Applied Network Science

RESEARCH Open Access

Py3plex toolkit for visualization and analysis of multilayer networks

Blaž Škrlj[1,2*], Jan Kralj[2] and Nada Lavrač[1,2]

*Correspondence: blaz.skrlj@ijs.si
[1]Jožef Stefan International Postgraduate School, Jamova 39, 1000 Ljubljana, Slovenia
[2]Jožef Stefan Institute, Jamova 39, 1000 Ljubljana, Slovenia

Abstract

Complex networks are used as means for representing multimodal, real-life systems. With increasing amounts of data that lead to large multilayer networks consisting of different node and edge types, that can also be subject to temporal change, there is an increasing need for versatile visualization and analysis software. This work presents a lightweight Python library, Py3plex, which focuses on the visualization and analysis of multilayer networks. The library implements a set of simple graphical primitives supporting intra- as well as inter-layer visualization. It also supports many common operations on multilayer networks, such as aggregation, slicing, indexing, traversal, and more. The paper also focuses on how node embeddings can be used to speed up contemporary (multilayer) layout computation. The library's functionality is showcased on both real and synthetic networks.

Keywords: Multilayer networks, Network visualization, Complex systems, Network embedding

Introduction

Analysis and visualization of complex networks offers novel opportunities to study intractable systems, such as protein interaction networks, transportation networks or social networks (Boccaletti et al. 2014; Wang et al. 2015; Pavlopoulos et al. 2008). As this vibrant research field offers novel tools at an increasing pace, development of freely available, scalable software resources is becoming a relevant research direction. Despite many existing tools for analysis and visualization of homogeneous networks, i.e. networks with only a single node type and non-annotated edges, tools that consider multilayer networks—also referred to as heterogeneous networks—are an active research area. In multilayer networks, many different types of annotations are taken into account, including relation-labeled edges and multiple node types. Such networks are considered, for example, when multiple layers of biological information (e.g., protein-protein, gene-gene interactions etc.) are available and need to be taken into account when studying diseases. We consider networks as multilayer when they contain at least two types of nodes or at least two types of edges.

This work is inspired by previous studies on multilayer networks formalized by Kiela et al. (2014). The paper first presents Py3plex, a Python library for analysis and visualization of heterogeneous (and homogeneous) networks. The visualization suite simplifies displaying of multilayered networks and network communities as well as network

embeddings (Grover and Leskovec 2016). Py3plex also implements a state-of-the-art procedure for converting heterogeneous networks to homogeneous networks (Kralj et al. 2018), a set of subroutines for partition enrichment (i.e. the process of learning qualitative explanations relating individual partitions, e.g., communities) using expert-curated domain knowledge Škrlj et al. (2018, 2019), and offers intuitive visualization of network-topological properties, such as the community structure.

In this paper, we also demonstrate the performance of embedding based network layout, where state-of-the-art network embedding algorithms—i.e., algorithms which map a given network to a predefined vector space—are used to obtain node coordinates in two dimensions. We show that this method is notably faster on larger networks, and can serve as a relevant improvement of contemporary force-directed layout computation. Finally, we explore how multiplex dynamic social networks can be visualized using the presented visualization technique. Such networks consist of the same set of nodes projected across different contexts. For example, behavior of users can be monitored across the social media they participate in (e.g., Twitter, Facebook etc.). We conclude with a discussion regarding both positive and negative aspects of the presented library along with suggestions for the further work.

This paper significantly extends the work on multilayer network visualization and analysis presented in our previous conference publication (Škrlj et al. 2019). First, the related work section ("Related work" section) has been extended by including tools Pajek (Batagelj and Mrvar 2001) and Tulip (Auber 2004; Auber et al. 2017) that inspired the creation of Py3plex. Next, we extended the description of the library's features ("Key features" section).

We discuss that—apart from simple statistical analysis—the library also offers the functionality to learn from networks using some of the recently introduced machine learning approaches. Further, we explore in more detail the functionality related to community enrichment, as it offers the functionality to explain a given network's partitioning using symbolic learning. Next, we added a section which explores how network embeddings can be used to construct network layout ("Embedding-based network layout" section). In this section, we first discuss relevant machine learning methodology and how it relates to two distinct steps in the layout construction. We next demonstrate how the presented layout compares to efficient Barnes-Hut-based minimization. As a demonstration of novel functionality, we added an example where a multiplex dynamic social network is visualized (see "Experimental evaluation" section). Finally, we extended the discussion and conclusions ("Conclusions and further work" section), where we discuss some of the current limitations, and the existing uses of the library.

Related work

This section presents the state-of-the-art network analysis libraries, relevant to the development of Py3plex. The most common approaches to network analysis can be split into two groups: GUI-based solutions and API-based solutions.

The most used GUI-based solutions include Cytoscape (Shannon 2003), Gephi (Bastian et al. 2009), and Pajek (Batagelj and Mrvar 2001). Cytoscape (Shannon 2003) is one of the largest network analysis projects to date. It supports custom manipulation of the loaded network, and is hence flexible both in terms of network visualization as well as analysis. The Gephi (Bastian et al. 2009) suite offers a similar set of functionalities, but it is known

for better visualization capabilities. Similarly, Pajek (Batagelj and Mrvar 2001) is used to analyse simple graphs, and was shown to scale well to large social networks. These solutions are mostly used in the final step of a network analysis project, where pre-computed node properties are used as part of the input. The tools are constantly updated with novel functionality, offering many graph analysis algorithms out-of-the-box.

The API-based solutions, which provide programmatic access from popular languages (such as Python, R, JavaScript, C++), are preferred when the entire network analysis and visualization is performed in the same environment. The NetworkX library (Hagberg et al. 2008) is prevalent for the Python environment, while the igraph library (Nepusz and Csárdi 2006) is commonly used among the R users. C, and C++ alternatives include SNAP (Leskovec and Sosič 2016), Boost Graph Library (BGL) (The Boost Graph Library 2002), and Tulip (Auber 2004; Auber et al. 2017). Compared to GUI-based analysis, API-based approaches result in a series of high-level function calls, which generate the desired output. Such approaches are commonly used for analysis when either the considered networks are large or the number of networks under consideration is high.

The above approaches focus on homogeneous networks consisting of single node (and edge) types. However, recent advances in multilayer network analysis have proven that the additional information associated with node and edge type provides insights regarding network structure and dynamics. Multilayer networks can be represented as higher order tensors (De Domenico et al. 2013), encoding inter- as well as intra-layer connections of varying intensity. In this paper, we focus on state-of-the-art implementations of this formalism, their functionality, and some of their drawbacks. The currently used libraries for visualization and analysis of multilayer networks include (1) libraries for the Python environment that include Pymnet[1] (Kivelä et al. 2014) and MultinetX[2] (Amato et al. 2017), as well as (2) a library for the R environment Muxviz[3] (De Domenico et al. 2015). Detailed analysis of these approaches is presented in "Comparison of multilayer network libraries" section, where they are compared to the presented Py3plex library.

Py3plex library architecture

This section explains the presented Py3plex library's architecture, followed by the description of individual components and subroutines.

Module organization

A high-level organization of the Py3plex library is shown in Fig. 1. The library includes methods for parsing and converting graphs from and to various formats, while the core library includes three modules:

- **The visualization module** consists of different subroutines used for network visualization of multilayer and single layer networks. Detailed description of the in-house developed visualization is given in "Py3plex multilayer network visualization" section.
- **The wrappers module** As many procedures are given as standalone executables, Py3plex offers a set of wrapper subroutines, useful for calling external state-of-the-art algorithms, for example, highly optimized network embedding routines (Grover and

[1] http://www.mkivela.com/pymnet/
[2] https://github.com/nkoub/multinetx
[3] http://muxviz.net/

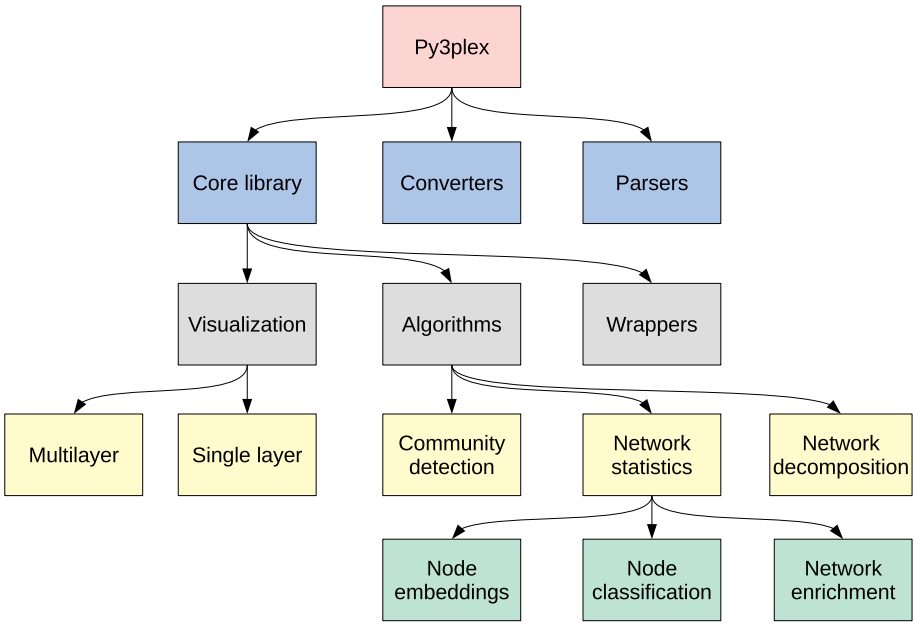


Fig. 1 The Py3plex library architecture. The library is organized in hierarchical manner—a set of core data structures (blue) is used in algorithms for visualization and analysis of multilayer networks

Leskovec 2016) as well as the InfoMap community detection algorithm (Rosvall et al. 2009). The methods offer easy access to multiplex community detection capabilities of Infomap, as well as community computation directly on the supra-adjacency matrix of a given network.

