Learning in Graphs – Applications to Community Mining

Graph Mining and Analysis with Python

INFMDI341 Machine Learning Avancé

Fragkiskos Malliaros, Michalis Vazirgiannis

June 2, 2014

1 Description

The goal of this lab is to work with graph (or network) data using the NexworkX library of Python (`http://networkx.github.io/`).

2 Part 1: Analyzing a Real-World Graph

In this part of the lab, we will analyze the `CA-GrQc` collaboration network, examining several structural properties. Arxiv GR-QC (General Relativity and Quantum Cosmology) collaboration network is from the e-print arXiv and covers scientific collaborations between authors papers submitted to General Relativity and Quantum Cosmology category. If an author i co-authored a paper with author j, the graph contains an undirected edge from i to j.

The graph is stored in the `ca-GrQc.txt` file[1], as an edge list:

```
# Directed graph (each unordered pair of nodes is saved once): CA-GrQc.txt
# Collaboration network of Arxiv General Relativity category (there is an
edge if authors coauthored at least one paper)
# Nodes: 5242 Edges: 28980
# FromNodeId ToNodeId
3466 937
3466 5233
...
```

1. Load the network data into an undirected graph G, using the `read_edgelist` function. Note that, the delimeter used to separate values is the tab character $\backslash t$ and additionaly, the text that follows the # character are comments. The general syntax of the function is the following:

```
read_edgelist(path, comments='#', delimiter=None, create_using=None,
nodetype=None, data=True, edgetype=None, encoding='utf-8')
```

2. Compute and print the following network characteristics: (1) number of nodes, (2) number of edges and (3) number of connected components. If the graph is not connected, find the connected

[1]The data can be downloaded from the following link: `http://snap.stanford.edu/data/ca-GrQc.txt.gz`.

components and store the largest connected component subgraph to graph GCC (also called *giant connected component*). Find the number of nodes and edges of the largest connected component (GCC) and examine in what fraction of the whole graph they correspond. What do you observe?

3. Analysis of the degree distribution of the graph. Extract the degree sequence of the graph using the following command

```
degree_sequence = G.degree().values()
```

Then, find and print the minimum, maximum, median and mean degree of the nodes of the graph. For this task, you can use the built-in functions `min, max, median, mean` of the NumPy library. Therefore, before that, you have to import the numpy module

```
import numpy as np
from __future__ import division # Depending on the version of Python
```

What do you observe? Let's now compute and plot the degree distribution of the graph. For this reason, we can use the `degree_histogram` function, that returns a list of the frequency of each degree value. Then, we can plot the degree histogram using the `matplotlib` library of Python

```
import matplotlib.pyplot as plt

y=nx.degree_histogram(G)
plt.plot(y,'b-',marker='o')
plt.ylabel("Frequency")
plt.xlabel("Degree")
plt.show()
```

What do you observe? Produce again the plot using log-log axis (`plt.loglog(...)`). How this observation can be interpreted? How this type of distribution is called?

4. Analysis of clustering structures in the graph.
(I) *Triangles*. As we have already discussed in the class, a triangle is a clique of three nodes, i.e., all nodes are connected among each other. Triangle subgraphs play a crucial role in the area of graph mining and social network analysis, since they are closely related to the existence of clustering structures in the graph. Let's now compute the total number of triangles in the `CA-Gr-Qc` collaboration graph using the following command

```
nx.triangles(G)
```

Note that, the `triangles(G)` function returns a dictionary (*(key, value)* form) with the number of triangles that each node participates to. Thus, after that, we need to sum up these values and to divide by 3 (why?) in order to compute the total number of triangles. For the last step, you can use the `sum()` function and the `dict_name.values()` to extract the values of the dictionary. What do you observe?

Additionally, we will compute and plot the triangle participation distribution, i.e., a distribution that shows the number of triangles that each node participates in (how many nodes participate to one triangle, how many nodes participate to two triangles, etc). Note that, this process is similar to the computation of the degree distribution. For this reason, you can use the following Python code (`t` is the dictionary with the number of triangles per node computed in the previous step)

```
t_values = sorted(set(t.values()))
t_hist = [t.values().count(x) for x in t_values]
```

What do these values represent? Then, plotting the values of `t_values` vs. `t_hist` we can obtain the triangle participation distribution. Use the `loglog()` plotting function. What do you observe?

(II) *Clustering Coefficient*. Next we will compute the average clustering coefficient[2] of the graph, which is a measure of the degree to which nodes in a graph tend to cluster together, i.e., to create tightly knit groups characterized by a relatively high density of ties. The *global clustering coefficient* is based on triplets of nodes. A triplet consists of three nodes that are connected by either two (open triplet) or three (closed triplet) undirected ties. A triangle consists of three closed triplets, one centered on each of the nodes. The global clustering coefficient is the number of closed triplets (or 3 x triangles) over the total number of triplets (both open and closed). Use the following built-in function to compute the average clustering coefficient.

```
nx.average_clustering(G)
```

5. NetworkX library also offers several functions for computing properties of a graph, such as node centrality measures[3] [4]. In network analysis, the *centrality* of a node is a measure that captures the relative importance of the node based on specific criteria. The most simple one, the *degree centrality*, is based on the number of neighbors that a node has, and the higher the degree centrality, the more important a node is. Other centrality measures include the *betweenness centrality* (based on the number of shortest paths that pass through a node) and the *eigenvector centrality* (based on the components of the largest eigenvector of the adjacency matrix – this is also the basis of Google's PageRank algorithm). Note that, the computation of some of these measures is costly.

