ADVANCED LEARNING FOR TEXT AND GRAPH DATA

Lab session 6: Graph mining

Lecture: Prof. Michalis Vazirgiannis
Lab: Giannis Nikolentzos and Stratis Limnios

December 20, 2018

1 Introduction

The goal of this lab is help you become familiar with graph concepts and to work with graph data. The lab is divided into three parts. In the first part, we will study the dynamics of a real-world graph. Then, we will use some algorithms to reveal its community structure. Finally, we will use graph kernels to measure the similarity between graphs and to perform graph classification. To analyze and manipulate graphs, we will use the NetworkX library (`http://networkx.github.io/`), while we will also make use of the GraKeL library (`https://github.com/ysig/GraKeL/`) in order to measure graph similarity.

2 Analyzing a Real-World Graph

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

```
# Directed graph (each unordered pair of nodes is saved once): CA-HepTh.txt
# Collaboration network of Arxiv High Energy Physics Theory category (there is an edge if autho
# Nodes: 9877 Edges: 51971
# FromNodeId    ToNodeId
24325   24394
24325   40517
24325   58507
24394   3737
24394   3905
24394   7237
...
```

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

1. Load the network data into an undirected graph G, using the `read_edgelist()` function of NetworkX. Note that, the delimeter used to separate values is the tab character $\backslash t$ and additionaly, that lines that start with 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 components and store the largest connected component subgraph (also called *giant connected component*) to variable GCC. 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 = [d for n, d in G.degree()]
```

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

```
import numpy as np
```

   What do you observe? Let's now compute and plot the degree distribution of the graph. To do that, 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?

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. 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 is to utilize information associated to the spectrum of the graph, in order to identify well-separated clusters. Algorithm 1 illustrates the

pseudocode of Spectral Clustering.

Algorithm 1 Spectral Clustering

Require: Graph $G = (V, E)$ and parameter k

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

To perform k-means, you can use scikit-learn's implementation of the algorithm (you will need to import the corresponding library: `from sklearn.cluster import KMeans`). The Spectral Clustering algorithm must return a dictionary keyed by node to the cluster to which the node belongs. After implementing the algorithm, apply it to the GCC of the `CA-HepTh` dataset, trying to identify 60 clusters.

2. *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 a network's 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:

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

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_result):

  # Add the body of the function here

  return modularity
```

Then, compute the modularity of the 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 the *Clauset-Newman-Moore* algorithm, a more sophisticated algorithm for extracting communities from large networks. This is a greedy algorithm that is based on modularity optimization. In other words, it seeks for clustering results that maximize modularity. You can apply the Clauset-Newman-Moore algorithm to the GCC of the `CA-HepTh` network as follows:

```
from networkx.algorithms.community import greedy_modularity_communities

# Partition graph using the Clauset-Newman-Moore algorithm
partition = greedy_modularity_communities(GCC)
```

Compute the modularity of the obtained clustering result. Since the Clauset-Newman-Moore algorithm 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?

4 Graph Classification using Graph Kernels

In the last part of the lab, we will focus on the problem of 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.

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 to be (1) symmetric: $k(x, x') = k(x', x)$, and (2) positive semi-definite. If a function satisfies the above two conditions on a set $\mathcal{X}$, it is known that there exists a map $\phi : \mathcal{X} \to \mathcal{H}$ into a Hilbert space $\mathcal{H}$, 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 $\mathcal{H}$. Kernel functions thus compute the inner product between examples that are mapped in a higher-dimensional feature space. However, they do not necessarily 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*. Most graph kernels decompose graphs into their substructures and then to measure their similarity, they count the number of common substructures. Graph kernels typically focus on some structural aspect of graphs such as random walks, shortest paths, subtrees, cycles, and graphlets.

1. We will first create a very simple graph classification dataset. The dataset will contain two types of graphs: (1) cycle graphs, and (2) path graphs. A cycle graph C_n is a graph on n nodes containing a single cycle through all nodes, while a path graph P_n is a tree with two nodes of degree 1, and all the remaining $n - 2$ nodes of degree 2. Each graph is assigned a class label: label 0 if it is a cycle or label 1 if it is a path. Figure 1 illustrates such a dataset consisting of three cycle graphs and three path graphs. Use the `cycle_graph()` and `path_graph()` functions of NetworkX to generate 100 cycle graphs and 100 path graphs of

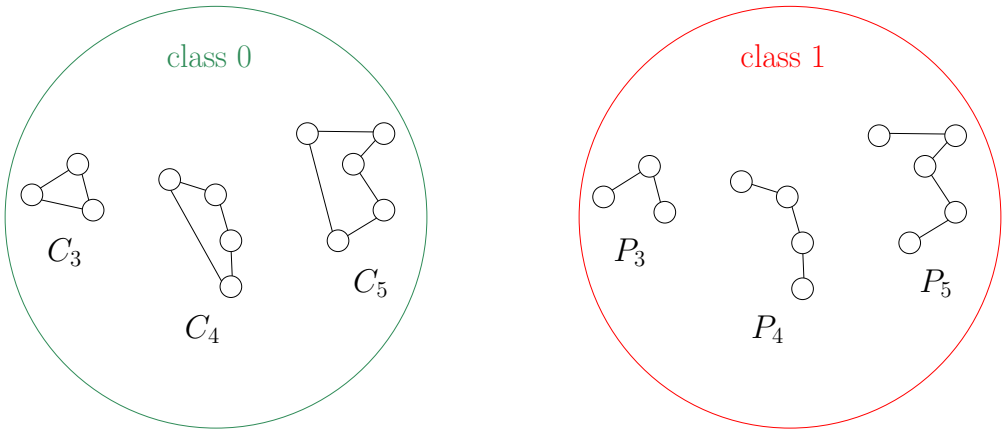

Figure 1: Dataset consisting of two sets of graphs: cycle graphs (left) and path graphs (right).

size $n = 3, \ldots, 102$, respectively. Store the 200 graphs in a list (Gs) and their class labels in another list (y).