- **The algorithms module** includes implementations of many commonly used algorithms, such as Louvain community detection (Blondel et al. 2008), node ranking, network statistics, the recently introduced network topology enrichment (Škrlj et al. 2018), and network node classification and network embeddings.

All the modules are built in an extensible manner, allowing for new routines to be easily added. As Py3plex is built on top of the NetworkX (Hagberg et al. 2008), network operations can include any of the methods primarily designed for homogeneous networks. This functionality provides flexibility when implementing novel multilayer network analysis algorithms.

Key features

As Py3plex consists of multiple building blocks, we next describe key building blocks comprising the library. The set of data structures underlying all aforementioned modules offers intuitive access to manipulation, creation, and visualization of multilayer networks. Additionally, users of the library can use this core set of functionalities to implement their own algorithms. Other modules comprising Py3plex also build on top of the "multilayer network" object, a data structure we introduce for easier manipulation and consistency throughout the library.

The visualization module includes graphical primitives, needed to construct multi- and single-layered visualizations of considered networks. The module is built so that it accepts the core network structure (described in the previous section) as input and subsequently

outputs a visualization of choice. Detailed formulation of the supported visualizations is given in the following sections.

A substantial portion of Py3plex's functionality is devoted to analysis of multilayer networks. Here, functionality such as network aggregation, centrality computation, and similar, offers straightforward analysis of the network's properties. This part of the library also includes some of the state-of-the-art approaches for network decomposition and enrichments described in the following section.

Comparison of multilayer network libraries

In this section, we compare the functionality of different multilayer network analysis and visualization libraries. Where relevant, we emphasize the novelties Py3plex offers and how they compare to existing alternatives[4]. Each library offers some functionality that is not available in other libraries, hence no library is exhaustive. We evaluate different aspects of functionality, ranging from their computational efficiency to the number of various analysis functions provided. Table 1 presents the comparison of multilayer network analysis and visualization libraries, of which functionality is discussed in this section.

To ensure sufficient speed of execution, Pymnet (Kivelä et al. 2014), Py3plex and MultinetX (Amato et al. 2017) include subroutine implementations in C (via Cython), but also Numpy (Walt et al. 2011) and Scipy (Jones et al. 2001) libraries for optimized linear algebra-based computation. The R-based MuxViz (De Domenico et al. 2015) uses the Octave library for efficient array-based operations.

Table 1 Comparison of multilayer network analysis and visualization libraries

	Py3plex	Pymnet (Kivelä et al. 2014)	MuxViz (De Domenico et al. 2015)	MultinetX (Amato et al. 2017)
Core features				
Programming language	Python 3 and C (via Numpy and Cython)	Python 3 and C (via Numpy)	R	Python 3 and C (via Numpy)
Basic statistics	✓	✓	✓	✓
Visualization of large networks	✓	-	✓	-
Visualization in 3D	-	✓	-	✓
Aggregation/decomposition	✓	✓	✓	-
Random graph generators	✓	✓	✓	✓
Adjacency matrix manipulation	✓	✓	✓	✓
[3pt] Additional features				
[3pt] Node classification	✓	-	-	-
Isomorphisms	✓	✓	-	-
Community detection	✓	-	✓	-
GUI version	-	-	✓	-
Tensor manipulation	✓	✓	✓	✓
Node ranking	✓	✓	✓	✓
Semantic topology enrichment	✓	-	-	-
Temporal networks	-	-	✓	✓
Network embedding	✓	-	-	-

[4]For a comprehensive overview of visualization tools, we refer the reader to the recent survey (McGee et al. 2019).

In terms of visualization, all packages compared, include at least some form of network visualization. Apart from 2D layouts, The MuxViz and Pymnet libraries also support 3D layouts. As discussed in "Py3plex multilayer network visualization" section, Py3plex offers a new approach to multilayer network visualization, whereas MultinetX, for example, is capable of displaying the supra-adjacency matrix (De Domenico et al. 2013) of the network. The multitude of different multilayer network visualization applications indicates that no single type of visualization is optimal for a given task, as each visualization is optimal for a particular range of network size and density.

Apart from MultinetX, all other libraries compared above include different methods for network aggregation and decomposition. A similar compendium of functionality is offered by Pymnet and MuxViz. All packages offer intuitive access to a network's supra-adjacency matrix, and hence provide many opportunities for implementation of computationally efficient multilayer network analysis algorithms. However, Py3plex also leverages full functionality of the recently developed HINMINE (Kralj et al. 2018) methodology for heterogeneous network decomposition and network node classification. As standard aggregation schemes (De Domenico et al. 2015) yield a homogeneous network, for example, by summing over edges in individual layers, HINMINE-based aggregation is based on frequency of relation-annotated directed paths (of length two) between the target node type. Compared to standard aggregation, HINMINE is thus suitable for situations where domain knowledge was used for annotating the network edges. Additionally, Py3plex also includes basic aggregation subroutines, as well as supports classification using Personalized PageRank-based feature vectors is also supported (Kralj et al. 2018). An interested reader can find the details in Appendix A.

All of the libraries listed above include methods for obtaining quick insights into a network's structure. The Pymnet and Py3plex are currently the only libraries supporting different isomorphism algorithms for evaluating network similarity.

In terms of other functionalities, we observe the following. MuxViz also supports GUI-based analysis solutions, as well as API-based ones.Once installed, it can be executed locally within the browser as standalone software.

We observe Pymnet offers one of the most intuitive API interfaces for manipulation of tensor representations of multilayer networks, such as slicing, indexing etc. In comparison, MultinetX and MuxViz offer similar functionality, whereas Py3plex operates on attribute-rich list-based representations of a network and is not necessarily suitable for all multilayer slicing tasks.

Aside from network analysis and visualization capabilities, Py3plex is the only library which also supports various forms of learning from multilayer networks. It provides the methodology for semantic enrichment of complex networks. The main objective in this emerging field is to associate previously known domain knowledge with topological structures of a network. For example, the functional characterization of a network's communities can only be assessed using curated domain knowledge. We refer to the use of such knowledge to understand network-topological properties as network enrichment.

The supported network enrichment, following the Community-Based Semantic Subgroup Discovery (CBSSD) methodology (Škrlj et al. 2018), can be described in two steps:

1 Partition detection. The network is partitioned into separate subnetworks, which commonly represent some form of functional similarity.
2 Enrichment. Individual communities are compared to the remainder of the network by using domain knowledge in the form of ontologies. Should a given ontology term (concept) be over- or under-represented in a considered community, the community shall be considered enriched with respect to this term.

The end result of such analysis are thus community-term pairs, sorted by e.g., term significance. Currently, Py3plex is the only library that supports both rule-based enrichment as well as standard Fisher's exact test-based enrichment, both introduced as part of the CBSSD methodology.

The current implementation of Py3plex contains wrapper routines for widely used network embedding algorithms, such as Node2vec and similar (Grover and Leskovec 2016). Such functionality is currently not offered in any other libraries.

Overall, Py3plex offers novel algorithms for understanding network structure, an intuitive interface for manipulation of multilayer networks as well as network decomposition and aggregation. However, the primary features of the current version of Py3plex are primarily network visualization and spatially efficient network manipulation (linear in terms of nodes and edges). This functionality offers additional state-of-the-art performance, not observed in the other libraries. Below, we first present the proposed network visualization ("Py3plex multilayer network visualization" section), followed by embedding-based node layout computation ("Embedding-based network layout" section), and finally on empirical evaluation of the proposed approaches on a set of artificial random multilayer networks, as well as real world biological and social networks ("Experimental evaluation" section).

Py3plex multilayer network visualization

One of the key contributions of this paper is a novel method for visualization of multilayer networks. In this section, we showcase the proposed visualization method and its implementation. Next, we evaluate the method's performance on a series of random networks, where we subsequently compare the results with Pymnet's network visualization capabilities.

Visualization methodology and implementation

The proposed set of visualization techniques aims to address the issue of visualizing multilayer networks. The implementation of the presented layout algorithm operates in three main steps described below.

Intra-layer layout computation. First, subnetworks consisting of same-typed nodes and edges between them are considered. For each node type, we compute a fast, force-directed layout on the single-type subnetwork. This results in (x, y) coordinate pairs for individual nodes that are scaled so that both coordinates are between 0 and 1. Individual, single-layer networks are derived from the existing NetworkX graph library (Hagberg et al. 2008), consequently offering full NetworkX functionality. Additionally, Py3plex provides the means to customize the majority of network properties, including edge shapes and colors, individual layer layouts, node sizes and colors, and overall network organization. Computationally expensive operations are vectorized using the Numpy numeric library (Walt et al. 2011). The Force Atlas 2

algorithm (based on Barnes-Hut n-body minimization), which was transpiled from Python to C, is used as the default option during visualization (Jacomy et al. 2014). Throughout the work, we refer to such layouts as spring layout plots.

