Lab session 4: Graph Mining

Lecture: Prof. Michalis Vazirgiannis
Lab: Stratis Limnios and Jesse Read

January 30, 2019

1 Introduction

The goal of this lab is to help you become familiar with methods that operate on graphs and which can be employed for addressing various tasks such as node classification, and graph classification. The lab will also allow you 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. Moreover we will use the GraKeL library [Siglidis et al., 2018] which has available most of the state of the art graph kernels, under a sklearn based framework. We will first implement a neural network that generates node embeddings. These embeddings are unsupervised, in the sense that they can be fed to conventional machine learning algorithms for solving any downstream task. Those embeddings will be evaluated in node classification. Next, we will implement some state-of-the-art graph kernel for graph classification.

2 Node Embeddings

In this part of the lab, we will generate embeddings for the nodes of a graph using a simple baseline method (e.g., spectral embeddings) and a more sophisticated method that is inspired by recent advances in natural language processing and which utilizes a probabilistic neural network. We will then evaluate the generated embeddings on a *multi-label classification* task.

In contrast to binary and multi-class classification, in the multi-label classification setting, there is no constraint on how many of the classes each instance can be assigned to. Hence, given a finite set of classes $\mathcal{C}$, every node can be assigned to one or more classes of the set $\mathcal{C}$. Multi-label classification is a challenging task especially if the set of classes is large.

Our dataset is a protein-protein (PPI) interaction network, that is, a graph that represents the presence or absence of physical interactions between proteins. More specifically, the dataset we will utilize is a subgraph of the PPI network for Homo Sapiens. The subgraph corresponds to the graph induced by nodes for which labels from the hallmark gene sets were available and represent biological states. The network contains $3,890$ nodes, $76,584$ edges, and 50 different labels. The graph is stored in the `Homo_sapiens.mat` file[1].

1. We will first implement a very popular and simple baseline method for generating node embeddings. The method is based on matrix factorization. Specifically, we will use the eigenvectors corresponding to the d smallest eigenvalues of the Laplacian or the normalized Laplacian matrix of the graph to generate feature vector representations for its nodes. Algorithm 1

[1]The graph can be downloaded from the following link: `https://snap.stanford.edu/node2vec/Homo_sapiens.mat`.

illustrates the pseudocode for generating such spectral embeddings.

Algorithm 1 Spectral Embeddings

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

Ensure: Matrix $\mathbf{U} \in \mathbb{R}^{n \times d}$ (i.e., embedding of each node of the graph)

1: Let $\mathbf{A}$ be the adjacency matrix of the graph

2: Compute the Laplacian matrix $\mathbf{L} = \mathbf{D} - \mathbf{A}$ or the normalized Laplacian $\mathbf{L}_{sym} = \mathbf{I} - \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}$. Matrix $\mathbf{D}$ corresponds to the diagonal degree matrix of graph G (i.e., degree of each node in the main diagonal), while matrix $\mathbf{I}$ is the identity matrix.

3: Apply eigenvalue decomposition to the Laplacian matrix $\mathbf{L}$ or the normalized Laplacian $\mathbf{L}_{sym}$ and compute the eigenvectors that correspond to the d smallest eigenvalues. Let $\mathbf{U} = [\mathbf{u}_1|\mathbf{u}_2|\ldots|\mathbf{u}_d] \in \mathbb{R}^{n \times d}$ be the matrix containing these eigenvectors as columns. The i-th row of $\mathbf{U}$ corresponds to the embedding of node v_i.

Next, we will use the above method to generate embeddings for the nodes of the PPI network. Fill in the body of the function `generate_spectral_embeddings()` which can be found in the `spectral_embedding.py` python script as shown below.

```
def generate_spectral_embeddings(A, d):

  # Add the body of the function here

  return U
```

Run the `spectral_embedding.py` python script to generate node embeddings and to create a file that contains these embeddings.

