ENS de Lyon, M2 Computer Science
2023/2024 **Pierre Borgnat (CNRS, LP ENSL)**

PRACTICAL SESSION : ML and/for Graphs: Networks, Embeddings, Spectral Clustering

The objective of this tutorial is to go through basic operations for handling networks or graphs, with the objective of testing some elements of network science and then of graph signal processing.

The practical sessions will be done preferentially in python, using the pygsp toolbox:
https://pygsp.readthedocs.io/en/stable/tutorials/intro.html

Note for old school non-computer science students: one could run everything in matlab if you prefer, using either the GSP toolbox:
https://epfl-lts2.github.io/gspbox-html/
or the Matlab / Octave toolbox GraSP: https://gitlab.inria.fr/bgirault/grasp

This practical session was prepared by Nicolas Tremblay (CNRS, now at GIPSA-lab, Grenoble) and Michael Defferrard (formerly at EPFL, now at Qualcomm Research), and I warmly thanks them for that.

The aim is to illustrate various element seen during the lectures. The codes are usually quite short and easy (and anyway, you will find the solutions in the web, even without ChatGPT).

- STEP 0 : SET UP YOUR INSTALLATION -

To an environment having `numpy scipy matplotlib networkx scikit-learn`, you should also add `pygsp` using: `pip install pygsp` for instance. Warning: the library was set up with python 3.6 ; it should work with more recent versions, we'll see...

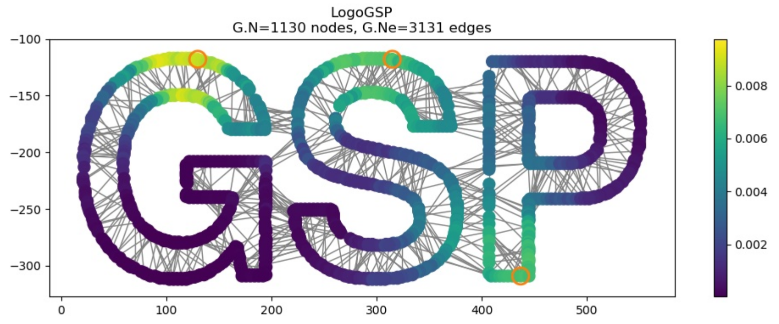
If you have set up correctly your environment, you should run the following code;

```
import numpy as np
import scipy
import pygsp
from matplotlib import pyplot as plt
import sklearn
import networkx as nx

W = scipy.sparse.rand(10, 10, 0.2)
W = W.toarray()
W = W - np.diag(np.diag(W))
W = W + np.transpose(W)
G = pygsp.graphs.Graph(W)
degree_vector = G.d
print(degree_vector)

G = pygsp.graphs.Logo()
G.estimate_lmax()
g = pygsp.filters.Heat(G, tau=100)
DELTAS = [20, 30, 1090]
s = np.zeros(G.N)
s[DELTAS] = 1
s = g.filter(s)
G.plot_signal(s, highlight=DELTAS, backend='matplotlib')
```

And the final result should be:



- Step 1: Graphs: models, properties, visualization -

For that, we will use: `from pygsp import graphs`

1) **Creation:**
The first thing you would like to do is to create a graph. In this PyGSP, a graph is encoded as a weighted adjacency matrix. That is because it is the most efficient representation to deal with when using spectral methods. As such, you can construct a graph from any adjacency matrix as follows.

```
W = np.random.uniform(size=(30, 30)) # Full graph.
W[W < 0.93] = 0 # Sparse graph.
W = W + W.T # Symmetric graph.
np.fill_diagonal(W, 0) # No self-loops.

G = graphs.Graph(W)

print(f'{G.N} nodes, {G.Ne} edges')
```

Graphs can also be created from network models, such as the well-known Erdös-Rényi, or a network with communities:

```
G = graphs.ErdosRenyi(50, p=0.2)
G = graphs.Community(N=100, Nc=5)
```

