Network visualizations with Pyvis and VisJS

Giancarlo Perrone[‡*], Jose Unpingco[‡], Haw-minn Lu[‡]

✦

Abstract—Pyvis is a Python module that enables visualizing and interactively manipulating network graphs in the Jupyter notebook, or as a standalone web application. Pyvis is built on top of the powerful and mature VisJS JavaScript library, which allows for fast and responsive interactions while also abstracting away the low-level JavaScript and HTML. This means that elements of the rendered graph visualization, such as node/edge attributes can be specified within Python and shipped to the JavaScript layer for VisJS to render. This declarative approach makes it easy to quickly explore graph visualizations and investigate data relationships. In addition, Pyvis is highly customizable so that colors, sizes, and hover tooltips can be assigned to the rendered graph. The network graph layout is controlled by a front-end physics engine that is configurable from a Python interface, allowing for the detailed placement of the graph elements. In this paper, we outline use cases for Pyvis with specific examples to highlight key features for any analysis workflow. A brief overview of Pyvis' implementation describes how the Python front-end binding uses simple Pyvis calls.

Index Terms—networks, graphs, relationship

Introduction

Successful Data Science is about discovering meaningful relationships in data. Visually representing these relationships using a network graph helps to accelerate understanding and make data driven decisions. Many research areas take advantage of the insight that network analysis techniques can offer. Fields in social networking, cognitive studies, telecommunications, and biological systems all leverage the applications of network theory and computation. Representing these relationships using a network graph is fundamental to all approaches, but generating an interactive and fluid graph visualization can be challenging, especially for large datasets. We introduce Pyvis, based upon the mature VisJS [vis20b] JavaScript library which enables fluid and interactive visualizations of complex network graphs. Pyvis seeks to simplify the interactive process by implementing an existing JavaScript graphics library to abstract away the low-level front end components, leaving the construction of these network data structures to Python.

The Pyvis network data structure matches the JavaScript VisJS object. This makes it easy to interpret and implement the underlying data structures from the Python layer, since the actual front end component is generated by the JavaScript library. A resulting static HTML document shows the network graph, with interactions such as dragging, zooming, hovering, and clicking.

_____

* *Corresponding author: gperrone@westhealth.org*
‡ *Gary and Mary West Health Institute*

These interactions help visualize dense complex networks that are hard to explore using static graphics.

Before open-sourcing Pyvis, we used it successfully to understand relationships among hundreds of variables in a complex survey. Although we maintained an efficient data structure to represent the trends in the survey responses, we still needed a way to visualize and interact with additional metadata. Pyvis made it easy to abstract our existing data structure into nodes and edges with our desired metadata and then render the visualization with VisJS to easily identify the interrelationships. In this paper, we describe the design of Pyvis with examples showing the data structures which are rendered by VisJS.

In the following section, we demonstrate how to get up and running with Pyvis in a smaller scope by showing off the common methods of creating a network. This will also include some exposure to the customizability options that makes Pyvis so useful.

In the Layout section, we will see exactly how nodes and edges can be spatially specified by interacting with various physics parameters interpreted by the front end engine.

Integrations with Jupyter and NetworkX will be presented to establish Pyvis compatibility with popular data science workflows.

Finally, a thought out example will include the interpretation of a practical Game of Thrones relationship dataset to demonstrate a Pyvis use case from the ground up. This minimal example will be a base case for the features that Pyvis supports.

Pyvis Usage

Installing Pyvis is straight-forward with details at the project documentation website [Gia18]. All of the following examples will utilize familiar Python data structures with some connections to the popular and powerful NetworkX package [HSS08]. The basic `Network` class is the container for graph and front end properties. All networks must be instantiated as a `Network` class instance:

```
from pyvis.network import Network
g = Network()
```

Nodes can be added by providing an integer or string `id` and an optional label.

```
g.add_node(1)
g.add_node(2)
print(g)
```

```
{
  "Nodes": [
      1,
      2
  ],
  "Edges": [],
  "Height": "500px",
```
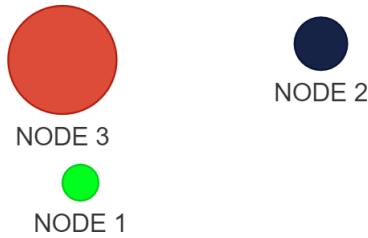
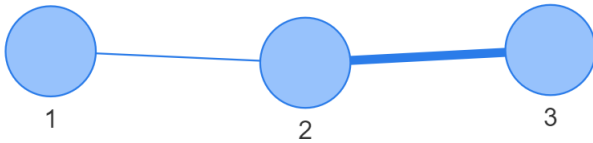*Fig. 1: Multiple nodes and attributes added at once*

