← Go Back to Data Analysis & Visualization

Course Content

New Package Introduction - Network Analysis

NetworkX

NetworkX is a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex **networks**. It is widely used in Network Analysis in Python, and has several inbuilt functionalities to help with the same. Some of NetworkX's features:

- Tools for studying the structure and dynamics of social, biological, and infrastructure networks
- A standard programming interface and graph implementation suitable for a wide range of applications
- A rapid development environment for collaborative and multidisciplinary projects
- An interface for existing numerical algorithms and code written in C, C++, and FORTRAN; and
- The ability to work with large non-standard data sets without difficulty

You can use NetworkX to load and save networks in standard and non-standard data formats; generate many different types of random and classic networks, analyze network structure; build network models, design new network algorithms, draw and visualize networks and a lot more.

In order to use the NetworkX package, we need to first install it on our local machine. You can install NetworkX using the pip command.

To install the package, please run the below command in a Jupyter notebook and once the command runs successfully, restart the kernel:

```
!pip install networkx
```

To import the package, run the following command in a Jupyter notebook:

```
import networkx as nx
```

There are many useful functions in NetworkX to work with graphs/network data. Let's go through some of the important functions that we use in NetworkX case studies that implement network analysis:

1. nx.Graph() to create a graph:

A graph is defined as a collection of nodes (vertices) and identified pairs of nodes (called edges, links, etc.). Nodes in NetworkX can be any hashable object, such as text, string, an image, an XML object, another Graph, a customized node object, and so on.

To create an empty graph, we need to call the **Graph()** class as shown below.

```
import networkx as nx
```

```
G = nx.Graph()
```

2. nx.from_pandas_edgelist():

From the above function, we can get a Pandas dataframe containing an edge list.

The parameters used by the function are as shown below:

```
from_pandas_edgelist(df, source='source', target='target', edge_attr=None, create_using=None, edge_key=None)
```

df: Pandas data frame

A graph's edge list representation

Source: str or int

For the source nodes, a valid column name (string or integer) (for the directed case)

target: str or int

For the target nodes, a valid column name (string or integer) (for the directed case).

edge_attr: str or int, iterable, True, or None

A valid column name (str or int) or iterable column name is used to retrieve items and add them as edge attributes to the graph. If true, the rest of the columns will be added. If none is specified, no edge attributes are added to the graph.

Create_using: NetworkX graph constructor, optional (default=nx.Graph).

To create a graph type. If this is a graph instance, it must be cleared before being populated.

edge_key : str or None, optional (default=None)

A valid column name for the edge keys (for a MultiGraph). If create using is a multigraph, the values in this column are used as edge keys when adding edges.

Below we see an implementation of this function in the case study titled Game of Thrones Network Analysis:

```
# nx.from_pandas_edgelist returns a graph from a Pandas DataFrame containing an edge list
G1 = nx.from_pandas_edgelist(book1, 'Person 1', "Person 2", edge_attr = "weight", create_using = nx.Graph())

G2 = nx.from_pandas_edgelist(book2, 'Person 1', "Person 2", edge_attr = "weight", create_using = nx.Graph())
```

3. nx.degree_centrality(G):

The degree centrality of a node v is the proportion of nodes to which it is connected.

The parameters used by the function are as shown below:

G: Graph

A NetworkX graph

nodes: dictionary

Dictionary of nodes with the value degree centrality

Below we see a function implementation of nx.degree_centrality() in the case study titled Game of Thrones Network Analysis:

```
# Degree Centrality
''' nx.degree_centrality(G) computes the degree centrality for nodes.
The degree centrality for a node v is the fraction of nodes it is connected to.'''

def deg_central(G):
    deg_centrality = nx.degree_centrality(G)
```

4. nx.eigenvector_centrality(G):

Eigenvector centrality calculates a node's centrality based on the centrality of its neighbors. The ith element of the vector defined by the equation is the eigenvector centrality for the node.

$$Ax = \lambda x$$

where is graph G's adjacency matrix with eigenvalue. If is the largest eigenvalue of the adjacency matrix, there is a unique solution with all of its entries being positive, according to the Perron-Frobenius theorem.

The parameters used by the function are as shown below:

```
eigenvector_centrality(G, max_iter=100, tol=1e-06, nstart=None, weight=None)
```

G: Graph

A NetworkX graph

max_iter: integer, optional (default=100)

A maximum number of iterations in power method.

tol: float, optional (default=1.0e-6)

Error tolerance is used to check convergence in power method iteration.

nstart: dictionary, optional (default=None)

the starting value of the eigenvector iteration for each node.

weight: None or string, optional (default=None)

If None, all edge weights are treated equally. Otherwise, the name of the edge attribute used as weight is stored. The weight is interpreted as the connection strength in this measure.