Or from a set of data points having features in $\mathbb{R}^d$:

```
n_points = 100
dimensionality = 2
points = np.random.uniform(size=(n_points, dimensionality))
plt.scatter(points[:, 0], points[:, 1]);
```

As we have seen, there are many ways to create a graph from such data: k-NN graph, ϵ-radius neighbourhood graph. One can also weight the edges with some kernel on the distance (e.g. a Gaussian one). Check the lines:

```
G = graphs.NNGraph(points, NNtype='knn', k=2, sigma = sigma**2,
rescale=False, symmetrize_type='maximum')
print(f'{G.N} nodes, {G.Ne} edges')

G = graphs.NNGraph(points, NNtype='radius', epsilon=0.1, sigma = sigma**2,
rescale=False, symmetrize_type='maximum')
print(f'{G.N} nodes, {G.Ne} edges')
```

2) **Visualization:**

To be able to plot a graph, we need to embed its nodes in a 2D or 3D space. The process to find coordinates for each nodes is called layout. Most included graph models define these. If that's not the case, or if you create the graph from an adjacency matrix, `pygsp.graphs.Graph.set_coordinates()` can be used to set it.

In the last example, the nodes had a natural 2D embedding. It is thus logical to plot it using the 2D coordinates of each point in the point cloud:

```
G.plot()
```

Nevertheless, the same graph could be drawn in many different ways. Below are two different options:

```
G2 = graphs.Graph(G.W)
G2.set_coordinates(kind='ring2D') #here with a spring based algorithm
G2.plot()
```

```
G2 = graphs.Graph(G.W)
G2.set_coordinates(kind='spring') #here with a spring based algorithm
G2.plot()
```

For a more interactive visualization, you can use Gephi or Cytoscape.

3) **Properties:**
The `pygsp.graphs.Graph` class we just instantiated is the base class for all graph objects, which offers many methods and attributes. Given a graph object, we can access and test some properties. As we've used before, there is the number of nodes and edges. We can also test if the graph is connected or directed.

```
print(G.N)
print(G.Ne)
print(G.is_connected())
print(G.is_directed())
```

You can retrieve the adjacency matrix `G.W` and plot it, or access also to the graph as a list.

```
v_in, v_out, weights = G.get_edge_list()
print(v_in.shape, v_out.shape, weights.shape)
```

You can also compute the graph Laplacian(s):

```
G.compute_laplacian('combinatorial')
#or G.compute_laplacian('normalized')
##the normalized Lap may yield an error if there exist a node with degree 0!
G.L
plt.spy(G.L, markersize=5);
```

4) **Exercise: The Barabasi-Albert model**

Create a graph from the Barabási-Albert model.
Compute and plot its degree distributions.
Plot the graph in 2D with a force-directed layout.

5) **Exercise: connectedness phase transition in Erdös-Rényi graphs**

Erdös-Rényi graphs have a "connectedness phase transition" at $p = \frac{\log N}{N}$. That is, as $N \to \infty$:

- if $p < \frac{\log N}{N}$ the graph is a.s. disconnected
- if $p > \frac{\log N}{N}$ the graph is a.s. connected

Illustrate this phase transition computationally

- Step 2: Spectral Graph Theory: spectral clustering, Laplacian eigenmaps -

We'll see here two of the most celebrated algorithms stemming from Spectral Graph Theory:

- Spectral clustering: clustering data using a similarity graphs
- Laplacian eigenmaps: dimensionality reduction which preserves geodesic distances

We will need:

```python
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
from pygsp import graphs
from additional_utils import AR_index, generate_concentric_circles
import networkx as nx
import sklearn
from sklearn import datasets, decomposition
from sklearn.cluster import KMeans
from mpl_toolkits.mplot3d import Axes3D
```

1) **The spectrum of a graph**

We consider here only undirected graphs, such that the Laplacian matrix is real symmetric, thus diagonalizable in an orthonormal eigenbasis: $\mathbf{L} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^\top$.

