ADVANCED LEARNING FOR TEXT AND GRAPH DATA
MASTER DATA SCIENCE - MVA
Lab 2: Graph Mining

Giannis Nikolentzos, Fragkiskos Malliaros, Antoine Tixier and Michalis Vazirgiannis

February 1, 2017

Description

The goal of this lab is help you become familiar with graph concepts and to work with graph data using the NexworkX library of Python (`http://networkx.github.io/`), a very popular library for the analysis and manipulation of graphs. We will first study the dynamics of a real-world graph. We will then use some algorithms to reveal its community structure. Finally, we will use graph kernels to measure the similarity between pairs of graphs and to perform graph classification.

Part 1: Analyzing a Real-World Graph

In this part of the lab, we will analyze the `CA-HepTh` collaboration network, examining several structural properties. Arxiv HEP-TH (High Energy Physics - Theory) collaboration network is from the e-print arXiv and covers scientific collaborations between authors of papers submitted to High Energy Physics - Theory 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-HepTh` file[1], as an edge list:

```
# Collaboration network of Arxiv High Energy Physics Theory category
# Nodes: 9877 Edges: 51971
# FromNodeId ToNodeId
24325 24394
24325 40517
24325 58507
...
```

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')
```

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

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 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. To this end, 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 can this observation be interpreted? How is this type of distribution called?

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 of the nodes in the graph. The experiments for this part will also be performed in the `CA-HepTh` collaboration network.

1. We will first implement and apply a very popular graph clustering algorithm, called *Spectral Clustering*. The basic idea of the algorithm (described in what follows) is to utilize the information associated with the spectrum of the graph to identify well-separated clusters. Algorithm 1 illustrates the pseudocode of Spectral Clustering.

   For the k-means algorithm, you can use the built-in function `cluster.KMeans` from `scikit-learn` (`from sklearn import cluster`). The algorithm must return a dictionary with nodes as keys and membership (cluster to which the nodes belong) as values. After implementing the algorithm, apply it to the GCC of the `CA-HepTh` dataset, trying to identify 60 clusters.

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 $\mathbf{C}_1, \mathbf{C}_2, \ldots, \mathbf{C}_k$

2. *Community Evaluation.* To assess the quality of a network's partition into communities, several metrics have been proposed. *Modularity* is one of the most popular. 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:

$$Q = \sum^{n_c} \left[\frac{l_c}{m} - \left(\frac{d_c}{2m} \right)^2 \right] \tag{1}$$

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 GCC of the `CA-HepTh` dataset. Create a new python script (`modularity.py`) and fill in the body of the `modularity()` function as shown below.

```
import networkx as nx

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

  # Add the body of the function here

  return modularity
```

Then, compute the modularity of following two clustering results: (i) the one obtained by the Spectral Clustering algorithm using $k = 60$, and (ii) the one obtained if we randomly partition the nodes into 60 clusters. To assign each node to a cluster, use the `randint(a,b)` function which returns a random integer n such that $a \leq n \leq b$. What is the performance of the Spectral Clustering algorithm compared to the algorithm that randomly clusters the nodes?

3. We will next employ *Louvain*, a more sophisticated algorithm for extracting communities from large networks. Louvain is a heuristic method that is based on modularity optimization. In other

words, Louvain seeks for clustering results that maximize modularity. You are given a python script (`community_detection.py`) that contains the function `louvain(G)` which implements the Louvain algorithm. The function returns a dictionary keyed by node to the cluster to which the node belongs. Apply the algorithm on the GCC of the `CA-HepTh` network as follows.

