Published in Towards Data Science

Vatsal Follow
Mar 21 · 7 min read · · Listen

Save

Louvain's Algorithm for Community Detection in Python

Apply Louvain's Algorithm in Python for Community Detection

Image taken by Ethan Unzicker from Unsplash

network of your choice using the NetworkX and Python-Louvaine module. The following is the structure of the article:

Table of Contents

What is Community Detection?

In graph theory, a network has a community structure if you are able to group nodes (with potentially overlapping nodes) based on the node's edge density. This would imply that the original network G, can be naturally divided into multiple subgraphs / communities where the edge connectivity within the community would be very dense. Overlapping communities are also allowed, so there can be overlapping nodes in the communities formed. This implies that nodes in separate communities have a sparse amount of edges.

> *The more general definition is based on the principle that pairs of nodes are more likely to be*

Thinking of this intuitively, it makes sense. Consider yourself in your own social networks like Instagram. You might be highly connected into many different communities related to things you're interested in. You could have a follow / following of accounts associated with friends, memes, sports, anime, etc. Each of those categorizations could be interpreted as communities, where you as a user are a node and the edges are generated by connecting you to other users who have similar interests as yourself. Thus, within your own social network, you would have a very dense community, and would be sparsely connected to other individuals outside of your communities.

There are many mathematical formulations for identifying communities within a given network. The focus of this article is specifically on the Louvain algorithm, however there exists many other algorithms like the Girvan–Newman algorithm, Jaccard index, etc. which can be used to solve problems in community detection.

Community Detection vs Clustering

Similar to clustering, traditional approaches to community detection can be labelled as unsupervised learning. The argument could be made that community detection and clustering are both one and the same. There have been many circumstances in literature where the terms have been used interchangeably.

I find that the distinction between the two is quite subtle, community detection focuses on generating groups of nodes based on the network structure, whereas clustering focuses on generating groups based on many attributes associated with the input data. Community detection remains specifically in the domain of graph theory and network analysis while clustering is traditionally used in non graph based applications. This is to say it is not impossible to use traditional clustering techniques like kMeans on a network (one would first need to generate embeddings associated with nodes within the network and then apply the clustering model to those embeddings).

What is Louvain's Algorithm

communities in large networks" [3] where they introduced a greedy method which would generate communities in O(n*log(n)) time where n is the number of nodes in the original network [2].

Modularity

Louvain's algorithm aims at optimizing modularity. Modularity is a score between -0.5 and 1 which indicates the density of edges within communities with respect to edges outside communities [2]. The closer the modularity is to -0.5 implies non modular clustering and the closer it is to 1 implies fully modular clustering. Optimizing this score yields the best possible groups of nodes given a network. Intuitively, you can interpret it as maximizing the difference between the actual number of edges and the expected number of edges. Modularity can be defined by the following formula :

$$Q = \frac{1}{2m} \sum_{ij} \left[A_{ij} - \frac{k_i k_j}{2m} \right] \delta(c_i, c_j)$$

Formula to calculate modularity on a weighted network. Image taken from Wikipedia [2]

- `Ai,j` represents the edges between nodes i and j
- `m` is the sum of all edge weights in the network
- delta is the Kronecker delta function
  - delta = 1 if i =j
  - delta = 0 otherwise
- `Ci` and `Cj` are the communities of the nodes
- `Ki` and `Kj` is the sum of weights connecting nodes i and j

Louvain's Algorithm

To maximize the modularity, Louvain's algorithm has two iterative phases. The first positive increase in the modularity. If there is no modular increase then the node remains in its current community, otherwise the communities are merged. This merging of communities can be represented by the following formula:

$$\Delta Q = \left[\frac{\Sigma_{in} + 2k_{i,in}}{2m} - \left(\frac{\Sigma_{tot} + k_i}{2m}\right)^2\right] - \left[\frac{\Sigma_{in}}{2m} - \left(\frac{\Sigma_{tot}}{2m}\right)^2 - \left(\frac{k_i}{2m}\right)^2\right]$$

Image taken from Wikipedia [2]

> *Where* `Sigma_in` *is sum of all the weights of the links inside the community* `i` *is moving into,* `Sigma_tot` *is the sum of all the weights of the links to nodes in the community* `i` *is moving into,* `k_i` *is the weighted degree of* `i`, `k_i,in` *is the sum of the weights of the links between* `i` *and other nodes in the community that* `i` *is moving into, and* `m` *is the sum of the weights of all links in the network.*
> *- [3] https://en.wikipedia.org/wiki/Louvain_method*

Both of the phases are executed until there is no possible modularity gain by merging communities together.