Try the following codes:

```python
G = graphs.Community(N=100, Nc=5)
G.plot()
eig_val, U = sp.linalg.eigh(G.L.toarray())
# you obtain the same thing than if you had done G.compute_fourier_basis()
# and looked at G.e and G.U
```

Look at `eig_val`. Look at `np.finfo(float).resolution`. How many connected components are in the graph? Check with the networkx function `nx.connected_components`:

```python
G_nx = nx.from_numpy_matrix(G.A.toarray())
sorted(nx.connected_components(G_nx), key = len, reverse=True)
```

Note that we used `sp.linalg.eigh` to diagonalize $\mathbf{L}$. because it is the eigensolver optimized for symmetric matrices. Also, `sp.linalg.eigh` cannot take sparse matrices as entries. For sparse matrices, optimized eigensolvers exist to obtain the first few eigenvalues and eigenvectors, such as

```python
eig_val, U = sp.sparse.linalg.eigsh(G.L, k=10, which='SM')
```

Use option `which='SM'` for smallest magnitude search, and `which='LM'` for largest magnitude search. `eigsh` is nevertheless only made to compute only a few eigenvectors. You will quickly see the utility of `sp.sparse.linalg.eigsh(G.L, k=10, which='SM')` versus `sp.linalg.eigh(G.L.toarray())` if you increase the size of the network to 10^4 nodes or larger for instance.

2) The Stochastic Block Model (SBM)

The Stochastic Block Model (SBM) is a latent variable model of structured random graphs.

- The set of N nodes is partitioned in k blocks $B_1, \dots, B_k$: $\cup_i B_i = \{1, \dots, N\}$ and $\forall i \neq j, B_i \cap B_j = \emptyset$.
- We define a probability matrix $\mathbf{P} \in \mathbb{R}^{k \times k}$ such that P_{ij} is the probability of connection between a node in B_i and a node in B_j.
- To generate an instance of SBM: for each pair of nodes in the graph, draw a Bernoulli random variable of parameter corresponding to their respective blocks.

To simplify, we will suppose that all intra-block probabilities are set to p_{in} and all inter-block probabilities are set to p_{out}. Also, we will consider only balanced SBMs (ie SBMs whose blocks have same size).

In the case of balanced SBMs, instead of the pair of parameters $(p_{\text{in}}, p_{\text{out}})$, we can consider the pair (c, ϵ) where c is the average degree of the graph and $\epsilon = \frac{p_{\text{out}}}{p_{\text{in}}}$ is a measure of fuzziness: the larger ϵ, the fuzzier the community structure. The relations between both pairs of parameters are:

$$p_{\text{in}} = \frac{kc}{N - k + (k-1)\epsilon N} \quad \text{and} \quad p_{\text{out}} = \frac{kc\epsilon}{N - k + (k-1)\epsilon N}$$

Note that, for a fixed triplet (ϵ, N, k), c is not a free parameter.

The classical inference SBM problem is: given an instance of an SBM G, infer the blocks.

Impossibility threshold: In the balanced (ie: all blocks have the same size) and sparse (ie average degree $c = \mathcal{O}(1)$) case, the following phase transition has been shown ([Decelle et al.](https://arxiv.org/pdf/1109.3041.pdf), [Massoulié](https://arxiv.org/pdf/1311.3085.pdf), [Abbe et al.](https://arxiv.org/pdf/1503.00609.pdf)). For a given pair (k, c), there exists a critical fuzziness ϵ_c such that as $N \to \infty$:

- if $\epsilon > \epsilon_c$, there exists no algorithm capable of extracting any structural information.
- if $\epsilon < \epsilon_c$, it is theoretically possible to retrieve some kind of structural information

with $\epsilon_c = \frac{c - \sqrt{c}}{c + \sqrt{c}(k-1)}$

When `epsi = epsi_c / 40`, ie, a very strong block structure, blocks appear even 'visually' with spring-based plotting algorithms:

```
N = 500; # number of nodes
k = 4; # number of blocks
c = 16; # average degree

epsi_c = (c - np.sqrt(c)) / (c + np.sqrt(c) * (k-1)); # critical fuzziness
epsi = epsi_c / 40; # this is a very strong block structure

pin = (k * c) / (N - k + (k - 1) * epsi * N)
pout = (k * c * epsi) / (N - k + (k - 1) * epsi * N)

z = np.tile(np.arange(k), (np.int(N/k),1))
truth = np.reshape(z.T, (N,1))[:,0]
# the ground truth community assignment truth[i] is the community number of node i

G = graphs.StochasticBlockModel(N=N, k=k, z=truth, p=pin, q=pout)

G.set_coordinates(kind='spring') #here with a spring based algorithm
G.plot()
```

3) **The Eigengap**

Let's look at the first few eigenvalues. The large gap between λ_k and λ_{k+1} is called the "spectral gap": it heuristically appears when there is a strong structure in k communities (even though there exist some theoretical justifications).

Exercice:

- Plot a SBM graph with the spring-based plotting algorithm with $\epsilon = \epsilon_c/3$ for instance: one can no longer extract communities 'visually' !
- Let's formally define the normalized spectral gap as $\Gamma = \frac{\lambda_{k+1}-\lambda_k}{\lambda_k}$. Plot the average of Γ as a function of the fuziness ϵ in the case of balanced SBM graphs.

4) **The first few eigenvectors**
We have seen that an eigengap usually exists in community-structured graphs. What can we say about the first few eigenvectors? Let's plot them in a balanced SBM graph:

```
N = 500; # number of nodes
k = 4; # number of blocks
c = 16; # average degree

epsi_c = (c - np.sqrt(c)) / (c + np.sqrt(c) * (k-1)); # critical fuzziness
epsi = epsi_c / 40; # this is a very strong block structure

pin = (k * c) / (N - k + (k - 1) * epsi * N)
pout = (k * c * epsi) / (N - k + (k - 1) * epsi * N)

z = np.tile(np.arange(k), (np.int(N/k),1))
truth = np.reshape(z.T, (N,1))[:,0] # the ground truth community assignment

G = graphs.StochasticBlockModel(N=N, k=k, z=truth, p=pin, q=pout)

eig_val, U = sp.sparse.linalg.eigsh(G.L, k=10, which='SM')
G.set_coordinates(kind='spring')

G.plot_signal(U[:,0])
G.plot_signal(U[:,1])
G.plot_signal(U[:,2])
G.plot_signal(U[:,3])
G.plot_signal(U[:,4])
```

Say we want to recover in which block belongs each node. The first eigenvector's information is useless (as long as we use the combinatorial Laplacian anyways) as it is constant. A solution is to plot each node i in 2D with coordinates $(u_2(i), u_3(i))$.

Look at `plt.scatter(U[:,1], U[:,2], c=G.truth)`: the nodes in the same block are in the same 'blob' in this 2D plot! This observation is the basis of spectral-clustering-type algorithms.

(Bonus question) Why is the 5th eigenvector so localized?

5) **Exercise: spectral clustering**

- Show that, in the last example, performing k-means in the $2D$ plane defined by $\mathbf{u}_2$ and $\mathbf{u}_3$ recovers the 3 block structures (the ground truth is in `G.truth`). For k-means, use the function `KMeans` in the toolbox `sklearn.cluster`. To measure performance, you can use the function `AR_index` which computes the Adjusted Rand index (both functions were already imported in the preamble).
- Given the intuition gained, write a general spectral clustering algorithm that takes as entry a graph G, a number of communities k, and outputs its community structure in k blocks.

- Plot the average performance of community mining with this spectral clustering algorithm on balanced SBMs versus the fuziness ϵ.