```
from community_detection import louvain

# Partition graph using the Louvain method
partition = louvain(G)
```

Compute the modularity of the obtained clustering result. Since Louvain optimizes modularity, we expect it to be high. Compare the current value of modularity with the value obtained from the random partitioning and the clustering result of Spectral Clustering. What do you observe?

Part 3: Graph Classification

In the last part of the lab, we will focus on the problem of graph similarity and graph classification. Graph classification arises in the context of a number of classical domains such as chemical data, biological data, and the web. In order to perform graph classification, we will employ graph kernels, a powerful framework for graph comparison.

In recent years, kernel methods have become very popular in machine learning. Kernels can be intuitively understood as functions measuring the similarity of pairs of objects. More formally, for a function $k(x, x')$ to be a kernel, it has (1) to be symmetric: $k(x, x') = k(x', x)$ and (2) to satisfy Mercer's condition, i. e. to be positive semi-definite. If a function satisfies the above two conditions on a set $\mathcal{X}$, it is known that there exists a map $\phi : \mathcal{X} \to \mathbb{H}$ into a Hilbert space $\mathbb{H}$, such that $k(x, x') = \langle \phi(x), \phi(x') \rangle$ for all $(x, x') \in \mathcal{X}^2$ where $\langle \cdot, \cdot \rangle$ is the inner product in $\mathbb{H}$ [5]. A Hilbert space is a complete inner product space [6]. In more details, an inner product space is a vector space endowed with an inner product operation. That inner product gives rise to a norm $||x|| = \sqrt{\langle x, x \rangle}$. If the inner product space is complete in this norm (i.e. if every Cauchy sequence of elements of the space converges to an element in the space), then the space is called a Hilbert space. Kernel functions compute the inner product between examples that are mapped in a higher-dimensional feature space. However, they do not explicitly compute the feature map ϕ for each example.

One advantage of kernel methods is that they can operate on very general types of data such as images and graphs. Kernels defined on graphs are known as *graph kernels*. Graph kernels allow kernel-based learning approaches such as SVMs to work directly on graphs. In this part of the lab, we will implement a recently introduced graph kernel, and we will compare it against two baseline kernels in the task of graph classification.

1. To perform graph classification, we need a dataset containing a list of graphs along with their class labels. We will create such a dataset by performing community detection on the GCC of the `CA-HepTh` graph. The intuition is the following: we expect the communities detected by an algorithm to share some common characteristics and structural properties. Consider the *Louvain* and *Spectral Clustering* algorithms. Since the two algorithms optimize different objective criteria, we also expect the structural properties of the communities detected by the first to differ from that of the communities detected by the second. Hence, we can create a dataset containing all the communities (subgraphs) detected by the two algorithms, and the task will then be to predict if a community was extracted by Louvain or by Spectral Clustering.

   Create a function `create_dataset()` and fill in its body as shown below.

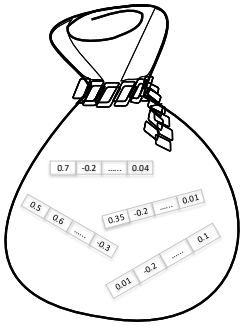

Figure 1: The bag-of-vectors representation.

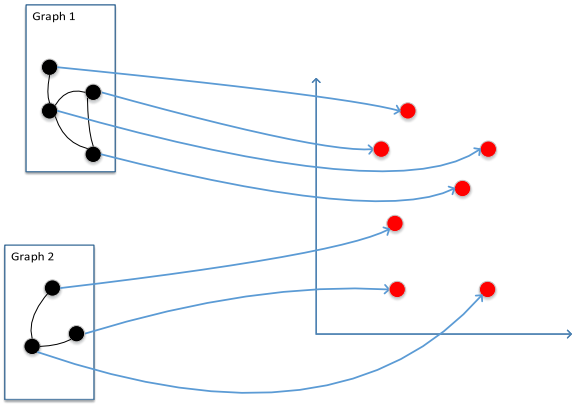

Figure 2: Graphs are embedded into a vector space using the eigenvectors associated with the largest eigenvalues (in magnitude).

```
# Define the function that creates the dataset
def create_dataset(G):