Problem Statement

The problem we're going to try and solve is pretty straight forward, given a network, we want to identify communities within that network.

Requirements

```
Python=3.8.8
numpy=1.20.1
python-louvain=0.16
networkx=2.5
matplotlib=3.3.4
```

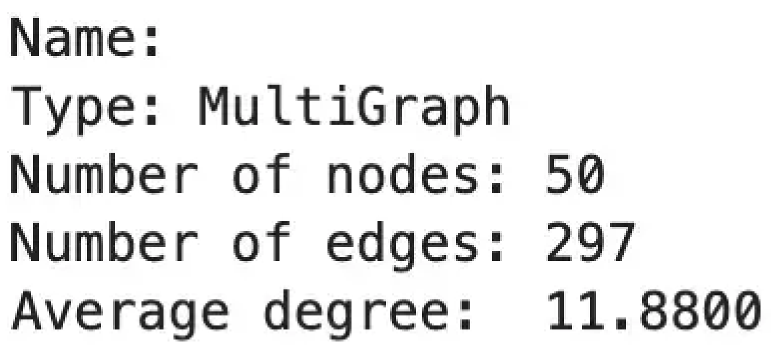
If you don't have the networkX package installed, here is the library documentation to

Generate Network

```python
import random
import networkx as nx
import numpy as np
from community import community_louvain
import matplotlib.pyplot as plt


def generate_network(n):
    '''
    This function will generate a random weighted network associated to the user specifed
    number of nodes.

    params:
        n (Integer) : The number of nodes you want in your network

    returns:
        A networkX multi-graph

    example:
        G = generate_network(n)
    '''
    # initialize dictionary with nodes
    graph_dct = {node:[] for node in range(n)}
    nodes = list(range(n))

    # generate edges
    for n,edge_list in graph_dct.items():
        edge_c = random.randint(min(nodes), int(max(nodes) / 2))
        el = random.sample(nodes, edge_c)
        graph_dct[n] = el

    # create networkx multi-edge graph
    G = nx.MultiGraph(graph_dct)
    return G

n = 500
G = generate_network(n)
```

```
plt.show()
```

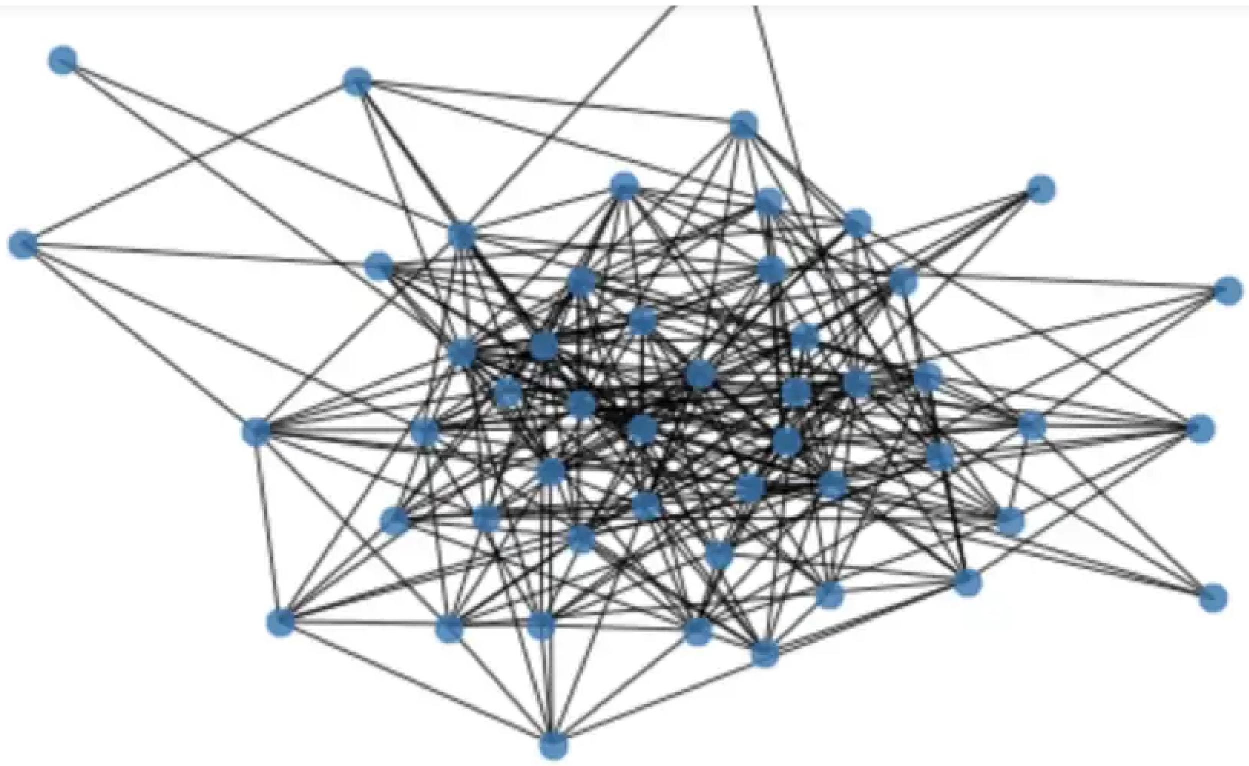
The following function above generates a network associated with a randomly generated degree distribution. The user can specify the number of nodes they would like in the associated network, for the purposes of this tutorial I selected 50. The following is the network statistics from the sample network I created.



```
Name:
Type: MultiGraph
Number of nodes: 50
Number of edges: 297
Average degree:  11.8800
```