2. We will next investigate if graph kernels can distinguish cycle graphs from path graphs. To this end, we will make use of the GraKeL library, a library that provides implementations of several popular graph kernels. Before computing the kernels, it is necessary to split the dataset into a training and a test set. We can use the `train_test_split()` function of scikit-learn as follows:

```
from sklearn.model_selection import train_test_split

G_train, G_test, y_train, y_test = train_test_split(Gs, y, test_size=0.1)
```

Note that before computing the kernels, NetworkX graphs need to be transformed to objects that can be processed by GraKeL. To achive that, you can use the `graph_from_networkx()` function of GraKeL which transforms a list of NetworkX graphs to a list of graph objects that can be handled by GraKeL.

We are interested in generating two matrices. A symmetric matrix $\mathbf{K}_{train}$ which contains the kernel values for all pairs of training graphs, and a second matrix $\mathbf{K}_{test}$ which stores the kernel values between the graphs of the test set and those of the training set. Given a graph kernel, we can obtain these two matrices very easily using the GraKeL library. For example, for the *shortest path kernel*, we can use the following code:

```
from grakel.kernels import ShortestPath

gk = ShortestPath(with_labels=False)

K_train = gk.fit_transform(G_train)
K_test = gk.transform(G_test)
```

After generating the two kernel matrices, we can use the SVM classifier to perform graph classification. More specifically, as shown below, we can directly feed the kernel matrices to the classifier to perform training and make predictions:

```
from sklearn.svm import SVC
```

```
# Initialize SVM and train
clf = SVC(kernel='precomputed')
clf.fit(K_train, y_train)

# Predict
y_pred = clf.predict(K_test)
```

Use the `accuracy_score()` function of scikit-learn to compute the classification accuracy. Then, instead of the shortest path kernel, use the *random walk kernel* and the *pyramid match graph kernel* to perform classification. Compute the classification accuracies. What do you observe? Do all graph kernels perform comparably to each other?

3. Finally, we will use the GraKeL library to classify the graphs of a real-world dataset. The name of the dataset is `MUTAG`, and comes from the field of bioinformatics. It is a binary classification dataset that consists of 188 mutagenic aromatic and heteroaromatic nitro compounds. Note that the graphs contained in the `MUTAG` dataset are node-labeled. Hence, each vertex is assigned a dicrete label from a set of labels. The GraKeL library provides the following function for loading the `MUTAG` dataset (or any other dataset contained in the repository: https://ls11-www.cs.tu-dortmund.de/staff/morris/graphkerneldatasets):

```
from grakel.datasets import fetch_dataset

mutag = fetch_dataset("MUTAG", verbose=False)
G, y = mutag.data, mutag.target
```

Split the dataset into training (90% of samples) and test sets (10% of samples). Then, use the following four graph kernels: (1) *vertex histogram*, (2) *shortest path kernel*, (3) *pyramid match graph kernel*, and (4) *Weisfeiler-Lehman subtree kernel*, to generate the two kernel matrices. Perform graph classification and compare the performance of the four kernels. Since the graphs contain node labels, to take these labels into account, set the `with_labels` argument of the shortest path and pyramid match graph kernels to $True$. To compute the Weisfeiler-Lehman subtree kernel, initialize a `WeisfeilerLehman` object and set its `base_kernel` argument to `VertexHistogram`.

References

Karsten M Borgwardt and Hans-Peter Kriegel. Shortest-path kernels on graphs. In *Proceedings of the 5th IEEE International Conference on Data Mining*, pages 8–pp, 2005.

Aaron Clauset, Mark EJ Newman, and Cristopher Moore. Finding community structure in very large networks. *Physical review E*, 70(6):066111, 2004.

Santo Fortunato. Community detection in graphs. *Physics reports*, 486(3-5):75–174, 2010.

Giannis Nikolentzos, Polykarpos Meladianos, and Michalis Vazirgiannis. Matching node embeddings for graph similarity. In *Proceedings of the 31st AAAI Conference on Artificial Intelligence*, pages 2429–2435, 2017.

Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M Borgwardt. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research*, 12(Sep): 2539–2561, 2011.

Giannis Siglidis, Giannis Nikolentzos, Stratis Limnios, Christos Giatsidis, Konstantinos Skianis, and Michalis Vazirgiannis. Grakel: A graph kernel library in python. *arXiv preprint arXiv:1806.02193*, 2018.

S Vichy N Vishwanathan, Nicol N Schraudolph, Risi Kondor, and Karsten M Borgwardt. Graph kernels. *Journal of Machine Learning Research*, 11(Apr):1201–1242, 2010.

Ulrike Von Luxburg. A tutorial on spectral clustering. *Statistics and computing*, 17(4):395–416, 2007.