  # Add the body of the function here

  return graphs, labels
```

Specifically, create a list `graphs` consisting of all the communities detected by the two algorithms and a second list `labels` of the same size containing their class labels (1 if the community was detected by Louvain, -1 otherwise). Set the number of clusters to be detected by *Spectral Clustering* to 60. Given a list of nodes `l`, you can get the subgraph induced by these nodes using the function `G.subgraph(l)`.

2. Given the dataset you created, your next task is to perform graph classification and to predict if a community was detected by the Louvain or the Spectral Clustering algorithm. To this end, you will implement the recently proposed *Pyramid Match graph kernel*. The Pyramid Match graph kernel assumes that each graph is represented as a bag-of-vectors as shown in Figure 1. These vectors correspond to the embeddings of the nodes of the graph. More specifically, let $\mathcal{G} = \{G_1, G_2, \ldots, G_N\}$ be a population of N graphs. Your task is to embed the nodes of each graph in the d-dimensional space as shown in Figure 2 (for a pair of graphs). To generate these embeddings, use the eigenvectors of the adjacency matrix corresponding to the d largest in magnitude

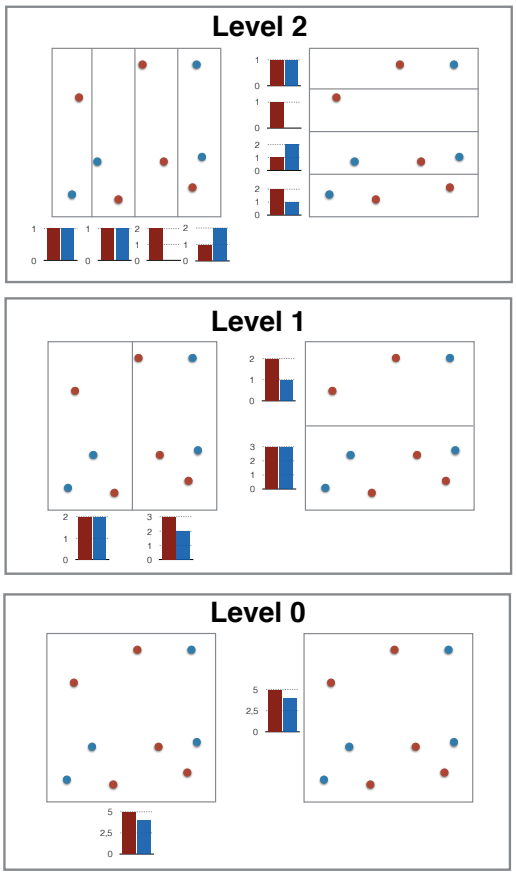

Figure 3: Example of histogram creation in the 2-dimensional space.

eigenvalues. Each row of the resulting matrix is the embedding of a node. Note that you can get the adjacency matrix of a graph using the function `nx.adjacency_matrix(G)`. To get the d largest in magnitude eigenvalues and their corresponding eigenvectors, you can use the function `eigs(A,k=d)` of `SciPy`. Sort the emerging eigenvectors by the magnitude of their eigenvalues. Notice that there may be some communities with $n < d$, where n is the number of nodes. We can embed the nodes of these graphs at most in the n-dimensional space. To project the nodes to the d-dimensional space, set the remaining $d - n$ coordinates to 0.

3. Given two graphs and their bag-of-vectors representations, the Pyramid Match graph kernel maps these vectors to multi-resolution histograms, and then compares the histograms with a weighted histogram intersection measure in order to find an approximate correspondence between the two sets of vectors. The Pyramid Match graph kernel works by partitioning the feature space into regions of increasingly larger size and taking a weighted sum of the matches that occur at each level as shown in Figure 3. Two points are said to match if they fall into the same region. Matches made within larger regions are weighted less than those found in smaller regions. Recall that in our setting, the embedding of each node is a point lying in the d-dimensional unit hypercube. We repeatedly fit a grid with cells of increasing size to the d-dimensional unit hypercube. Each cell is related only to a specific dimension and its size along that dimension is doubled at each iteration, while its size along the other dimensions stays constant and equal to 1. Given a sequence of levels from 0 to L, then at level l, the d-dimensional unit hypercube has 2^l cells along each dimension and $D = 2^l d$ cells in total. Given a pair of graphs $G_1, G_2 \in \mathcal{G}$, let $H^l_{G_1}$ and $H^l_{G_2}$ denote the histograms

of G_1 and G_2 at level l, and $H^l_{G_1}(i)$, $H^l_{G_2}(i)$, the number of vertices of G_1, G_2 that lie in the i^{th} cell. The number of points in two sets which match at level l is then computed using the histogram intersection function:

$$I(H^l_{G_1}, H^l_{G_2}) = \sum_{i=1}^{D} \min\left(H^l_{G_1}(i), H^l_{G_2}(i)\right) \tag{2}$$

The matches that occur at level l also occur at levels $0, \ldots, l-1$. We are interested in the number of new matches found at each level which is given by $I(H^l_{G_1}, H^l_{G_2}) - I(H^{l+1}_{G_1}, H^{l+1}_{G_2})$ for $l = 0, \ldots, L-1$. The number of new matches found at each level in the pyramid is weighted according to the size of that level's cells. Matches found within smaller cells are weighted more than those made in larger cells. Specifically, the weight for level l is set equal to $\frac{1}{2^{L-l}}$. Hence, the weights are inversely proportional to the length of the side of the cells that varies in size as the levels increase. The pyramid match kernel is then defined as follows:

$$k_\Delta(G_1, G_2) = I(H^L_{G_1}, H^L_{G_2}) + \sum_{l=0}^{L-1} \frac{1}{2^{L-l}} \left(I(H^l_{G_1}, H^l_{G_2}) - I(H^{l+1}_{G_1}, H^{l+1}_{G_2})\right) \tag{3}$$

By computing the kernel for all pairs of graphs, we can build a positive semidefinite kernel matrix $\mathbf{K} \in \mathbb{R}^{N \times N}$ such that $\mathbf{K}_{ij} = k_\Delta(G_i, G_j)$ for some $G_1, \ldots, G_N \in \mathcal{G}$. Kernel matrices are symmetric, positive semidefinite and are also known as Gram matrices. Next we describe the pseudocode for computing the kernel matrix.

Algorithm 2 Pyramid Match Graph Kernel

Input: Bag-of-vectors representations of N graphs $(G_1, \ldots, G_N)$, and parameters L and d
Output: Kernel matrix $\mathbf{K}$