Fig. 2: Edges with a custom weight

```
 "Width": "500px"
}
```

The `add_nodes` method consumes a list of nodes (**Fig 1**):

```
nodes = ["a", "b", "c", "d"]
g.add_nodes(nodes)
g.add_nodes("hello")
```

Keyword arguments can be used to add properties to the nodes in `Network`:

```
g = Network()
g.add_nodes(
    [1,2,3],
    value=[10, 100, 400], # values adjust node size
    x=[21.4, 154.2, 11.2],
    y=[100.2, 23.54, 32.1],
    label=["NODE 1", "NODE 2", "NODE 3"],
    color=["#00ff1e", "#162347", "#dd4b39"]
)
g.show("example.html")
```

The following node properties influence the resulting visualization:

- size - The raw circumference of a single node
- value - Circumference of node but scaled according to all values
- title - The title displays over each node while mousing over it
- x - X coordinate of node for custom layouts
- y - Y coordinate of node for custom layouts
- label - A label appearing under each node
- color - The color of the node

Nodes must exist in the network instance in order to add edges

```
g.add_edge(1, 2)
# will adjust edge thickness
g.add_edge(2, 3, weight=5)
```

Edges can be added all at once by supplying a list of tuples to a call to *add_edges()*. The following is an equivalent result (**Fig 2**):

```
g.add_edges([(1, 2), (2, 3, 5)])
g.show("example.html")
```

Notice how an optional element is included in the 3-tuple above (2, 3, 5) representing the weight of the edge. This additional edge data allows for expressing weighted networks and is clearly noticeable in the visualization.

Layout

In situations where your network involves complex connections, Pyvis allows you to manually explore these relationships with intuitive mouse interactions. Nodes can be dragged into more visible positions if the view is obstructed.

All of this is made possible by the front end engine provided by VisJS. Their extensive documentation defines several options for supplying layout and physics configurations to instances of a network. These physics options are fundamental to VisJS, so tweaking the physics of the rendered simulation is as simple as providing the parameters to the specific solver.

The physics options dictates how a user can interact with the objects in the graph. The intent of the physic options is to make manipulating graph objects feel more intuitive when moving nodes around. As an example, the user can manipulate a portion of a graph that is densely populated to view a graph segment of the interest more clearly. VisJS implements several physical simulations such as Barnes Hut [BH86]. Others are mentioned in the VisJS documentation [vis20a].

We can configure the physics engine from within Pyvis:

```
g = Network()
# physics solvers supported:
# barnesHut, forceAtlas2Based, repulsion,
# hierarchicalRepulsion
g.barnes_hut(
 gravity=-80000,
 central_gravity=0.3,
 spring_length=250,
 spring_strength=0.001,
 damping=0.09,
 overlap=0,
)
print(g.options.physics)
{'enabled': True,
'stabilization':
<pyvis.physics.Physics.Stabilization
object at 0x7f99e6a03f90>,
'barnesHut': <pyvis.physics.Physics.barnesHut
object at 0x7f99e6de3710>}
```

In order to avoid the scenario of "guessing" parameter values for an optimal network physics configuration, VisJS offers a useful interaction for experimenting with theses values.

These interactions are enabled via Pyvis (**Fig 3**):

```
# choose to only show the physics options
g.show_buttons(filter_=["physics"])
```

Here, we choose to display the options for the physics component of the network. Omitting a filter in the call will display the configuration of the entire network including nodes, edges, layout, and interaction. The JSON options displayed in the visualization represent the current configuration depending on the displayed sliders. You can copy/paste those options to supply your network with custom settings:
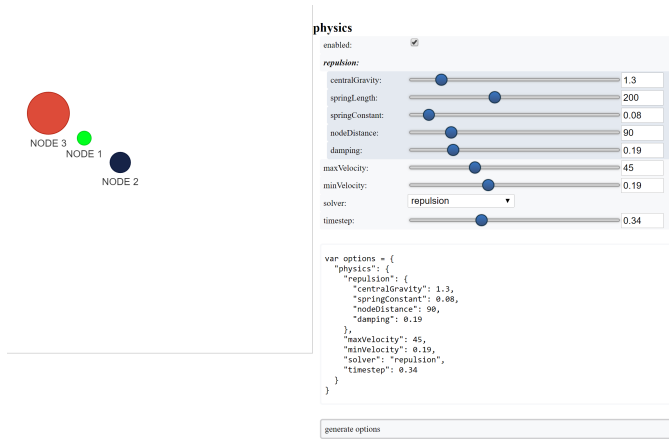
Fig. 3: Live layout GUI with physics filter