Network statistics. Image provided by the author.

The following is a visual representation of how the network looks.


How the randomly generated network looks. Image provided by the author.

Apply Louvain's Algorithm

We can simply apply Louvain's algorithm through the Python-Louvain module. You can find further documentation associated with the function we're going to be referencing here [4].

```python
comms = community_louvain.best_partition(G)
```

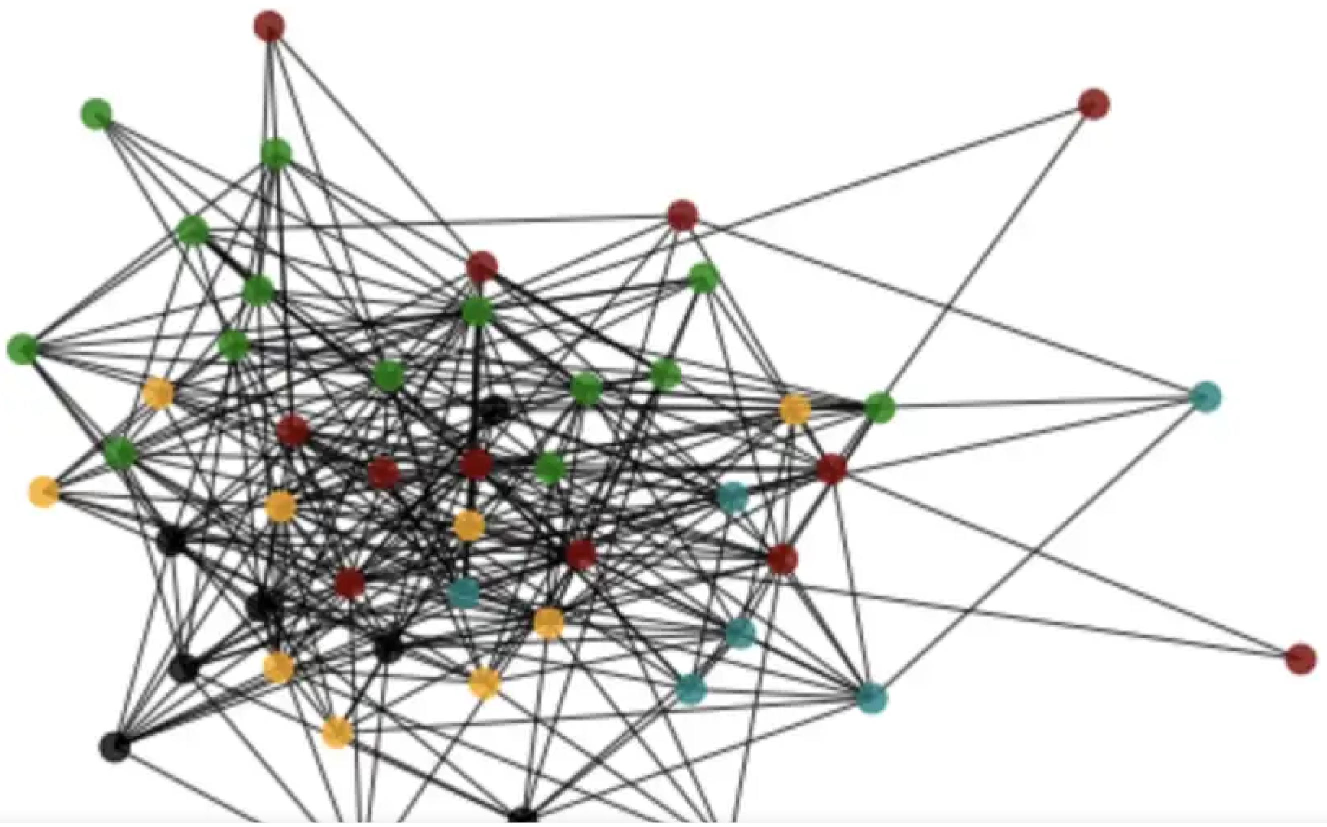
louvaine.py hosted with ❤ by GitHub view raw