To go further, you can read ["A tutorial on spectral clustering" by U. von Luxburg](https://arxiv.org/pdf/0711.0189.pdf). You will learn among other things that a degree-corrected version is in general preferable (using the normalized Laplacian instead of the combinatorial Lap for instance). For arguments to choose the "best" degree-correction, see ['Improved spectral community detection in large heterogeneous networks' by Tiomoko Ali and Couillet](http://www.jmlr.org/papers/volume18/17-247/17-247.pdf).

6) **Creating a graph**

We have seen the spectral clustering algorithm with inputs a graph G and a number of desired clusters k. In many cases, however, the input data is simply N vectors in dimension d. The construction of the graph is a problem in itself! Let us look at a toy point cloud example where spectral clustering is useful: two concentric circles. k-means will obviously not work on this example.

```
N_in=300 # number of nodes inside
N_out=500 # number of nodes outside
sigma_in= 0.15
sigma_out= 0.15
data, truth = generate_concentric_circles(N_in, N_out, sigma_in, sigma_out, d=2)

plt.scatter(data[:,0], data[:,1], c=truth, cmap=plt.cm.Spectral)
```

Exercise: which similarity graph should one create from the data? Recall the two main ways of creating a graph: k-nearest neighbours or ϵ-proximity.

- What is one of the main differences between the graphs created by these two techniques? (regarding the degree distribution for instance)
- How does the performance change vs the choice of k (the number of neighbours in the kNN graph)?
- How does the perf change vs the choice of ϵ (in the ϵ-graph)?

7) **Laplacian eigenmaps**

Consider the data

$$\mathbf{X} = (\mathbf{x}_1|\mathbf{x}_2|\dots|\mathbf{x}_N)^\top \in \mathbb{R}^{N\times d},$$

where each data point $\mathbf{x}_i$ lives in dimension $\mathbb{R}^d$.

Laplacian eigenmaps are a non-linear dimension reduction technique that relies on the assumption that the data lives in a low-dimensional manifold of $\mathbb{R}^d$. Different from PCA applied directly on $\mathbf{X}$, Laplacian eigenmaps take into account the intrinsic geometry of the data by first building a similarity graph and then looking at the first few eigenvectors of its Laplacian matrix. One possible formulation of the problem is the following.

Problem. Consider an embedding $f : \mathcal{V} \to \mathbb{R}^m$ and denote by $\mathbf{y}_i = f(i) \in \mathbb{R}^m$ the coordinates of node i in the embedding space. The goal is to find an embedding such that points close to each other on the manifold are also close to each other in the embedded space, thus: an embedding preserving local distances on the manifold.

A solution proposed by [Belkin and Niyogi](http://www.cs.rochester.edu/ stefanko/Teaching/09CS446/Laplacian.pdf) for an embedding in dimension m is to:

i) first create a similarity graph;

ii) compute the first $m+1$ right-eigenvectors of its random walk Laplacian matrix $(\mathbf{u}_0|\mathbf{u}_1|\dots|\mathbf{u}_m)$. The random walk Laplacian matrix is defined as

$$\mathbf{L}_{rw} = \mathbf{D}^{-1}\mathbf{L} = \mathbf{I} - \mathbf{D}^{-1}\mathbf{W}.$$

One can show by the way that the normalized Laplacian $\mathbf{L}_n = \mathbf{I} - \mathbf{D}^{-1/2}\mathbf{W}\mathbf{D}^{-1/2}$ and the random walk Laplacian share the same eigenvalues and that $\mathbf{L}_n$'s eigenvectors and $\mathbf{L}_{rw}$'s right eigenvectors are connected via a simple transformation $\mathbf{u}_{rw} = \mathbf{D}^{-1/2}\mathbf{u}_n$.

iii) Embed each node i via $\mathbf{y}_i = (u_1(i)|u_2(i)|\dots|u_m(i))^\top \in \mathbb{R}^m$. In fact, $\mathbf{u_0}$ is constant thus useless for the embedding.