Fig. 4: NetworkX graph rendered with Pyvis

Fig. 5: Network rendered in Jupyter Notebook cell

```
g.set_options(
    """
    var options = {
        "physics": {
            "repulsion": {
                "centralGravity": 1.3,
                "springConstant": 0.08,
                "nodeDistance": 90,
                "damping": 0.19
            },
            "maxVelocity": 45,
            "minVelocity": 0.19,
            "solver": "repulsion",
            "timestep": 0.34
        }
    }
    """
)
print(g.options)

{'physics': {'repulsion': {'centralGravity': 1.3,
'springConstant': 0.08,
'nodeDistance': 90,
'damping': 0.19},
'maxVelocity': 45,
'minVelocity': 0.19,
'solver': 'repulsion',
'timestep': 0.34}}
```

The methods of a `Network` instance construct an internal structure compatible with VisJS, demonstrated by the consistent pattern of JSON outputs seen above.

NetworkX Support

Although Pyvis supports its own methods for constructing a network data structure, you might feel more comfortable using the more established and dedicated NetworkX package. Pyvis allows you to define a NetworkX graph instance to then supply it to Pyvis (**Fig** 4).

```
import networkx as nx
from pyvis.network import Network

nxg = nx.random_tree(20)
g=Network(directed=True)
g.from_nx(nxg)
g.show("networkx.html")
```

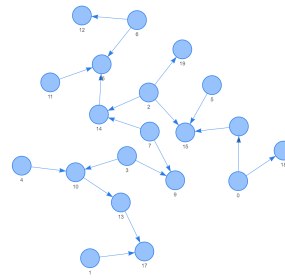Pyvis current behavior recognizes the basic topology of a NetworkX graph, not accounting for any custom attributes provided. Any other attributes like node color, size, and layout would need to be manually added to the resulting Pyvis graph.

Future plans are to fully integrate NetworkX graphs to fully interpret them, preserving attributes in the resulting Pyvis visualizations.

Jupyter Support

For efficient prototyping of visualized graphs, Pyvis aims to utilize Jupyter's front-end IFrame features to embed the graph in a notebook output cell. With that in mind, embedding a Pyvis visualization into a Jupyter notebook is essentially the same as described above. The only difference is that one should pass in a notebook argument during instantiation. The result of the visualization is shown in the output cell below the `show()` invocation. Pyvis upon the call to `show()` writes the HTML that serves an IFrame, which displays the result in the output cell (**Fig 5**).

One thing to keep in mind is that an HTML file is always generated due to the dependence on the VisJS JavaScript bindings.

Example

To get a better understanding of the flow of a typical Pyvis network visualization, we can take a look at the following code snippet to show off a typical application of the features. I have taken a Game of Thrones dataset ([Bev] Storm of Swords Dataset) defining the relationships between characters and the frequencies between them to create a network to naturally
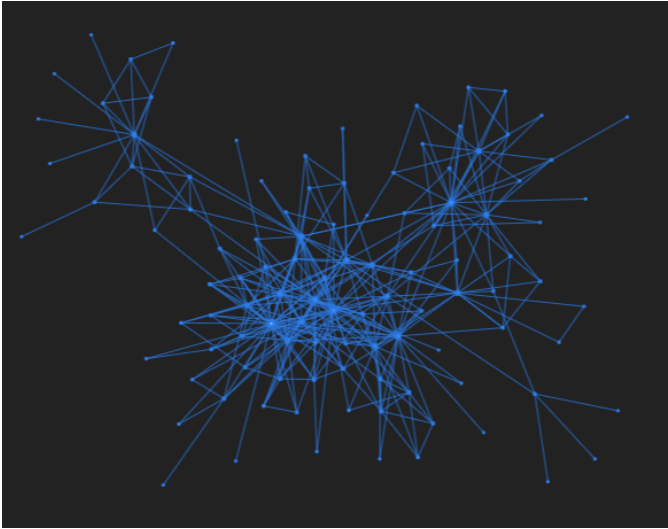
Fig. 6: *Game of Thrones network dictates relationships between characters*

Fig. 7: *Zooming into Game of Thrones network offers concise view*

express this. Specifically, it is a csv file containing pairs of characters and a weight between them. The final visualization contains 107 nodes. (**Fig** 6)

```
from pyvis.network import Network
import pandas as pd

got_net = Network(
    height="750px",
    width="100%",
    bgcolor="#222222",
    font_color="white"
)

# set the physics layout of the network
got_net.barnes_hut()
got_data = pd.read_csv("stormofswords.csv")

sources = got_data['Source']
targets = got_data['Target']
weights = got_data['Weight']

edge_data = zip(sources, targets, weights)

for e in edge_data:
    src = e[0]
    dst = e[1]
    w = e[2]

    got_net.add_node(src, src, title=src)
    got_net.add_node(dst, dst, title=dst)
    got_net.add_edge(src, dst, value=w)

neighbor_map = got_net.get_adj_list()

# add neighbor data to node hover data
for node in got_net.nodes:
    node["title"] += " Neighbors:<br>" + \
            "<br>".join(neighbor_map[node["id"]])
    node["value"] = len(neighbor_map[node["id"]])

got_net.show("gameofthrones.html")
```