```
for i ← 1 to N do
    for l ← 1 to L do
        for j ← 1 to 2^l d do
            Count how many points of G_i fall into region j (i.e. calculate H^l_{G_i}(j))
        end for
    end for
end for

for i ← 1 to N do
    for j ← i to N do
        for l ← 1 to L do
            Compute intersection of histograms of G_i and G_j at level l (equation 2)
        end for
        Compute kernel value K_ij (equation 3))
        K_ji ← K_ij
    end for
end for
```

Write a function `compute_pm_kernel()` and fill in its body as shown below.

```
# Define the function that computes the Pyramid Match graph kernel
def compute_pm_kernel(graphs, L, d):

  # Add the body of the function here

  return K
```

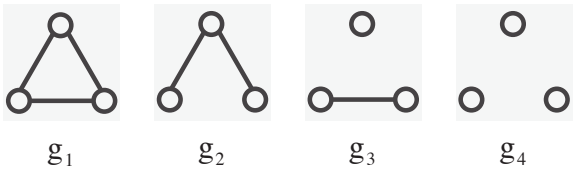

Figure 4: The four different graphlets of size 3.

The function takes as input a list of graphs, the number of levels L, and the number of dimensions d, and returns the emerging kernel matrix. Run the function to compute the kernel matrix. Set $d = 10$ and $L = 5$.

4. Afer creating the kernel matrix $\mathbf{K}$, to evaluate the algorithm, we will perform graph classification using the function `svm_classification(K,labels)` that you are given (`kernel_eval.py` script). The function makes use of an SVM classifier, and performs 10-fold cross-validation, while the parameter C of the SVM is optimized on the training set only. The function returns a dictionary with accuracies and optimal values of C for all folds. The average performance of the classifier corresponds to the value keyed by the string `mean_accuracy`. Print the mean accuracy and comment on the obtained result. Could the kernel distinguish between the communities generated by Louvain and those generated by Spectral Clustering?

5. You are given a python script (`baseline_kernels.py`) containing the implementations of two additional graph kernels:

   - The first kernel is called the *Graphlet kernel*, and compares graphs by counting graphlets. A graphlet corresponds to a small subgraph (typically of 3,4 or 5 vertices). Figure 4 illustrates all the graphlets of size 3. Use the function `compute_graphlet_kernel(graphs)` to compute the Graphlet kernel for connected graphlets of size 3. The function takes as input a set of graphs and returns the kernel matrix $\mathbf{K} \in \mathbb{R}^{N \times N}$ where N is the number of graphs in the dataset. The kernel matrix $\mathbf{K}$ stores the kernel values for all pairs of graphs as described above.
   - The second kernel is the *Weisfeiler-Lehman subtree kernel*. For a number of iterations h, the Weisfeiler-Lehman subtree kernel counts pairs of matching subtree patterns of height 0 in two graphs, while at each iteration updates the labels of the nodes of the graph using the Weisfeiler-Lehman test of isomorphism. Since our graphs are not labeled, we set the labels of the nodes equal to their degrees. The function `compute_wl_subtree_kernel(graphs,h)` takes as input a set of graphs and the number of iterations h to be performed by the Weisfeiler-Lehman algorithm, and returns the kernel matrix $\mathbf{K} \in \mathbb{R}^{N \times N}$ where N is the number of graphs in the dataset.

   Use the two functions to compute the kernel matrices of the two baseline kernels, and evaluate them in the graph classification task. Set the parameter h of the Weisfeiler-Lehman subtree kernel to 6. Follow the same experimental procedure as in the case of the Pyramid Match graph kernel. Compare the performance of the two kernels against the performance of the Pyramid Match graph kernel. 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.

[3] S. Fortunato, and D. Hric. Community detection in networks: A user guide. Physics Reports 659: 1-44, 2016.

[4] S. Vishwanathan, N. Schraudolph, R. Kondor, and K. Borgwardt. Graph kernels. Journal of Machine Learning Research, 1201-1242, 2010.

[5] A. Smola, and B. Schlkopf. Learning with kernels. GMD-Forschungszentrum Informationstechnik, 1998.

[6] M. Reed, and B. Simon. Functional Analysis, Methods of Modern Mathematical Physics. Academic Press, 1980