   Let's now compute two of these centrality measures: degree and eigenvector, using the following code

```
# Degree centrality
deg_centrality = nx.degree_centrality(G)

# Eigenvector centrality
eig_centrality = nx.eigenvector_centrality(G)
```

   These functions return a dictionary with the centrality values of each node. Are the degree and eigenvector centrality values correlated? In other words, if a node has high degree centrality, does this imply that the eigenvector centrality will be high as well? Let's first extract the centrality values for each node, sorted according to the node id

```
# Sort centrality values
sorted_deg_centrality = sorted(deg_centrality.items())
sorted_eig_centrality = sorted(eig_centrality.items())

# Extract centralities
deg_data=[b for a,b in sorted_deg_centrality]
eig_data=[b for a,b in sorted_eig_centrality]
```

   Variables `deg_data` and `eig_data` store the centrality values of each node (sorted based on node id). In order to examine the correlation, we can compute the *Pearson correlation coefficient*[5] of these variables, using the built-in function[6]

[2]http://en.wikipedia.org/wiki/Clustering_coefficient
[3]http://networkx.lanl.gov/reference/algorithms.centrality.html
[4]http://en.wikipedia.org/wiki/Centrality
[5]http://en.wikipedia.org/wiki/Pearson_product-moment_correlation_coefficient
[6]http://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.pearsonr.html

```
# Import library
from scipy.stats.stats import pearsonr

print "Pearson correlation coefficient ", pearsonr(deg_data, eig_data)
```

Additionally, we can plot the values of degree vs. eigenvector centrality (using the `plot` function), in order to visually observe potential relationship.

6. Let's now repeat some parts of the above experiment for a random graph. A *random graph* is obtained by starting with a set of n isolated vertices and adding successive edges between them at random. In our case, we will create a random graph using the Erdős-Rényi $G(n, p)$ random graph model, where the graph contains n nodes and each of the edges is included with probability p. Create a random graph R using the $G(200, 0.1)$ model (i.e., 200 nodes and $p = 0.1$), using the following command

   ```
   R = nx.fast_gnp_random_graph(200, 0.1)
   ```

   Now, compute again the minimum, maximum, median and mean degree of the nodes of the graph, as well as the degree distribution (do the same plot as in the previous case (`plot()`)). What do you observe? Is there any difference in the structure of random vs. real graphs (e.g., the `CA-GrQc` collaboration network)?

   In order to make more clear the difference of the structural properties between real-world (e.g., social networks, collaboration networks) and random networks, we can examine additional properties similar to the case of `CA-GrQc` previously. We will focus on the clustering properties, trying to stress out that random graphs do not inherently show a clustering structure. Let's compute again the number of triangles of the random graph (`triangles()` function) and then plot the triangle participation distribution. What do you observe and how this plot can be compared to the case of the `CA-GrQc` real graph? Similarly, what is happening on the average clustering coefficient of random graphs?

7. Write a function `random_sample(G, num_sample_nodes)` that extracts a random sample of nodes from the graph, i.e., nodes that are chosen uniformly at random from the network (the number of these nodes is the `num_sample_nodes` input parameter). Use `sample_nodes = set()` to store the nodes that are sampled. To extract a random node and add it in the set, you can use the following command

   ```
   import random
   sample_nodes.add(random.choice(G.nodes()))
   ```

   Next we provide the basic points and structure of the code for this task.

   ```
   # Import modules
   import networkx as nx
   import random
   import numpy as np

   # Define the function
   def simple_random_sample(G, num_sample_nodes):

     # Body of the function

     return sample_nodes
   ```

```
# Use the main function as it follows
G = nx.read_edgelist("ca-GrQc.txt", comments='#', delimiter='\t',
nodetype=int, create_using=nx.Graph())
sample_nodes = simple_random_sample(G, 5)
```

3 Part 2: Community Detection

In the second part of the lab, we will focus on the community detection (or clustering) problem in graphs. Typically, a community corresponds to a set of nodes that highly interact among each other, compared to the intensity of interactions (as expressed by the number of edges) with the rest nodes of the graph.

Next, we will apply a simple methodology for community detection that is based on the notion of hierarchical clustering. The idea is to construct a tree of clusters, in order to identify groups of nodes with high similarity, based on some similarity measure. Initially, we have to define a similarity (or distance) measure between the nodes of the graph. This is chosen to be the average shortest path length.

1. The experiments for this part will be performed in a small dataset that has been used as a benchmark in several community detection algorithms. The *karate* dataset is a friendship social network between 34 members of a karate club at a US university in the 1970. Load the file from the networkx library

   ```
   z=nx.karate_club_graph()
   ```

2. Visualize this graph, trying to observe the existence of any potential clusters in the graph. For this reason use the `nx.draw_networkx(z,pos)` command, where z is the network and pos is the plotting layout (use the following: `pos=nx.spring_layout(z)`).

3. Run the following code that computes shortest path distances, creates the distance matrix between nodes and applies hierarchical clustering. Note that, in the hierarchical clustering algorithm, the linkage criterion determines the distance between sets of observations (nodes in our case) as a function of the pairwise distances between observations.

   ```
   import matplotlib.pyplot as plt
   from scipy.cluster import hierarchy
   from scipy.spatial import distance

   path_length=nx.all_pairs_shortest_path_length(z)
   n = len(z.nodes())
   distances=numpy.zeros((n,n))

   for u,p in path_length.iteritems():
           for v,d in p.iteritems():
                   distances[int(u)-1][int(v)-1] = d

   hier = hierarchy.average(distances)
   ```

4. Finally, use the dendrogram offered by the `hierarchy` library (`hierarchy.dendrogram(hier)`) to visualize the results of the hierarchical clustering. What do you observe?

References

[1] JP Onnela. Notes in Analysis of Large-Scale Networks using NetworkX. Harvard University, 2013.

[2] Derek Greene. Graph and Network Analysis. Web Science Doctoral Summer School, University College Dublin, 2011.