conference on Learning Representations*. https://openreview.net/forum?id=_Ju5YF437WB6

[64] Zekun Tong, Yuxuan Liang, Henghu Ding, Yongxing Dai, Xinke Li, and Changhu Wang. 2021. Directed Graph Contrastive Learning. *Advances in Neural Information Processing Systems* 34 (2021).

[65] Zekun Tong, Yuxuan Liang, Changsheng Sun, Xinke Li, David Rosenblum, and Andrew Lim. 2020. Digraph inception convolutional networks. *Advances in Neural Information Processing System* 33 (2020), 17907–17918.

[66] Zekun Tong, Yuxuan Liang, Changsheng Sun, David S Rosenblum, and Andrew Lim. 2020. Directed graph convolutional network. *arXiv preprint arXiv:2004.13970* (2020).

[67] Vincent A Traag, Ludo Waltman, and Nees Jan Van Eck. 2019. From Louvain to Leiden: guaranteeing well-connected communities. *Scientific Reports* 9, 1 (2019), 1–12.

[68] Cunchao Tu, Yuan Yao, Zhengyuan Zhang, Ganqu Cui, Hao Wang, Changxin Tian, Jie Zhou, and Cheng Yang. 2018. OpenNE: An Open Source Toolkit for Network Embedding. https://github.com/thunlp/OpenNE.

[69] William George Underwood, Andrew Elliott, and Mihai Cucuringu. 2014. Exploting social network structure for person-to-person sentiment analysis. *Transactions on Neural Networks and Learning Systems* 25, 1 (2014), 24–34.

[70] Yizhou Sun, Ryusei Satou, and Kenji Gretz. 2017. A review of Graph Neural Networks for signed and directed graphs. In *Advances in Neural Information Processing Systems* 30 (2017). 4564–4573.

[71] YuXi Wang, Xiaoyu Yang, and Yang Xiang. 2021. SNE: signed network embedding. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 3165–3166.

[72] Shuhan Yuan, Xintao Wu, and Yang Xiang. 2017. SNE: signed network embedding. *IEEE Transactions on Neural Networks and Learning Systems* 32, 1 (2020), 4–24.

[73] Yahoo! finance. 2021. S&P 1500 data. https://finance.yahoo.com/. [Online; accessed 19-January-2021].

[74] Hongxia Yang. 2019. AlGoGraph: A Comprehensive Graph Neural Network Platform. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 3165–3166.

[75] Shuhan Yuan, Xintao Wu, and Yang Xiang. 2017. SNE: signed network embedding. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer, 183–195.

[76] Xitong Zhang, Yixuan He, Nathan Brugnone, Michael Perlmutter, and Matthew Hirn. 2021. Magnet: A neural network for directed graphs. *Advances in Neural Information Processing Systems* 34 (2021).