In the network, the size of a node correlates to the number of relationships it contains. This calculation benefits from the use of an adjacency list to easily record the information pertaining to each node's neighbors. To see this, the character "Tyrion"

contains many connections, resulting in a larger node. Furthermore, Tyrion's strongest connections are easily noticed by the thick edges, and it is easy to see that Tyrion and Sansa are a strong relationship in the network.

At a glance, the resulting relationship network looks too intertwined to make any practical conclusions. However, the beauty of Pyvis is that each and every component of the network can be focused. For example, zooming in to a dense portion of the network, we can hover over a particular node to get a glimpse of the scenario. (**Fig** 7)

This hover tooltip offers the context behind a particular node. We can see the immediate neighbors for each and every node since we provided a *title* attribute during the network construction. This simple example can be expanded upon to create more custom interactions tailored to specific needs of a dataset.

The network also uses weights. By providing a *value* attribute to each node we can see these values being represented by a node's size. In the code I used the amount of neighbors to dictate the node weight. This is a strong visual cue which makes it easy to see which nodes have the most connections.

The edge weights are assigned in a similar manner, although the dataset already provided the connection strength between nodes. These edge weights are distinguishable in the final visualization, once again proving the usefulness of Pyvis' front-end features.

Under the Hood

VisJS reduces the definition of a network to a declarative set of objects. Nodes, Edges, and an Options JSON object are given to the VisJS Network constructor. The following basic example from their documentation proves this:

```
// create an array with nodes
var nodes = new vis.DataSet([
    {id: 1, label: 'Node 1'},
    {id: 2, label: 'Node 2'},
]);

// create an array with edges
var edges = new vis.DataSet([
    {from: 1, to: 2},
]);
```

```javascript
// create a network
var container = document.getElementById('mynetwork');

// provide the data in the vis format
var data = {
   nodes: nodes,
   edges: edges
};
var options = {};

// initialize your network!
var network = new vis.Network(container, data, options);
```

This pattern makes Jinja [Pro] templating an obvious candidate for generalizing a set of JavaScript declarations. VisJS documentation provides a complete set of supported attributes for each data structure, so incorporating them into the Python layer involves representing each object as Python objects which are then serialized and sent to Jinja to handle the templating.

A simple example of this process in action is outlined below:

```python
self.html = template.render(nodes=nodes, edges=edges)
```

In this case, a template HTML file is rendered with node and edge data matching a format compatible with a VisJS Network instance.

Conclusion

Pyvis is a powerful python module for visualizing and interactively manipulating network graphs in a standalone web application or a Jupyter notebook. Pyvis brings the power of VisJS to Python, thus enabling data scientists who use Jupyter to interactively visualize network graphs with all the fluid interactions of a pure-JavaScript application. Future directions for Pyvis include supporting the front end interactivity with more JavaScript enabled features, and optimization/caching of node positions for larger networks. Those with JavaScript and VisJS knowledge would be able to provide insight towards prospective front end features, since these will leverage actual VisJS references. As Pyvis use case grows in scope, additional features and suggestions will be requested, paving the path for a robust version of Pyvis meeting user experience expectations.

Code samples presented here, and with the corresponding poster presentation, as well as other supplemental material are available at West Health's github repository at https://github.com/WestHealth/scipy2020/tree/master/pyvis.

REFERENCES

[Bev]   Andrew Beveridge. Network of thrones. URL: https://www.macalester.edu/~abeverid/thrones.html.
[BH86]   J K Barnes and Piet Hut. A hierarchical o(n log n) force-calculation algorithm. *Nature*, 324:446–449, 1986. doi:10.1038/324446a0.
[Gia18]   Giancarlo Perrone. Pyvis interactive network visualizations, 2018. URL: https://pyvis.readthedocs.io/en/latest/.
[HSS08]   Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11 – 15, Pasadena, CA USA, 2008. doi:10.25080/issn.2575-9752.
[Pro]   The Pallets Projects. jinja. URL: https://jinja.palletsprojects.com/.
[vis20a] vis.js community. Network - physics, 2020. URL: https://visjs.github.io/vis-network/docs/network/physics.html.
[vis20b] vis.js community. vis.js, 2020. URL: https://visjs.org.