This should return the associated communities detected from G in the form of a dictionary. The keys of the dictionary are the nodes and the values correspond to the community in which that node belongs to.

Visualize Communities

Be aware that this visualization is only possible because I selected a small number of

```python
unique_coms = np.unique(list(comms.values()))
cmap = {
    0 : 'maroon',
    1 : 'teal',
    2 : 'black',
    3 : 'orange',
    4 : 'green',
    5 : 'yellow'
}

node_cmap = [cmap[v] for _,v in comms.items()]

pos = nx.spring_layout(G)
nx.draw(G, pos, node_size = 75, alpha = 0.8, node_color=node_cmap)
plt.show()
```



Be aware that the results associated with the pipeline introduced here are quite meaningless since the algorithm was run on a randomly generated network.

Concluding Remarks

This article only introduced one of the many potential algorithms associated with community detection. There are many various algorithms introduced in mathematics to solve problems related to community detection. It would be very beneficial to your own development to learn, implement and test the performance of each algorithm on various datasets.

You can follow along on the code associated with this article on my GitHub in the following Jupyter Notebook I've created.

Resources

- [1] https://en.wikipedia.org/wiki/Community_structure
- [2] https://en.wikipedia.org/wiki/Louvain_method
- [3] https://arxiv.org/abs/0803.0476
- [4] https://github.com/taynaud/python-louvain
- [5] https://pypi.org/project/python-louvain/

If you enjoyed this article, there are many others on network analysis, nlp, recommendation systems, etc. which I've also written about. Check them out below.

Text Similarity w/ Levenshtein Distance in Python

Building a Plagiarism Detection Pipeline in Python

towardsdatascience.com

Get unlimited access

Open in app

Building a MultiClass Node Classification Model in Python

towardsdatascience.com

Text Summarization in Python with Jaro-Winkler and PageRank

Building a Text Summarizer with Jaro-Winkler and PageRank

towardsdatascience.com

Link Prediction Recommendation Engines with Node2Vec

Using Node Embeddings for Link Prediction in Python

towardsdatascience.com

Mining & Modelling Character Networks — Part II

This article will cover the Python implementation of mining & modelling character networks

towardsdatascience.com

Word2Vec Explained

Explaining the Intuition of Word2Vec & Implementing it in Python

towardsdatascience.com

389

Get unlimited access Open in app

Hybrid Recommendation Systems in Python

towardsdatascience.com

Node2Vec Explained

Explaining & Implementing the Node2Vec Paper in Python

towardsdatascience.com

Bayesian A/B Testing Explained

towardsdatascience.com

Sign up for The Variable

By Towards Data Science

Every Thursday, the Variable delivers the very best of Towards Data Science: from hands-on tutorials and cutting-edge research to original features you don't want to miss. Take a look.

Emails will be sent to amokhtarzadeh22@gmail.com. Not you?