Multilayer network drawing. In the second step, nodes are drawn along a diagonal line based on their type (i.e. layer they belong to). This is achieved by calculating the actual coordinates of each point by separating each consecutive layer from the preceding layer by exactly a single coordinate unit on both axes. The result is that, for a node in the i-th layer, where we calculated coordinates (x, y) in step 1, we now update the coordinates to $(x + i, y + i)$. As nodes in different layers are now separated by at least one coordinate unit, each subnetwork corresponding to different node type is drawn in a separate region with no overlap between different layers. Each layer includes a network with its own local organization, independent of the inter-layer connections.

Inter-layer edge drawing. The final step involves drawing of inter-layer edges, which are represented as arces connecting different nodes. One of the main problems we faced during the implementation of this step was edge positioning and parameterization, as there are many different ways of drawing an arc between two nodes. We considered three possible scenarios in terms of inter-layer edge drawing:

1 The edges are only on the upper part of the layer diagonal;
2 The edges are only on the bottom part of the layer diagonal;
3 The edges are on both sides of the diagonal projection.

An inter-layer edge between nodes n_1 and n_2 of the network (represented by points (x_1, y_1) and (x_2, y_2), respectively) is drawn as follows. First, an artificially introduced point (x_3, y_3) is calculated by taking the midpoint between (x_1, y_1) and (x_2, y_2) and scaling its y coordinate by a factor of τ (a parameter of the method). Scaling the y coordinate by a fixed amount (τ) instead of shifting it ensures that each edge is, in effect, shifted up by a different value, allowing all edges to be visible in the final image. Changing the value of parameter τ offers simple manipulation of inter-layer edges as follows:

1 when $\tau > 1$, the arc will be located above the diagonal;
2 when $\tau = 1$, the arc will be a line between the two nodes (a third point on $f(x)$);
3 when $\tau < 1$, the arc will be located below the diagonal.

Once n_1, n_2 and n_3 are obtained, points n_1 and n_2 are connected by a parabolic arc that passes through all three points. Even though the current implementation constructs inter-layer edges using interpolation over three points, Py3plex supports adding an arbitrary number of intermediary points, in which case cubic arc interpolation is used to produce the line between n_1 and n_2. In our experiments, however, we show that even a single intermediary point allows for clear visualization.

We further propose a heuristic for automatically determining an arc's position, i.e. whether an arc should be located above or below the diagonal. We achieve this as follows. Given n_1 and n_2 as defined in the previous paragraph, the arc is located above the diagonal if and only if $int(n_3) > n_3$, where $int(n_3)$ represents the integer representation of the coordinates of n_3. Integer-based rounding works, as layers are separated by exactly a single coordinate unit. All operations for arc computation, scaling and transformation are vectorized for better performance. Should the network

appear incomprehensible, natural logarithms are applied to node representations—filled circles based on individual node degrees—which simplifies visualization of denser networks.

Py3plex can easily visualize more than ten layers with tens of thousands of nodes. Compared to existing solutions, diagonal projection of multiple layers enables visualization using standard layout algorithms, with additional specification of inter-layer edges. An example visualization using the presented Py3plex library is shown in Fig. 2, with an alternative visualization using a single layer force-directed layout of the whole network is included for comparison.

Strengths and potential drawbacks

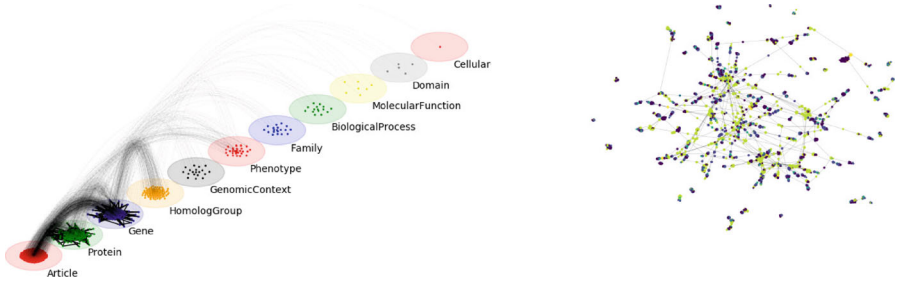
In this section, we discuss the strengths and weaknesses of the proposed visualization. We begin by describing the strengths and continue with the discussion of the cognitive load and other potential drawbacks.

Strengths

Many aspects of the visualization presented in this paper can be customized to emphasize either the node or layer properties. For example, in Fig. 3 we colored differently the inter-layer edges corresponding to specific relations. Additionally, such edges can be plotted either on the upper or the lower side of the diagonal containing networks, making it possible for the user to emphasize only the selected inter-layer edges. The height, types of lines, colors and transparency can be fine-tuned to the user's preferences. The colors of intra-layer edges and nodes can also be customized—for example, special nodes or sets of nodes can be colored to emphasize the intra-layer structure.

Potential drawbacks

In this section, we also discuss some of the drawbacks the presented approach introduces, especially when considering larger networks. One of the main problems with such complex visualizations is the amount of overlapping edges, which was shown to be problematic for the user experience by Purchase (1997). We split the following discussion into two main parts: intra-layer overdrawing and inter-layer overdrawing.



(a) Multilayer diagonal projection (b) Single layer spring layout plot

Fig. 2 Comparison between a multilayer visualization and single layer (spring layout) layout supported in Py3plex. The presented diagonal projection is shown in subfigure (**a**), and the spring layout plot in (**b**). The inter-layer connectivity is more clearly expressed in (**a**) than in (**b**). Further, organization between the nodes of the same type can not be observed in subfigure (**b**), as the layout algorithm considers all the connections. Py3plex supports both visualization styles

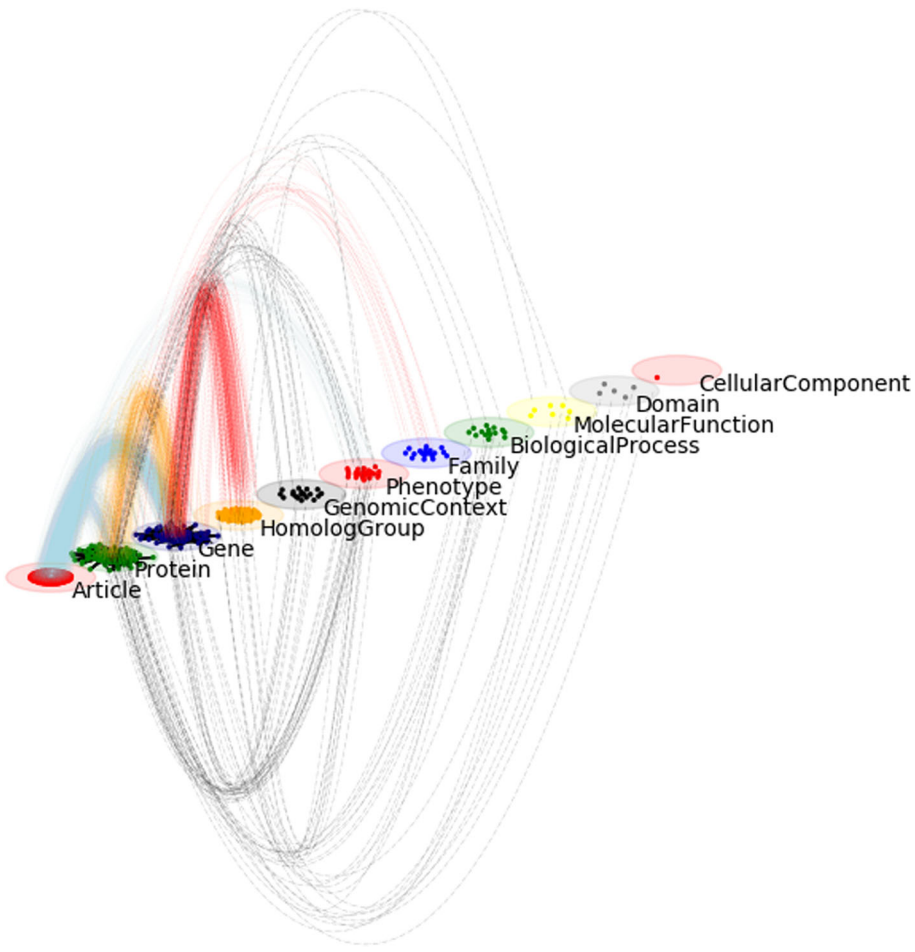


Fig. 3 A customized Py3plex visualization. Here, the blue inter-layer edges correspond to *refers to* relation, orange ones to *codes for* and red to *belongs to* relation

Intra-layer edge overdrawing. Non-planar, real-world networks are impossible to draw without some edge overlap. The problem with drawing networks within layers is thus inherent to all other libraries which visualize such network. Py3plex tackles this issue by enhancing transparency of edges based on density, edge thresholding, as well as controlling edge widths. Further, node sizes also take up substantial amounts of layout space, thus need to be adapted accordingly. Even though the space in which individual intra-layer networks are drawn is smaller than the whole canvas, the Force Atlas 2 algorithm which is used for intra-layer layout computation, disperses the nodes based on their connectivity patterns. This way, it partially separates the densely connected clusters (some arc overlap is reduced this way). To preserve topological properties of intra-layer networks, some overlap (e.g., within functional clusters) is inevitable.

