ADVANCED LEARNING FOR TEXT AND GRAPH DATA
MASTER DATA SCIENCE - MVA
Labs 2 and 3: Graph Mining

Fragkiskos Malliaros, Antoine Tixier and Michalis Vazirgiannis

February 11/17, 2016

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 a 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 (for simplicity, let us use the GCC of the graph):

```
t = nx.triangles(GCC)
```

Note that, the `triangles(GCC)` 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?

Approximate triangle counting. Recall that the (i, j)-th element of the $\mathbf{A}^\ell$ counts the number of paths of length ℓ that start from node i and end at node j. A triangle in a graph is defined as a path of length 3 that starts and ends at the same node. Thus, the i-th diagonal element of $\mathbf{A}^3$ counts the number of triangles that node i participates to. Taking into account that each triangle is counted twice for each of the three participating nodes (clockwise and anticlockwise paths starting from node i), the total number of distinct triangles in the graph will be given by the following formula:

$$\Delta(G) = \frac{1}{6}\,\mathrm{tr}(\mathbf{A}^3) = \frac{1}{6}\sum_{i=1}^{|V|} A_{ii}^3,$$

where $\mathrm{tr}(\mathbf{A})$ is the trace of a matrix. Since matrix $\mathbf{A}$ is a real, square matrix, then $\mathrm{tr}(\mathbf{A}) = \sum_{i=1}^{|V|} \lambda_i$, and more generally, $\mathrm{tr}(\mathbf{A}^\ell) = \sum_{i=1}^{|V|} \lambda_i^\ell$, where $\lambda_1, \ldots, \lambda_{|V|}$ are the eigenvalues of $\mathbf{A}$. Thus, the total number of triangles in G can be computed by the spectrum of the adjacency matrix as follows:

$$\Delta(G) = \frac{1}{6}\sum_{i=1}^{|V|} \lambda_i^3.$$

This formulation offers a spectral computation of the total number of triangles $\Delta(G)$. In general, such a method for computing the number of triangles is of similar complexity as the method that is applied directly on the graph (e.g., function `triangles()` of Python), which is generally high (it can be $\mathcal{O}(|V|^3)$ for dense graphs). The idea here is to approximate the number of triangles by taking into account the special spectral properties of real-world graphs. For this reason, let's compute and plot the eigenvalues of the adjacency matrix $\mathbf{A}$ versus their rank. What do you observe?

Now, we can approximate the number of triangles $\tilde{\Delta}_k(G)$ using only the top k eigenvalues of the adjacency matrix. Why is this possible? Compute and plot the error of approximation for various values of k:

$$\mathrm{error}_k = \frac{|\tilde{\Delta}_k(G) - \Delta(G)|}{\Delta(G)}.$$

What do you observe? How many eigenvalues should we retain to achieve good approximation? Notice that, in practice, the gain that we achieve in terms of execution time is due to the fact that we can efficiently compute the top k eigenvalues of the adjacency matrix using iterative methods, such as the Lanczos algorithm[2], which compute the eigenvalues one by one[3] (thus, it is not required to compute the whole spectrum). Can we also approximate the number of triangles $\Delta^i(G), \quad \forall i \in V$, i.e., in a per node basis? Recall that, since the adjacency matrix $\mathbf{A}$ is real and symmetric, applying eigenvalue decomposition we have that $\mathbf{A} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^T$ and $\mathbf{A}^3 = \mathbf{U}\mathbf{\Lambda}^3\mathbf{U}^T$.

Triangle participation. 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_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[4] of the graph, which is a measure of the degree to which nodes in a graph tend to cluster together, i.e., to create

[2] `https://en.wikipedia.org/wiki/Lanczos_algorithm`
[3] See also the following paper: `http://link.springer.com/article/10.1007%2Fs10115-010-0291-2`
[4] `http://en.wikipedia.org/wiki/Clustering_coefficient`