Informal justification. Let us write

$$\mathbf{Y} = (\mathbf{y}_1^\top|\dots|\mathbf{y}_N^\top)^\top \in \mathbb{R}^{N\times m}.$$

Once $\mathbf{W}$ is created, the idea is to find an embedding such that two points are close to each other in the embedding space if they are connected and far from each other if not. If $m = 1$ this means finding $\mathbf{Y}$ such that

$$\sum_{i\sim j} \mathbf{W}_{ij}(\mathbf{Y}_{i1} - \mathbf{Y}_{j1})^2 = \mathbf{Y}^\top \mathbf{L}\mathbf{Y}$$

is minimized. The obvious solution to this problem is the constant vector (associated to eigenvalue 0). To avoid such trivial solution, we enforce $\mathbf{Y}$ to be orthogonal to $\mathbf{1}$. The solution is thus $\mathbf{Y} = \mathbf{u}_2$.

Two toy graphs.

```
# Swiss Roll

Axes3D

n_points = 1000
X, color = datasets.make_swiss_roll(n_points, random_state=0)

fig = plt.figure(figsize=(4, 4))
ax = fig.add_subplot(111, projection='3d')
ax.scatter(X[:, 0], X[:, 1], X[:, 2], c=color, cmap=plt.cm.Spectral)
ax.view_init(4, -72)

# two concentric spheres in 3D

X, truth = generate_concentric_circles(N_in, N_out, sigma_in, sigma_out, d=3)

Axes3D

fig = plt.figure(figsize=(4, 4))
ax = fig.add_subplot(111, projection='3d')
ax.scatter(X[:,0], X[:,1], X[:,2], c=truth, cmap=plt.cm.Spectral)
ax.view_init(40, -72)
```

Exercise: PCA vs Laplacian eigenmaps

- Write the Laplacian eigenmaps algorithm and (at least visually) compare its embedding performance on the swiss roll dataset vs PCA embedding (both embeddings are in 2D). For PCA, use: `decomposition.PCA`. To compute the first few eigenvectors of the random walk Laplacian, it is more efficient (and stable) to compute the first few eigenvectors of the normalized Laplacian via eigsh (because it is nicely symmetric), and then use $\mathbf{u}_{rw} = \mathbf{D}^{-1/2}\mathbf{u}_n$.
- Do the same with concentric data in 3D:
  `data, truth = generate_concentric_circles(N_in, N_out, sigma_in, sigma_out, d=3)`

8) **Bonus Exercice: Laplacian Eigenmaps on real data**

Let's use the famous Digit dataset:

```
digits = datasets.load_digits(n_class=6)
X = digits.data
y = digits.target
n_samples, n_features = X.shape
n_neighbors = 30

print(f"{n_samples} digits, {n_features} features")
```

Look at it:

```
# Plot images of the digits
n_img_per_row = 20
img = np.zeros((10 * n_img_per_row, 10 * n_img_per_row))
for i in range(n_img_per_row):
    ix = 10 * i + 1
    for j in range(n_img_per_row):
        iy = 10 * j + 1
        img[ix:ix + 8, iy:iy + 8] = X[i * n_img_per_row + j].reshape((8, 8))

plt.figure(figsize=(8,8))
plt.imshow(img, cmap=plt.cm.binary)
plt.xticks([])
plt.yticks([])
plt.title('A selection from the 64-dimensional digits dataset');
```

Create a graph, display it, find clusters....

- Step 3: More analyses of Graph Signals with Graph Fourier Transform -

The truth is that this practical session was prepared by Michaël Defferrard and Nicolas Tremblay for a research summer school a few years back.
You can find this tutorial, and more, online:
https://github.com/mdeff/pygsp_tutorial_graphsip

If you were quick, you can continue with parts 3 and 4:

- **3. Graph signals: gradient, divergence, smoothness**
- **4. Fourier: modes, transform**

(and more if you like).
I hope you enjoyed your first steps in GSP!