Inter-layer edge overdrawing. In our visualization, the majority of the inter-layer overlaps are noticed at the upper-most (or bottom-most) parts of the parabolic arcs. Techniques for emphasizing edges (e.g., transparency-based filtering etc.) that can be adopted to further emphasize individual edges are discussed in "Strengths" section. Additionally, as the presented inter-layer edges span between

layers on the upper and the lower part of the main diagonal, some of the edge overlap can be reduced by redistributing the edges accordingly across the empty regions of the canvas. Such positioning can be automaticaly determined by Py3plex. Next, heights of individual arcs can be manually configured, enabling definition of custom, less overlapping groups of arcs. While presented solutions do not entirely solve the problem, we believe that for non-planar graphs (especially in 2D), if the user knows what aspect to emphasize, the proposed solution can provide sensible visualizations. While separating the layers may incur more overall edge overdrawing, we believe that this cost is outweighed by the benefits (i.e. added visual clarity offered by visually separated layers) of our visualization.

Finally, we discuss how intra- as well as inter-layer edges contribute to understanding of the plots. Should the number of inter-layer edges increase in layers that are very close together, such setting is prone to cognitive overload and can be addressed by specifying a different layer order. Very sparse networks with many layers are also harder to visualize using the presented visualization, as with many layers, information regarding connectivity can become harder to comprehend—the inter-layer edges can span across larger regions of canvas and are harder to follow.

Cognitive load

In this section we discuss the potential implications of the presented methodology with respect to cognitive load, as this aspect of visualization can be critical in determining the usability of a given visualization method. This section follows guidelines from Huang et al. (2009), who investigated how complex networks remain understandable to a non-expert human observer. Via a variety of cognitive tasks (such as triangle counting), Huang et al. showed that networks with only tens of nodes and hundreds of edges can already pose a problem when it comes to their interpretation and understanding. While the work by Huang et al. is not focused on multilayer networks, visualization properties they recognize as relevant are also applicable when visualizing multilayer networks.

Huang et al. (2009) identify several key factors of cognitive load presented by a given network visualization. The factors that are most relevant for the following discussion regarding cognitive load of Py3plex plots are the following:

Domain complexity. Not all domains are equally complex. Huang et al. point out that visualizations of biological networks should differ from the ones used for displaying social networks. Py3plex is adapted in line with these findings. The layer-level diagonal visualization introduced in this work offers intuitive segmentation of e.g., biological information (e.g., DNA, RNA, protein etc.); however, such levels are not necessarily present in social networks. In order to address this issue, Py3plex offers functionality to aggregate layers, which can be adapted to specific use-cases.

Data complexity. This aspect is closely related to the studied domain. We observed for example, that biological networks contain more node and edge types, than the social networks, requiring different visualization strategies. The internal data structure used for visualization and manipulation is a heterogeneous information network with (optional) attributes assigned to nodes. This structure was expressive enough for the examples shown in this work, consisting of multiple node and edge types. It could be further adapted to e.g., hypergraphs, should the need arise.

Visual complexity. Visualizations can have varying degrees of complexity. Here, aspects such as edge or node overlaps and the number of different elements visualized need to be considered. The complexity of the visualizations obtained using Py3plex can be high, as it displays multiple layers along with inter- and intra-layer edges. We refer the reader to "Potential drawbacks" section for detailed discussion of the visualization aspects which influence the final output the most.

Interactivity

Because one of the output options supported by Py3plex is a Matplotlib canvas (Hunter 2007), the resulting visualizations can easily be:

- Zoomed-into. A square region of the visualization is selected and zoomed-into. This functionality offers e.g., a way to emphasize only certain layers of interest.
- Stretched. The interactive viewer offers simple functionality for adapting the shape of the resulting visualization, offering fine-tuning with respect to e.g., overlapping text.
- Animated. Matplotlib offers animation functionality, making possible the construction of e.g., dynamic visualizations, consisting of multiple (e.g., time-dependent) frames. An example of such an animation is available online in .gif format[5].

Embedding-based network layout

Visualization of large networks commonly results in long computation times and incomprehensible layouts. Recent advancements in the field of machine learning on graphs can be leveraged to facilitate the process of network visualization. Even though embedding-based data visualization is becoming commonplace in contemporary machine learning, such techniques span back to Harel and Koren (2002), who initially investigated how network embeddings can be used for visualization. They use principal component analysis (PCA) as the main embedding mechanism. Even though PCA can offer valuable insights when projecting the data into orthogonal space of lower dimension, it does not necessarily maintain all network-topological properties which represent key parts of the considered network. The initially considered network embedding (PCA) does not take into account higher-order node neighborhoods, missing out on e.g., densely connected parts of networks that can only be accessible when considering longer random walks.

This section is structured as follows. First, we describe the role of network embedding algorithms in a standard machine learning setting. Next, we show how methods for non-linear dimensionality reduction can be combined with scalable node embeddings for network layout construction. Finally, we present a simple benchmark of the presented layout algorithm compared with a generic, force-directed layout.

Network embedding

Recent advancements in learning from complex networks commonly consist of two main steps: network embedding and learning. Recent approaches for embedding construction include DeepWalk (Perozzi et al. 2014), Node2vec (Grover and Leskovec 2016), Struc2vec (Ribeiro et al. 2017), and similar approaches, all of which attempt to capture node information and encode it in the form of d-dimensional vectors. In this work we are interested

[5] https://github.com/SkBlaz/Py3plex/raw/master/example_images/animation.gif

in the embedding phase of the these algorithms, as well as the use of resulting node embeddings as the first step in the presented layout calculation algorithm. The feature matrix (table) can be used with less additional preprocessing compared to sophisticated relational nature of a graph. The presented visualization approach first constructs such an embedding, and subsequently projects it to a two-dimensional vector space; this space of (x, y) pairs represents initial node coordinates. Schematic representation of this idea is shown in Fig. 4.

We next describe some of the key steps of node2vec, as recently given in (Kralj et al. 2019). We believe understanding of how node2vec operates shall offer the reader intuition as to why use the selected embedding method as a building block of the proposed visualization.

Network embedding using node2vec

A recently developed approach to vectorizing network nodes is the node2vec algorithm (Grover and Leskovec 2016), which uses the random walks to calculate features that express similarities between pairs of nodes.

The node2vec algorithm takes as input a network of n nodes, represented as a graph $G = (V, E)$ where V is the set of network nodes and E is the set of connections, or edges, in the network. The algorithm returns a matrix $f \in \mathbb{R}^{|V| \times d}$ with a pre-defined number of columns d. Matrix f is interpreted as a collection of d-dimensional feature vectors with the i-th row of the matrix corresponding to the feature vector of the i-th node in the network. We write $f(u)$ to mean the row of matrix f, corresponding to node u. The goal of the algorithm is to construct feature vectors $f(u)$ in such a way that the feature vectors of all nodes that share a certain neighborhood will be similar. Matrix f is calculated as follows:

$$\mathcal{E}(G) = \underset{f \in \mathbb{R}^{|V| \times d}}{\arg\operatorname{Max}} \sum_{u \in V} \left(-\log \left(\sum_{v \in V} e^{f(u) \cdot f(v)} \right) + \sum_{n_i \in N_S(u)} f(n_i) \cdot f(u) \right) \tag{1}$$

where $N(u)$ denotes the network neighborhood of node u given a *sampling strategy*, and $\mathcal{E}$ the embedding constructor (node2vec in this case). In the above expression the inner sum calculates the similarities between a node and all nodes in its neighborhood. This

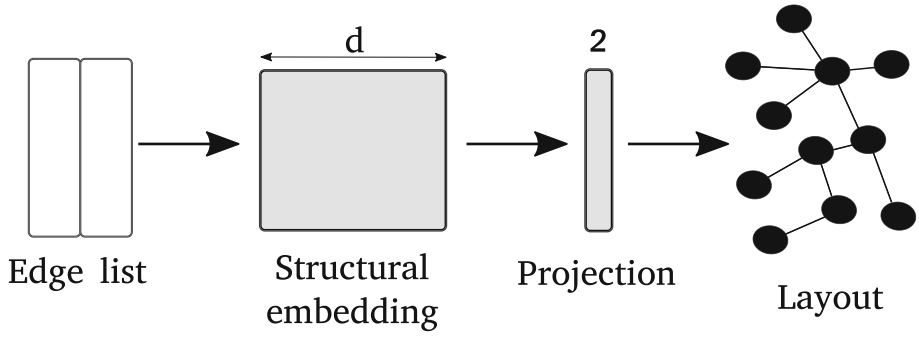


Fig. 4 Schematic representation of network embedding based layout proposed in this work. Input network is first projected to a d-dimensional vector space, where structural properties of individual nodes are captured. Next, the obtained embedding is projected to two dimensions. The two dimensional projections serve as initial points for force-directed layout computation (Layout)

sum is large if the feature vectors of nodes in the same neighborhood are collinear, however it also increases if feature vectors of nodes have a large norm. The first value of each summand decreases when the norms of feature vectors increase, thereby penalizing collections of feature vectors with large norms.