2. We will next implement *DeepWalk*, a more sophisticated algorithm for generating node embeddings [Perozzi et al., 2014]. DeepWalk builds on recent advances in unsupervised feature learning which have proven very successful in natural language processing. DeepWalk learns representations of a graph's vertices by running short random walks. These representations capture neighborhood similarity and community membership. The employed model is analogous to Skipgram: given a vertex v_i, it estimates the likelihood of observing the previous and the following vertices visited in the random walk. Specifically, DeepWalk solves the following optimization problem:

$$\phi \quad -\log \prod_{\substack{j=i-w \\ j \neq i}}^{i+w} P(v_j|\phi(v_i)) \tag{1}$$

where $\phi : V \rightarrow \mathbb{R}^d$ is a function that maps vertices to their embeddings, and w is the size of the sliding window. Hence, given the embedding of a vertex, DeepWalk maximizes the probability of its neighbors in the walk. To make learning efficient both in terms of running time and memory, DeepWalk uses the Hierarchical Softmax to approximate the probability distribution.

Given a graph $G = (V, E)$ and a starting vertex v_i, a random walk of length t is a stochastic process with random variables $w_{v_i}^{(1)}, w_{v_i}^{(2)}, \ldots, w_{v_i}^{(t)}$ such that $w_{v_i}^{(1)} = v_i$ and $w_{v_i}^{(j)}$ is a vertex chosen uniformly at random from the neighbors of vertex $w_{v_i}^{(j-1)}$. In other words, a random walk is a sequence of vertices: we first select a neighbor of v_i at random, and move to this neighbor. Then, we select a neighbor of this new vertex at random, and move to it, etc.

You will implement a function that given a graph and a starting vertex performs a random walk of specific length, and returns a list of the visited vertices. Fill in the body of the function `random_walk()` which can be found in the `deepwalk.py` python script as shown below. Note that G is a NexworkX graph. Use the function `G.neighbors(`v`)` to get the neighbors of a vertex v.

```
def random_walk(G, node, walk_length):

  # Add the body of the function here

  walk = [str(node) for node in walk]
  return walk
```

Next, you will implement a function that given a graph, it runs a number of random walks from each node of the graph, and returns a list of all random walks. Fill in the body of the function `generate_walks()` which can be found in the `deepwalk.py` python script as shown below. Note that the function `G.nodes()` returns a list containing all the nodes of graph G.

```
def generate_walks(G, num_walks, walk_length):

  # Add the body of the function here

  return walks
```

After implementing the two functions, run the `deepwalk.py` python script to learn representations of the vertices of the PPI graph, and to create a file that contains these embeddings.

3. To evaluate the two methods, we will perform multi-label classification using the `evaluate.py` python script that you are given. The script randomly samples 90% of the nodes, and uses them as training data, while the rest of the nodes are used as test data. This process is repeated 5 times, and the micro and macro-average F1-scores of each iteration are displayed. Use the `evaluate.py` script to evaluate the embeddings generated by both methods. What do you observe? Regenerate node embeddings using different values for the hyperparameters (e.g., dimensionality of embeddings, walk length, window size, number of walks per node). Which of these hyperparameters have a large impact on the obtained performance?

3 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

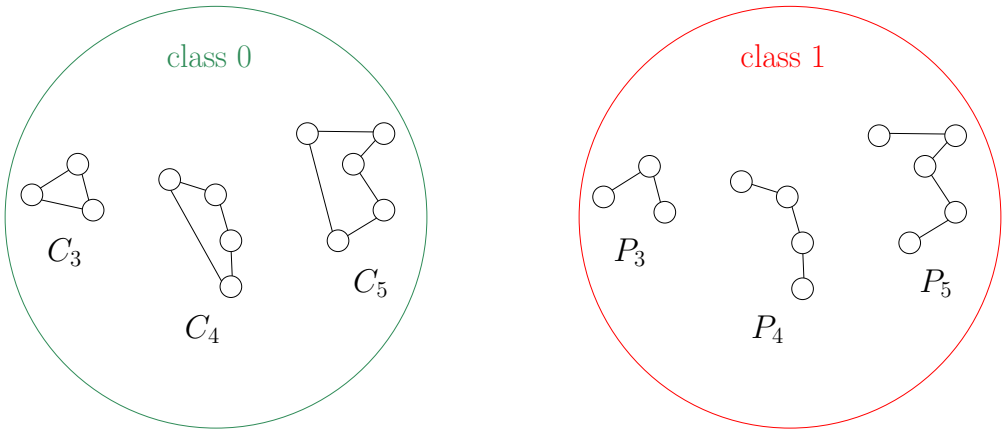

Figure 1: Dataset consisting of two sets of graphs: cycle graphs (left) and path graphs (right).

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 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.

Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online Learning of Social Representations. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 701–710, 2014.

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.