Below we see a function implementation of nx.eigenvector_centrality() in the case study titled Game of Thrones Network Analysis:

```
# Eigenvector Centrality
''' nx.eigenvector_centrality computes the eigenvector centrality for the graph G.
Eigenvector centrality computes the centrality for a node based on the centrality of its neighbors.
The eigenvector centrality for node i is the i-th element of the vector x defined by the equation Ax=kx'''

def eigen_central(G):
    eigen_centrality = nx.eigenvector_centrality(G, weight = "weight")
```

5. nx.betweenness_centrality(G):

The betweenness centrality of a node v is the fraction of all-pairs shortest paths that pass through v.

$$c_B(v) = \sum_{s,t \in V} \frac{\sigma(s,t|v)}{\sigma(s,t)}$$

where V is the set of nodes, $\sigma(s,t)$ is the number of shortest (s,t)-paths, and $\sigma(s,t|v)$ is the number of those paths passing through some node v other than s,t. If $s = t$, $\sigma(s,t) = 1$, and if $v \in s,t$, $\sigma(s,t|v) = 0$ [2].

The parameters used by the function are as shown below:

```
betweenness_centrality(G, k=None, normalized=True, weight=None, endpoints=False, seed=None)
```

G: Graph

A NetworkX graph

k: int, optional (default=None)

If k is None, estimate betweenness using k node samples. The value of k < = n, where n is the number of graph nodes. Higher values provide a more accurate approximation.

normalized: bool, optional

If True, the betweenness values for graphs and directed graphs are normalized by 2/((n-1)(n-2)), where n is the number of nodes in G.

weight : None or string, optional (default=None)

If none, all edge weights are treated equally. Otherwise, the name of the edge attribute used as weight is stored. Weights are used to calculate weighted shortest paths, which are then translated into distances.

endpoints: bool, optional

Include the endpoints in the shortest path count if true.

seed: integer, random_state, or None (default)

An indicator of the state of random number generation. See Randomness. It should be noted that this is only used if k is not None.

Below we see a function implementation of nx.betweenness_centrality() in the case study titled Game of Thrones Network Analysis:

```
# Betweenness Centrality
'''nx.betweenness_centrality(G) computes the shortest-path betweenness centrality for nodes.
Betweenness centrality of a node v is the sum of the fraction of all-pairs shortest paths that pass through v.
'''
def betweenness_central(G):
    betweenness_centrality = nx.betweenness_centrality(G, weight = "weight")
```

The following is a helpful article from AskPython.org that gives an in-depth introduction to NetworkX:

NetworkX Package - Python Graph Library

Louvain

This package adds the capability of community detection using the Louvain method. The name of the package is **community**, but it refers to Python-Louvain on PyPI.

The Louvain Community Detection Algorithm is a straightforward method for determining a network's community structure. It is a heuristic method for optimizing modularity.

To install the package, please run the below command in a Jupyter notebook and once the command runs successfully, restart the kernel:

```
!pip install python-louvain
```

To import the package, run the following command in a Jupyter notebook:

```
import community as community_louvain
```

1. community_louvain.best_partition()

```
community_louvain.best_partition(graph, partition=None, weight='weight,' resolution=1.0, randomize=None, random
```

The parameters used by the function are as shown below:

graph: networkx.Graph

The NetworkX graph that has been decomposed into

partition: dict, optional

The algorithm will begin to use this node partition. It's a dictionary in which the keys are nodes and the values are communities.

weight: str, optional

the key in the graph to be used as a weight. Default to 'weight'.

Resolution: double, optional

The default value of 1 represents the time described in "Laplacian Dynamics and Multiscale Modular Structure in Networks." M. Barahona, R. Lambiotte, J.-C. Delvenne

Randomize: boolean, optional

Will generate different partitions at each call by randomizing the node evaluation order and the community evaluation order.

random_state: int, RandomState instance or None, optional (default=None)

If a random state is an int, it is the seed used by the random number generator; if a random state is an instance of RandomState, it is the random number generator. If None, the RandomState instance used by np. random is used as the random number generator.

Below we see a function implementation of community_louvain.best_partition() in the case study titled Game of Thrones Network Analysis:

```
# compute the best partition
partition = community_louvain.best_partition(G, random_state = 12345)
```

Please find below the documentation page for the Community package for your reference:

Community (Python-Louvain)

Colorlover

To install the package, run the below command in a Jupyter notebook and once the command runs successfully, restart the kernel.

```
!pip install colorlover
```

To import the package, run the following command in a Jupyter notebook

```
import colorlover as cl
```

The parameters used by the function are shown below:

Pos:

A Dictionary of Nodes

Labels:

list of labels of len(pos) to be displayed

Color :

color of nodes. If it is "None", the plotly's default color is used.

Size:

Size of the dots representing the nodes

Opacity:

The Opacity value is between 0 and 1, defining the color opacity.

Below we can find the function implementation in the Game of Thrones case study:

```
colors = cl.scales['12']['qual']['Paired']

def scatter_nodes(G, pos, labels = None, color = 'rgb(152, 0, 0)', size = 8, opacity = 1):
```

Please find below the documentation page for the colorlover package for your reference:

Link

Happy Learning!

Previous Next