Expression (1) has a probabilistic interpretation which models a process of randomly selecting nodes from the network. The probability $P(n|u)$ of node n following node u in the selection process is proportional to $e^{f(n)\cdot f(u)}$. Assuming that selecting one node is independent from selecting any other node, we can calculate the probability of selecting all nodes from a given set A as $P(A|u) = \prod_{n\in A} P(n|u)$, and Eq. 1 can then be rewritten as follows:

$$\mathcal{E}(G) = \underset{f\in\mathbb{R}^{|V|\times d}}{\arg\operatorname{Max}} \sum_{u\in V} \log\left(P(N_S(u)|f(u))\right). \tag{2}$$

Term $N_S(u)$ in Eqs. (1) and (2) denotes the neighborhood of u given a sampling strategy S and is calculated by simulating a random walker traversing the network starting at node u. The transition probabilities for traversing from node n_1 to node n_2 depends on the node n_0 the walker visited before node n_1, making the process of traversing the network a second order random walk. The unnormalized transition probabilities are set using two parameters, p and q, and are equal to:

$$P(n_2|\text{previous step moved from node } n_0 \text{ to } n_1) = \begin{cases} \frac{1}{p} & \text{if } n_2 = n_0 \\ 1 & \text{if } n_2 \text{ can be reached from } n_1 \\ \frac{1}{q} & \text{otherwise} \end{cases}.$$

Parameters p and q are referred to as the *return* parameter and the *in-out* parameter, respectively. A low value of the return parameter p means that the random walker is more likely to backtrack its steps, meaning the random walk will be closer to a breadth first search. On the other hand, a low value of the parameter q encourages the walker to move away from the starting node and the random walk resembles a depth first search of the network. To calculate the maximizing vector f, a set of random walks of limited size is simulated starting from each node in the network to generate several samples of the set $N_S(u)$.

The function maximizing expression (1) is calculated using stochastic gradient descent. The value of (1) is estimated at each generated sampling of the neighborhoods $N_S(u)$ for all nodes in the network to discover the vector f that maximizes the expression for the simulated neighborhood set.

Extensions to multilayer networks

In this section, we discuss how the node2vec embedding algorithm relates to the considered multilayer network visualization. The original implementation of node2vec operates only on homogeneous, weighted networks. As such, we primarily use it to obtain intralayer layout, which is also the computationally more expensive part of the layout computation. However, as Py3plex can easily return the supra-adjacency matrix, node2vec could also be applied to such matrix directly in order to obtain global node representations. The considered node2vec was also recently extended to multilayer tissue networks indicating such extensions are possible (Zitnik and Leskovec 2017). We test a similar idea with dynamic networks in "Experiment one: benchmark of layout computation time" section.

Reducing embedding dimensionality using t-SNE

Note that even though the nodes of a network can be embedded in 2 dimensional space directly, the obtained representations are normally not representative of the network's structure. If $\mathcal{E}(G)$ represents the network embedding function, we introduce an additional operation, $\mathcal{P}(\mathcal{E}(G))$, i.e. a projection of the obtained network embedding (see previous section) to a 2-dimensional, real-valued vector space. Embedding-based graph drawing was previously proven to scale to very large networks (Hachul and Jünger 2006), thus we believe similar ideas implemented with more recent approaches could offer similarly good performance on large multilayer networks with many layers.

In the experiments shown in the following sections, we use t-SNE projections (Maaten and Hinton 2008) for obtaining the final set of node coordinates. The t-SNE algorithm taks as input a set of high dimensional vectors and projects them to a low-dimensional vector space while maintaining (as much as possible) the similarities between the vectors. For use in data visualization, the low-dimensional space has dimension 2 or 3. The algorithm works in two steps.

1 Similarities between pairs of input vectors (in our case, the d-dimensional node embeddings) are calculated.
2 Low-dimensional vectors are calculated such that the vector-pair-similarities of the low dimensional vectors re-create the original similarities as closely as possible.

Similarities between input vectors $\{x_1, \ldots, x_n\}$ are modeled as follows. Let x_i and x_j denote two points in the input d-dimensional embedding. The similarity between data point x_j and x_i is modeled as the probability that x_i would pick x_j as its neighbor if neighbors were picked in proportion to their probability density under a Gaussian centered at x_i and is calculated as follows:

$$p_{ij} = \frac{e^{\left(-||x_i - x_j||^2/2\sigma_i^2\right)}}{\sum_{g \neq h} e^{\left(-||x_g - x_h||^2/2\sigma_i^2\right)}},$$

where σ_i is the variance of the Gaussian, centered on data point x_i. In t-SNE, σ_i is determined so that the *perplexity* of the Gaussian equals a fixed value, specified by the user. The desired perplexity parameter can be interpreted as the number of neighbors of each data point. From $p_{i|j}$, the joint probability p_{ij} is calculated as $p_{ij} = \frac{p_{j|i} + p_{i|j}}{2n}$.

In finding the low-dimensional representation $\{y_1, \ldots, y_n\}$, t-SNE models similarities between pairs of representation vectors using the Cauchy distribution, calculated as follows:

$$q_{ij} = \frac{\left(1 + ||y_i - y_j||^2\right)^{-1}}{\sum_{g \neq h} \left(1 + ||y_g - y_h||^2\right)^{-1}}.$$

Using this model, t-SNE uses stochastic gradient descend to minimize the Kullback-Leibler divergence (Kullback and Leibler 1951) between the original probabilities p_{ij} and the new probabilities q_{ij}. We refer the interested reader to the original paper (Maaten and Hinton 2008) for technical details of this optimization.

Final formulation

Embedding $\mathcal{E}$, described in "Network embedding using node2vec" section and projection $\mathcal{P}$, described in "Reducing embedding dimensionality using t-SNE" section, represent

two fundamentally different mappings. First, $\mathcal{E}$ attempts to capture a given node's neighbourhood information, whilst $\mathcal{P}$ attempts to construct a low-dimensional embedding by preserving the input's dimension (distances between input feature vectors).

The mappings $\mathcal{E}$ and $\mathcal{P}$ are applied sequentially to obtain coordinate tuples $t_{xy} \in \mathbb{R}^2$, where t_{xy} can already serve as the coordinates for plotting individual nodes, or can be used as the initialization of the force-directed layout. The presented algorithm can thus be described in three simple steps:

1 Network embedding into d-dimensional vector space.
2 Projection of d-dimensional embeddings into two dimensions.
3 (Optional) few iterations of distance minimization.

For the experimental evaluation discussed in the next section we used node2vec for the network embedding part, and t-SNE for projections. Note that the idea discussed in this section is both embedding, as well as projection-agnostic—arbitrary embedding algorithm's output can be projected to two dimensions using an arbitrary projection method. We recognize as relevant related work the body of literature focusing on node embedding learning, summarized in (Goyal and Ferrara 2018), a survey in which node2vec proved to be one of the best performing algorithms. Similarly, the recently introduced Embedding Projector (Smilkov et al. 2016) offers visualization of embeddings projected via PCA or other non-linear projections.

Experimental evaluation

In the following sections, we discuss the performance of Py3plex with respect to visualization, as well as analysis tasks. We begin by comparing the proposed embedding-based layout to some of the contemporary layout algorithms. Next, we demonstrate the scalability of the library and conclude with an analysis of a dynamic multiplex social network.

Experiment one: benchmark of layout computation time

We next present a simple benchmark, where we tested the speed of the presented method in comparison with the Force Atlas 2 algorithm (FA2) (which uses Barnes-Hut approximation for the n-body problem for faster minimization). We used the FA2 algorithm as it is widely used in software such as Gephi (Bastian et al. 2009), representing a relevant baseline for this task.

We compared the computation time of the two algorithms on a real-life protein-protein interaction network described below. The IntAct protein-protein interaction network is currently one of the largest resources for mining the human proteome. To perform the experiments, we first downloaded the current version of protein-protein interaction network from the IntAct database (Orchard et al. 2013), which at the time of writing consists of more than 350,000 nodes and approximately 3.8 million edges. In IntAct, the nodes represent individual proteins, and the (undirected) edges represent their interactions. The edges are weighted, where the edge weights correspond to experimental reliability of the interactions between the corresponding proteins, and take values between 0 and 1. This data base consists of protein-protein interaction pairs, scored with a real value representing the confidence of a given interaction. For comparing layouts, we used all edges with score of at least 0.2, yielding a network with more than 100,000 nodes

and 400,000 edges. We computed layouts for each network five times, and averaged the computation times.

The obtained benchmark times along with computed layouts are summarized in Table 2. It can be observed that the embedding based layout looks notably different to any of the force-directed ones. After many rounds of minimization, the embedding based layout starts to resemble the force-directed one. We believe this experiment demonstrates the power of using network embeddings for qualitative analysis. We additionally discuss the obtained results in "Conclusions and further work" section. We continue with a benchmark study, where we compare the visualization time of Py3plex, compared with Pymnet, an alternative Python based library for visualization of such networks.