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[5] [6]. 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*[7] of these variables, using the built-in function[8]

```
# 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.

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

6. **Random graph model.** Let's now repeat this 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. **Kronecker graph model.** As we observed, random graphs are not good models for generating networks with properties similar to those observed in real graphs. In the previous question, we examined graphs generated by the Erdős-Rényi $G(n, p)$ model, and we saw that the produced graphs fail to generate networks with skewed degree distribution. In the related literature, several models have been proposed to deal with this issue. Here, we will examine the degree distribution of a simple graph model that is based on the properties of the **Kronecker product** of matrices. In general, given two matrices $\mathbf{A} = [a_{i,j}]$ of size $n \times m$ and $\mathbf{B} = [b_{i,j}]$ of size $n' \times m'$, the Kronecker product matrix $\mathbf{C}$ of size $(n * n') \times (m * m')$ will be given as follows:

$$\mathbf{C} = \mathbf{A} \otimes \mathbf{B} \begin{pmatrix} a_{1,1}\mathbf{B} & a_{1,2}\mathbf{B} & \cdots & a_{1,m}\mathbf{B} \\ a_{2,1}\mathbf{B} & a_{2,2}\mathbf{B} & \cdots & a_{2,m}\mathbf{B} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1}\mathbf{B} & a_{n,2}\mathbf{B} & \cdots & a_{n,m}\mathbf{B} \end{pmatrix}$$

The Kronecker model was introduced by Leskovec et al. [3] as a simple generation model for real-world graphs, based on the Kronecker product of matrices. More precisely, assuming an initiator adjacency matrix $\mathbf{A}_1$ of size $\ell \times \ell$, the Kronecker graph after k iterations is defined as the graph with the following adjacency matrix:

$$\mathbf{A}_k = \underbrace{\mathbf{A}_1 \otimes \mathbf{A}_1 \otimes \cdots \otimes \mathbf{A}_1}_{k \text{ iterations}} = \mathbf{A}_{k-1} \otimes \mathbf{A}_1.$$

In practice, a stochastic version of the Kronecker model is used, in the sense that the initiator matrix $\mathbf{A}_1$ is not the binary adjacency matrix itself but the probability matrix for the existence of an edge. For example, in the typical case of a 2×2 initiator matrix $\mathbf{A}_1 = [a\ b; c\ d]$, each value represents the probability of existence of the corresponding edge. Starting by such an initiator

matrix and applying the Kronecker product for a desired number of iterations k, the resulting adjacency matrix of the graph corresponds to a realization of the matrix $\mathbf{A}_k$, i.e., each edge (i, j) is introduced to the graph with probability $A_k(i, j)$.

1. Consider the folowing 2×2 initiator matrix $\mathbf{A}_1 = [0.99, 0.26; 0.26, 0.53]$. The values of the initiator matrix are important, in the sense that they are related to the properties of the generated graph. In [3], the authors propose a method to learn these values from the original network.
2. Repeat the Kronecker product for $k = 10$ times and get the $\mathbf{A}_k$ adjacency matrix (use the `np.kron(A,B)` built-in function for Kronecker multiplication). Note that, this matrix is not binary.
3. For each entry $\mathbf{A}_k(i, j)$ of matrix $\mathbf{A}_k$, include the edge (i, j) with probability $\mathbf{A}_k(i, j)$. This matrix will be the adjacency matrix of the final graph. (You can use the `random.random()` function to generate random numbers).

Now that we have the adjacency matrix, we can get the corresponding graph using the function `nx.from_numpy_matrix(A_k, create_using=nx.Graph())`. (For simplicity, we consider the graph as undirected). Then, we can examine the degree distribution of the produced graph. What do you observe?

3 Part 2: Robustness of the Network

In general, the robustness of a network is related to the capability of the network to retain its structure and connectivity properties, after losing a portion of its nodes and edges. Let us focus our attention to the case where the nodes of a network are deleted based on their degree with two strategies:

(i) *Random deletion:* delete a randomly selected node.

(ii) *Targeted deletion:* delete a node chosen among the ones with the highest degree in the network.

Depending on the type of networks, the above two strategies can simulate various scenaria. For example, in the case of the Internet graph (nodes correspond to routers and edges capture physical connections between routers), a random deletion can be interpreted as an *error* that occurred in the network, where a router switched off due to technical problems (e.g., electricity problems). On the other hand, the targeted removal of a high-degree nodes can simulate the case of an attack to the network, where the removal of nodes aims to cause big damage to the network.

How the structure of the network is affected after random and targeted deletions of nodes? It has been noted that, due to the existence to heavy-tailed degree distribution, real-world networks tend to be robust under random removal of nodes (e.g., errors) and vulnerable under the attacks to high degree nodes. The notion of robustness can be quantified by structural characteristics of the network, such as the fragmentation of the network into disconnected components. For instance, we can examine how the fraction of nodes that belong to the largest connected component (GCC) and the rest isolated components is affected by the random/targeted removal of nodes.

In this part of the lab, we will examine the robustness of the `CA-GrQc` network, with respect to the above strategies.

1. For this reason, we will write two Python functions, `random_node_deletion(G, fraction_to_delete)` and `targeted_node_deletion(G, fraction_to_delete)`, for each of the two

strategies. Both functions should take as input the fraction of nodes to be removed and will return the graph after the removal.

2. To assess the robustness, we will examine how the size of GCC and the rest components is affected, under several values of the fraction of removed nodes (e.g., in the range $[0\%, 30\%]$).

3. Plot the size of GCC and the one of the rest components vs. the fraction of deleted nodes, for both strategies (i.e., random and targeted). What do you observe?

4. Then, you can repeat the same experiment for a random graph, generated by the Erdős-Rényi $G(n, p)$ model. Use the following built-in function to construct the graph: `G = nx.fast_gnp_random_graph(1000, 0.01)`. What do you observe?

4 Part 3: 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. Then, we will implement and apply a more sophisticated graph clustering algorithm, called spectral clustering. Additionally, we will see how to evaluate the results of a clustering algorithm.

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.read_gml("karate.gml")
```