Experiment two: overall performance

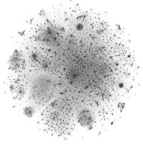
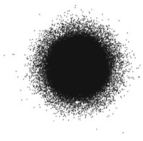
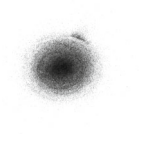
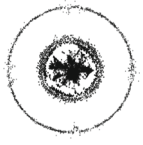
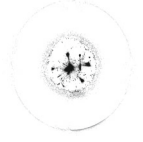
In this section we first present the result of comparing visualization times of Py3plex and Pymnet libraries. Next we discuss a practical case study where we visualize a multiplex dynamic social network. Multiplex networks are comprised of the same set of nodes projected across layers. Here, no physical inter-layer edges are commonly present, as they correspond to *is a* relation.

We present the results for the times needed to obtain a visualization of networks of different complexities; here, we compare Py3plex to Pymnet library. Even though Pymnet does not support the diagonal projection and Py3plex does not support the default 3D linear projections, both methods are useful for visualization of different aspects of multilayer networks.

The main aim of this section is to demonstrate that Py3plex offers better support for visualization of larger networks.

The experimental setting for this task was designed as follows. Random multilayer Erdős-Rényi (ER) networks were generated using the Pymnet (Kivelä et al. 2014) library. We used this network model purely for its simplicity and control over the node and edge space. While ER networks do not exhibit such real-world properties as, for example, Stochastic Block Models (Holland et al. 1983), we believe the larger ER networks considered exhibit enough complexity for comparisons to be meaningful for the considered

Table 2 Comparison of force directed layout with the presented embedding based layout

Iterations	Time (BH)	Visualization (BH)	Time (Embedding)	Visualization (Embedding)
0	1min		23 min	
10	24min		61 min	
100	426min		480 min	

The BH denotes the Barnes-Hut-based Force Atlas2 layout computation. Note that apart from force minimization of the non-embedded network we also present results of minimizing a network's coordinates, initialized using embedding projections

comparisons — in this section, we are only interested in comparing the computation time needed to visualize the networks. As the computation and visualization times are mostly dependent on the numbers of nodes and edges, if Py3plex outperforms Pymnet on these networks, we also expect it to outperform Pymnet on other networks of comparable size.

Such random networks are parameterized using parameter N corresponding to the total number of nodes, parameter L corresponding to the number of layers, and parameter p corresponding to the re-wiring probability. We generated the networks in the following parameter ranges:

$p \in \{0.05, 0.1, 0.2, 0.3\}$,
$N \in \{5, 10, 20, 50, 80, 100\}$,
$L \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$.

Once generated, the time needed for visualization was recorded. We compared visualization times of Pymnet and Py3plex, as the remaining Python-based alternative, MultinetX, does not support drawing of coupled edges.

We show the final results of our experiment in the form of box plots, where $|N|$ or $|E|$ is plotted on the x-axis and the time needed is plotted on the y-axis (Figs. 5 and 6). Networks with more than 100 nodes were not considered for this benchmark, as it took Pymnet more than two hours for visualization. Nonetheless, Py3plex was able to visualize a network with $|N| = 4{,}000$ and $|E| = 18{,}600$ under two hours, even though the obtained network is not necessarily useful for visualization purposes. The machine used for benchmark testing was an off-the-shelf Lenovo y510p laptop.

Visualization of dynamic multiplex networks

Real-world multilayer or multiplex networks are commonly subject to either edge or node dynamics; for example, friendships form over time, or biological phenotypes emerge and disappear (Secrier et al. 2012). Here, the number of e.g., edges can be subject to notable

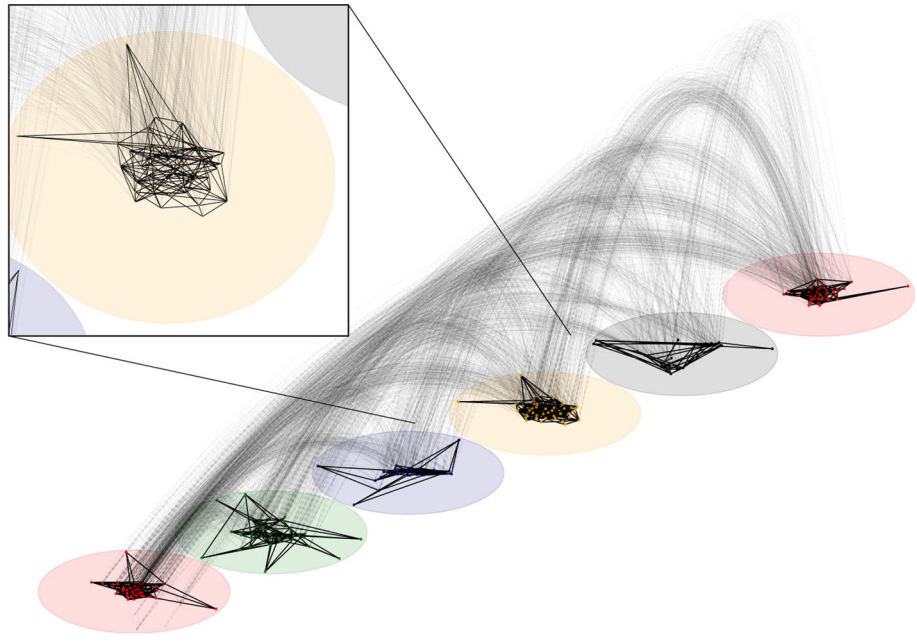
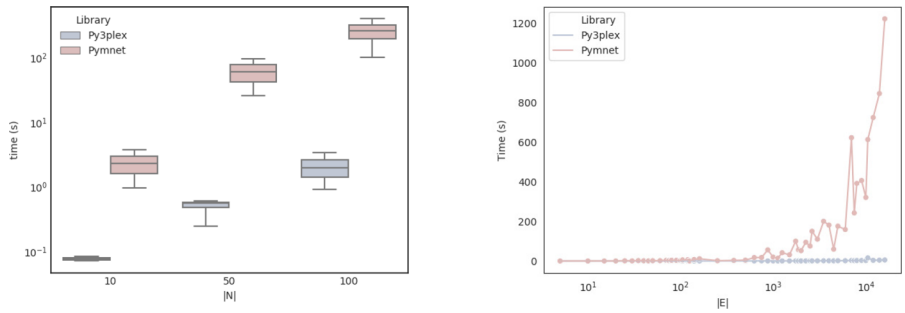


Fig. 5 An example benchmark network used for computation time comparison



(a) Visualization time with respect to the number of nodes.

(b) Visualization time with respect to the number of edges.

Fig. 6 Time consumed to produced the visualization. Time with respect to the number of nodes is shown in (**a**), and time with respect to the number of edges in (**b**)

change, hence the temporal component of a given network can not be neglected. In this section we showcase some of Py3plex's rather simple extensions to dynamic networks. A natural representation for a dynamic network is some form of video output, where individual e.g., time windows or slices are used as frames. We present Py3plex's functionality for production of such frames.

We consider a dynamic multiplex social network initially introduced in Omodei et al. (2015). Here, the same set of users is observed in terms of retweets, mentions and comments. We represented the network by considering each *type of interaction* as a separate layer, as the users remain the same. The network consists of 392,000 users, we consider more than 100,000 time points, each representing an event (edge) between the users. The edges represent either retweets, mentions or comments.

Visualization preparation

As the users in separate layers are the same, we do not plot any inter-layer edges, as in the presented visualization they do not contain any added value and only result in cognitive overload. Next, we compute layouts for individual nodes as follows. We first create a simpler, homogeneous network consisting of all users. There, we compute the layout for each node. We use the same layout for each layer, hence, for example, a node's position in e.g., the *retweet* layer directly corresponds to the one in e.g., *mentions* layer.

We take into account the network dynamics (temporal links) by considering time slices of size 12,000. Although there exist more intricate methods for taking into account the temporal information, we believe the presented visualization serves the demonstration purpose, as it can easily be extended to e.g., a moving window scenario. For clarity, we do not display isolated nodes.

Finally, we fix the order of layers, and plot each frame (slice) as a block in a grid of 4×3 images, showing the evolution of the network. Note that the obtained frames can easily be used to construct a video, should the need arise. Apart from the grid of networks we also attach simple line plots, which show the overall layer activity with respect to a given time slice. This simpler representation of temporal evolution is used to assess whether the presented visualization captured any distinct events in the evolution of the considered network.

Visualization result

The resulting visualization is shown in Fig. 7. The considered time granularity (1000 points) yields re-tweet and mention networks (red and green) with constant activity. The response network (blue) is subject to the highest variability in terms of link quantity. At e.g., slices 3, 10 and 22 the network is subject to high activity — we believe this indicates the spread of viral news. Other slices, such as for example 8 and 9 indicate there are also periods, when only a few mentions appear.

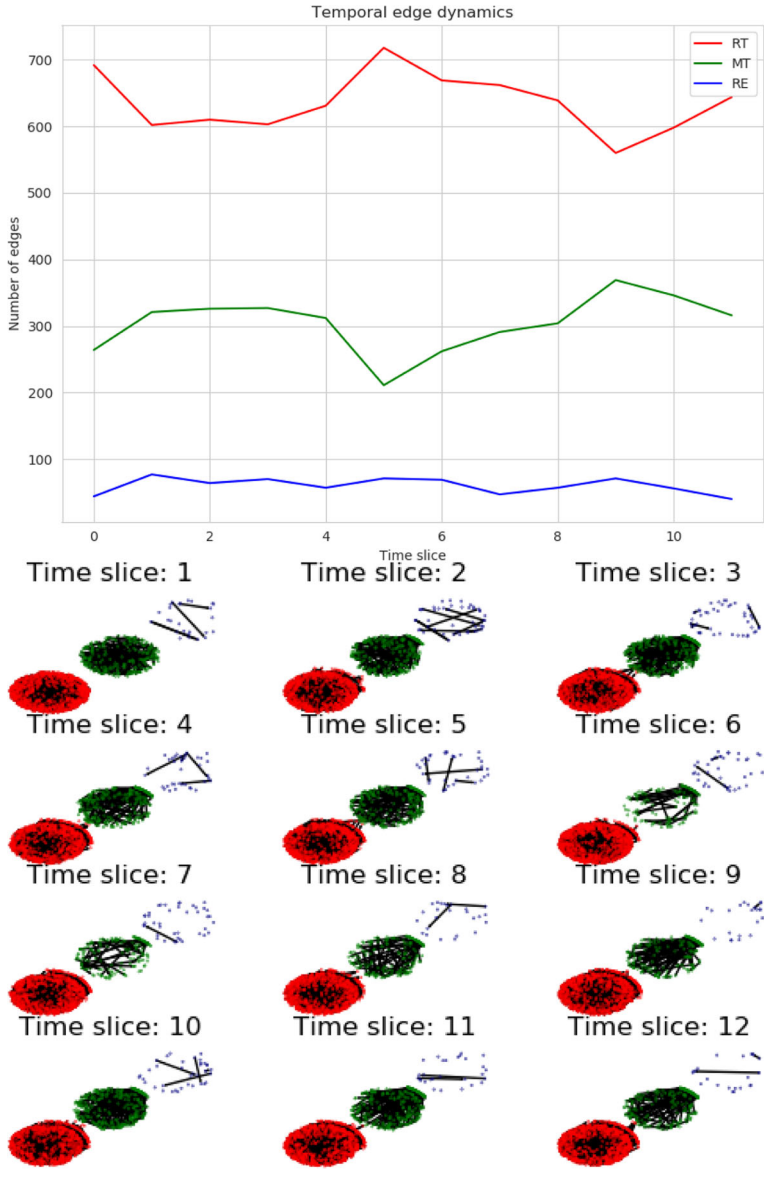


Fig. 7 Multiplex network structure at 12 different time slices. Here, the red (first) layer corresponds to re-tweets, green (second) layer to mentions, and the third (blue) to responses in the considered Twitter social network (Omodei et al. 2015). Note that for demonstration purposes, we visualized only first 12 slices of 1,000 time points. Notable changes in response dynamics can be observed (blue networks and line)—spikes in activity can be seen well in network time slices. Such changes of network density can be studied to better understand complex social dynamics

Conclusions and further work

We have developed and implemented Py3plex, a network analysis and visualization library focused on multilayer networks. In addition to most of the functionality offered by other state-of-the-art libraries, Py3plex introduces a novel visualization, suitable for larger multilayer networks, as to our knowledge, there currently do not exist any methods that can visualize networks composed of thousands of nodes along with their intra- as well as inter-layer organization. Py3plex is suitable for the development of algorithms, as it offers fast lookup and indexing routines, which scale to networks consisting of multiple layers with hundreds of thousands of edges.

One of the new concepts introduced in this paper is a novel multilayer network visualization layout. Py3plex plots intra-layer connections separately to the inter-layer ones, yielding more comprehensive visualizations. We belive the presented diagonal projection could also be extended into an additional dimension (orthogonal), hence offering more efficient space consumption. The presented library currently only includes primitive tools for *animating* multilayer networks—here, the temporal component is split into small slices, which are animated (network snapshots are merged into a single animation). We believe this aspect of Py3plex could be further improved, as many real-world multilayer networks also have a dynamic component.

Another novelty of this work is embedding based visualization. We believe the experiments conducted serve as a proof of concept, yet we believe many other embedding algorithms could also be used to improve the layout's quality. Furthermore, the recently introduced UMAP algorithm (McInnes et al. 2018) could also be used instead of t-SNE, making the layout computation even more scalable. We believe this embedding could be especially useful in situations where larger networks with distinct topological structure are considered. Even though we demonstrated the embedding based layout on a biological network we believe it could serve as a viable visualization alternative to visualize other types of networks, such as for example the transportation vnetworks.

In this work, we compared the visualization capabilities of Py3plex to these of the Pymnet library. Even though we demonstrated that Py3plex layout scales better, Pymnet's 3D visualization is better suited for small demonstrative purposes, where the key idea of a studied system needs to be conveyed as clearly as possible. We believe that the two libraries are in this sense complementary, as we found Py3plex to be the preferrable tool used for visualization of actual, larger multilayer networks.

In terms of scalability, we attempted to re-implement some of the common bottlenecks, such as the layout computation, aggregation, and indexing using efficient vectorized operations. However, we have yet to improve the traversal algorithms, as they are not the key focus of this work. We will re-implement random graph generation and crawling first in C, and eventually on GPUs for maximum performance.

Finally, we believe that Py3plex will serve as a testbed for the development of novel algorithms, especially the ones focusing on multialyer dynamics and structure. An example which proves this functionality is the recently implemented variation of layer

entanglement (Renoust et al. 2014), which was constructed using only the data structures offered by Py3plex. By offering fast prototyping, this library can serve to exchange the ideas related to multilayer network analysis.

One of the main drawbacks of Py3plex visualizations is the cognitive load. While the user can currently observe high-level organization of a multilayer network, it can be hard to interpret the structure of individual layers and their contributions to the structure of the whole network. Even though Py3plex can plot directly to an interactive canvas, we believe this issue represents an interesting research direction and will be addressed in follow up work.

As the Py3plex analysis suite is by no means exhaustive, further work includes the implementation of network dynamics analysis methods as well as the more efficient, GPU-based random samplers. Py3plex aims to bridge the gap between machine learning approaches and complex networks, and it does not yet include extensive tensor manipulation, unfolding, and construction routines offered in the Pymnet library.

Appendix A: Network decomposition functions

In this section, we give an overview of the heterogeneous network decomposition functions introduced in HINMINE (Kralj et al. 2018), supported by Py3plex.

Given a heuristic function f, a weight w of an edge between the two nodes u and v is computed as follows:

Let B denote the set of all nodes of the base type. We use the following notations: $f(t,d)$ denotes the number of times a term t appears in the document d and D denotes the corpus (a set of documents). We assume that the documents in the set are labeled, each document belonging to a class c from a set of all classes C. We use the notation $t \in d$ to describe that a term t appears in document d. Where used, the term $P(t)$ is the probability that a randomly selected document contains the term t, and $P(c)$ is the probability that a randomly selected document belongs to class c. We use $|d|$ to denote the length (in words) of a document, and avgdl denotes the average document length in the corpus. All weight functions (heuristics) supported by Py3plex are summarized in Table 3.

Table 3 Term weighing schemes, taken from (Kralj et al. 2018), tested for decomposition of heterogeneous networks and their corresponding formulas

Scheme	Formula
`tf`	$f(t,d)$
`if-idf`	$f(t,d) \cdot \log\left(\frac{\lvert D\rvert}{\lvert\{d' \in D : t \in d'\}\rvert}\right)$
`chi^2`	$f(t,d) \cdot \sum_{c \in C} \frac{(P(t \wedge c)P(\neg t \wedge \neg c) - P(t \wedge \neg c)P(\neg t \wedge c))^2}{P(t)P(\neg t)P(c)P(\neg c)}$
`ig`	$f(t,d) \cdot \sum_{c \in C, c' \in \{c,\neg c\} t' \in \{t,\neg t\}} \left(P(t',c') \cdot \log \frac{P(t' \wedge c')}{P(t')P(c')}\right)$
`gr`	$f(t,d) \cdot \sum_{c \in C} \frac{\sum_{c' \in \{c,\neg c\}} \sum_{t' \in \{t,\neg t\}} \left(P(t',c') \cdot \log \frac{P(t' \wedge c')}{P(t')P(c')}\right)}{-\sum_{c' \in \{c,\neg c\}} P(c) \cdot \log P(c)}$
`delta-idf`	$f(t,d) \cdot \sum_{c \in C} \left(\log \frac{\lvert c\rvert}{\lvert\{d' \in D : d' \in c \wedge t \in d'\}\rvert} - \log \frac{\lvert \neg c\rvert}{\lvert\{d' \in D : d' \notin c \wedge t \notin d'\}\rvert}\right)$
`rf`	$f(t,d) \cdot \sum_{c \in C} \log\left(2 + \frac{\lvert\{d' \in D : d' \in c \wedge t \in d'\}\rvert}{\lvert\{d' \in D : d' \notin c \wedge t \notin d'\}\rvert}\right)$
`bm25`	$f(t,d) \cdot \log\left(\frac{\lvert D\rvert}{\lvert\{d' \in D : t \in d'\}\rvert}\right) \cdot \frac{k+1}{f(t,d)+k \cdot \left(1-b+b \cdot \frac{\lvert d\rvert}{\text{avgdl}}\right)}$

Acknowledgements
We would like to thank to Lucija Luetič for proofreading the manuscript, which notably improved the work's quality. The work of the first author was funded by the Slovenian Research Agency through a young researcher grant. The work of other authors was supported by the Slovenian Research Agency (ARRS) core research programme *Knowledge Technologies* (P2-0103) and ARRS funded research project *Semantic Data Mining for Linked Open Data* (financed under the ERC Complementary Scheme, N2-0078). The work was supported also by European Union's Horizon 2020 research and innovation programme under grant agreement No 825153, project EMBEDDIA (Cross-Lingual Embeddings for Less-Represented Languages in European News Media).