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 networkx as nx
import numpy as np
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=np.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)
```

Finally, use the dendrogram offered by the `hierarchy` library (`hierarchy.dendrogram(hier)`) to visualize the results of the hierarchical clustering. What do you observe? In case where the goal is to detect two communities, where should we "cut" the dendrogram?

4. Next, we will implement the *spectral clustering* algorithm. The basic idea of the algorithm is to utilize information associated to the spectrum of the graph, in order to identify well-separated clusters. Next we describe the pseudocode of the algorithm.

Algorithm 1 Spectral Clustering

Input: Graph $G = (V, E)$ and parameter k
Output: Clusters $\mathbf{C}_1, \mathbf{C}_2, \ldots, \mathbf{C}_k$ (i.e., cluster assignments of each node of the graph)

1: Let $\mathbf{A}$ be the adjacency matrix of the graph.
2: Compute the Laplacian matrix $\mathbf{L} = \mathbf{D} - \mathbf{A}$. Matrix $\mathbf{D}$ corresponds to the diagonal degree matrix of graph G (i.e., degree of each node v (= number of neighbors) in the main diagonal).
3: Apply eigenvalue decomposition to the Laplacian matrix $\mathbf{L}$ and compute the eigenvectors that correspond to k smallest eigenvalues. Let $\mathbf{U} = [\mathbf{u}_1|\mathbf{u}_2|\ldots|\mathbf{u}_k] \in \mathbb{R}^{m\times k}$ be the matrix containing these eigenvectors as columns.
4: For $i = 1, \ldots, m$, let $y_i \in \mathbb{R}^k$ be the vector corresponding to the i-th row of $\mathbf{U}$. Apply k-means to the points $(y_i)_{i=1,\ldots,m}$ (i.e., the rows of $\mathbf{U}$) and find clusters $\mathbf{C}_1, \mathbf{C}_2, \ldots, \mathbf{C}_k$.

For the implementation of spectral clustering, you can use the built-in function `kmeans2` for the k-means algorithm (you will need to import the corresponding library: `from scipy.cluster.vq import kmeans2`). Then, we will apply the algorithm to the *karate* dataset, trying to identify two clusters.

5. *Community Evaluation.* To assess the quality of a clustering algorithm, several metrics have been proposed. *Modularity* is one of the most popular and widely used metrics to evaluate the quality of networks partition into communities. Considering a specific partition of the network into clusters, modularity measures the number of edges that lie within a cluster compared to the expected number of edges of a null graph (or configuration model), i.e., a random graph with the same degree distribution. In other words, the measure of modularity is built upon the idea that random graphs are not expected to present inherent community structure; thus, comparing the observed density of a subgraph with the expected density of the same subgraph in case where edges are placed randomly, leads to a community evaluation metric. Modularity is given by the following formula:

$$Q = \sum^{n_c} \left[\frac{l_c}{m} - \left(\frac{d_c}{2m} \right)^2 \right]$$

where, $m = |E|$ is the total number of edges in the graph, n_c is the number of communities in the graph, l_c is the number of edges within the community c and d_c is the sum of the degrees of the nodes that belong to community c. Modularity takes values in the range $[-1, 1]$, with higher values indicating better community structure.

Next, we will use modularity to compare different clustering results of the *karate* dataset. Create a new python script (`modularity.py`) and fill in the body of the `modularity()` function as shown below. Then, compute the modularity of the three clustering results (the third one corresponds to the solution given by the hierarchical clustering algorithm). What is the performance of the spectral clustering algorithm? Also, examine visually the quality of each one of these clustering solutions, drawing the graph and coloring the nodes based on the cluster that they belong to.

```
import networkx as nx

# Define the function of modularity
def modularity(G, clustering):

# Add the body of the function here

return modularity

# Use the main function as it follows
G = nx.read_gml("karate.gml")

# Different clustering solutions
clustering = []
clustering.append([0,0,0,0,0,0,0,0,1,1,0,0,0,0,1,1,0,0,1,0,1,0,1,1,1,1,1,1,1,1,1,1])
clustering.append([0,1,0,1,0,0,1,0,1,1,0,0,0,0,0,1,0,0,0,1,1,0,1,0,1,0,0,1,1,1,0,1,0,1])
clustering.append([0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,1,1,0,0,1,0,1,0,1,1,1,1,1,1,1,1,0,1,1])

# Compute modularity for each case and print the graph
for i in range(len(clustering)):
        print modularity(G, clustering[i])
        nx.draw(G, pos=nx.spring_layout(G), node_color=clustering[i])
        plt.show()
```

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.

[3] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani. Kronecker Graphs: An Approach to Modeling Networks. Journal of Machine Learning Research, 11:985-1042, 2010.