Authors' contributions
The authors' contributions are as follows. BŠ and JK implemented the library and conducted the experiments. BŠ, JK and NL designed the experiments and conducted theoretical overviews and analysis. All authors read and approved the final manuscript.

Availability of data and materials
The Py3plex library as well as the datasets used in this paper are freely accessible at web site https://github.com/SkBlaz/Py3plex, where many working examples of the Py3plex functionality are also offered.

Competing interests
The authors declare that they have no competing interests.

Received: 9 March 2019 Accepted: 30 August 2019
Published online: 29 October 2019

References
Amato R, Kouvaris NE, San Miguel M, Díaz-Guilera A (2017) Opinion competition dynamics on multiplex networks. New J Phys 19(12)
Auber D, Archambault D, Bourqui R, Delest M, Dubois J, Lambert A, Mary P, Mathiaut M, Mélançon G, Pinaud B, et al. (2017) TULIP 5. Springer
Auber D (2004). In: Jünger M, Mutzel P. (eds). Tulip — A Huge Graph Visualization Framework. Springer, Berlin. pp 105–126
Bastian M, Heymann S, Jacomy M (2009) Gephi: an open source software for exploring and manipulating networks. In: Third International AAAI Conference on Weblogs and Social Media
Batagelj V, Mrvar A (2001) Pajek—analysis and visualization of large networks. In: International Symposium on Graph Drawing. Springer. pp 477–478
Blondel VD, Guillaume J-L, Lambiotte R, Lefebvre E (2008) Fast unfolding of communities in large networks. J Stat Mech Theory Exp 2008(10):10008
Boccaletti S, Bianconi G, Criado R, del Genio CI, Gómez-Gardeñes J, Romance M, Sendiña-Nadal I, Wang Z, Zanin M (2014) The structure and dynamics of multilayer networks. Phys Rep 544(1):1–122
De Domenico M, Nicosia V, Arenas A, Latora V (2015) Structural reducibility of multilayer networks. Nat Commun 6:6864
De Domenico M, Porter MA, Arenas A (2015) MuxViz: A tool for multilayer analysis and visualization of networks. J Complex Netw 3(2):159–176
De Domenico M, Solé-Ribalta A, Cozzo E, Kivelä M, Moreno Y, Porter MA, Gómez S, Arenas A (2013) Mathematical formulation of multilayer networks. Phys Rev X 3(4). https://doi.org/10.1103/physrevx.3.041022
Goyal P, Ferrara E (2018) Graph embedding techniques, applications, and performance: A survey. Knowl-Based Syst 151:78–94
Grover A, Leskovec J (2016) Node2vec: Scalable feature learning for networks. In: Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '16. ACM, New York. pp 855–864
Hachul S, Jünger M (2006) An experimental comparison of fast algorithms for drawing general large graphs. In: Healy P, Nikolov NS (eds). Graph Drawing. Springer, Berlin, Heidelberg. pp 235–250
Hagberg A, Swart P, S Chult D (2008) Exploring network structure, dynamics, and function using NetworkX. In: Proceedings of the 7th Python in Science Conference (SciPy)
Harel D, Koren Y (2002) Graph drawing by high-dimensional embedding. In: International Symposium on Graph Drawing. Springer. pp 207–219
Holland PW, Laskey KB, Leinhardt S (1983) Stochastic blockmodels: First steps. Soc Networks 5(2):109–137
Huang W, Eades P, Hong S-H (2009) Measuring effectiveness of graph visualizations: A cognitive load perspective. Inf Vis 8(3):139–152
Hunter JD (2007) Matplotlib: A 2d graphics environment. Comput Sci Eng 9(3):90
Jacomy M, Venturini T, Heymann S, Bastian M (2014) ForceAtlas2, A continuous graph algorithm for handy network visualization designed for the Gephi software. PloS ONE 9(6):98679
Jones E, Oliphant T, Peterson P, et al (2001) SciPy: Open source scientific tools for Python
Kivelä M, Arenas A, Barthelemy M, Gleeson JP, Moreno Y, Porter MA (2014) Multilayer networks. J Complex Netw 2(3):203–271
Kullback S, Leibler RA (1951) On information and sufficiency. Ann Math Stat 22(1):79–86
Kralj J, Robnik-Šikonja M, Lavrač N (2018) HINMINE: Heterogeneous Information Network Mining with Information Retrieval Heuristics. J Intell Inf Syst 50(1):29–61
Kralj J, Robnik-Sikonja M, Lavrac N (2019) Netsdm: Semantic data mining with network analysis. J Mach Learn Res 20(32):1–50
Leskovec J, Sosič R (2016) Snap: A general-purpose network analysis and graph-mining library. ACM Trans Intell Syst Technol 8(1):1–1120
Maaten Lvd, Hinton G (2008) Visualizing data using t-sne. J Mach Learn Res 9(Nov):2579–2605

McGee F, Ghoniem M, Melançon G, Otjacques B, Pinaud B (2019) The state of the art in multilayer network visualization. Comput Graph Forum 0(0). https://doi.org/10.1111/cgf.13610
McInnes L, Healy J, Melville J (2018) Umap: Uniform manifold approximation and projection for dimension reduction. arXiv preprint arXiv:1802.03426
Nepusz G, Csárdi G (2006) The igraph software package for complex network research. Complex Syst 1695(5):1–9
Omodei E, De Domenico MD, Arenas A (2015) Characterizing interactions in online social networks during exceptional events. Front Phys 3:59
Orchard S, Ammari M, Aranda B, Breuza L, Briganti L, Broackes-Carter F, Campbell NH, Chavali G, Chen C, Del-Toro N, et al (2013) The mintact project—intact as a common curation platform for 11 molecular interaction databases. Nucleic Acids Res 42(D1):358–363
Pavlopoulos GA, O'Donoghue SI, Satagopam VP, Soldatos TG, Pafilis E, Schneider R (2008) Arena3d: visualization of biological networks in 3d. BMC Syst Biol 2(1):104
Perozzi B, Al-Rfou R, Skiena S (2014) Deepwalk: Online learning of social representations. In: Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14. ACM, New York. pp 701–710
Purchase H (1997) Which aesthetic has the greatest effect on human understanding? In: International Symposium on Graph Drawing. Springer. pp 248–261
Renoust B, Melançon G, Viaud M-L (2014). In: Missaoui R, Sarr I (eds). Entanglement in Multiplex Networks: Understanding Group Cohesion in Homophily Networks. Springer, Cham. pp 89–117
Ribeiro LFR, Saverese PHP, Figueiredo DR (2017) Struc2vec: Learning node representations from structural identity. In: Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '17. ACM, New York. pp 385–394
Rosvall M, Axelsson D, Bergstrom CT (2009) The map equation. Eur Phys J Spec Top 178(1):13–23
Secrier M, Pavlopoulos GA, Aerts J, Schneider R (2012) Arena3d: visualizing time-driven phenotypic differences in biological systems. BMC Bioinformatics 13(1):45
Shannon P (2003) Cytoscape: A software environment for integrated models of biomolecular interaction networks. Genome Res 13(11):2498–2504
Škrlj B, Kralj J, Lavrač N (2019) Cbssd: community-based semantic subgroup discovery. J Intell Inf Syst. https://doi.org/10.1007/s10844-019-00545-0
Škrlj B, Kralj J, Vavpetič A, Lavrač N (2018) Community-based semantic subgroup discovery. In: Appice A, Loglisci C, Manco G, Masciari E, Ras ZW (eds). New Frontiers in Mining Complex Patterns. Springer, Cham. pp 182–196
Škrlj B, Kralj J, Lavrač N (2019) Py3plex: A library for scalable multilayer network analysis and visualization. In: Aiello LM, Cherifi C, Cherifi H, Lambiotte R, Lió P, Rocha LM (eds). Complex Networks and Their Applications VII. Springer, Cham. pp 757–768
Smilkov D, Thorat N, Nicholson C, Reif E, Viégas FB, Wattenberg M (2016) Embedding projector: Interactive visualization and interpretation of embeddings. arXiv preprint arXiv:1611.05469
The Boost Graph Library (2002) User Guide and Reference Manual. Addison-Wesley Longman Publishing Co., Inc., Boston
Walt Svd, Colbert SC, Varoquaux G (2011) The numpy array: a structure for efficient numerical computation. Comput Sci Eng 13(2):22–30
Wang Z, Wang L, Szolnoki A, Perc M (2015) Evolutionary games on multilayer networks: a colloquium. Eur Phys J B 88(5):124
Zitnik M, Leskovec J (2017) Predicting multicellular function through multi-layer tissue networks. Bioinformatics 33(14):190–198

Publisher's Note