Graph Machine Learning - Lab 1
Graph Signal Processing

Xiaowen Dong (xdong@robots.ox.ac.uk)

In this session[1], we will be looking into graph signal processing, a fast-growing field that deals with signals that are defined on the vertex set of weighted graphs. We will be using **PyGSP**, a Python package for graph signal processing operations. You can install **PyGSP** with **pip** or **conda**. Alternatively, if you prefer MATLAB, you can make use of **GSPBOX**, a counterpart of **PyGSP** developed by the same team that has the same functionality. The sample code provided in this lab notes would need to be adapted accordingly.

- **PyGSP**: `https://github.com/epfl-lts2/pygsp`
- **GSPBOX**: `https://github.com/epfl-lts2/gspbox`

Now let us get started. First we import the necessary Python modules.

```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
from pygsp import graphs, filters, plotting
```

Graph. A graph consists of a set of nodes $\mathcal{V}$, and a set of edges $\mathcal{E}$ with associated edge weights. It can be encoded in the adjacency matrix $\mathbf{W} \in \mathbb{R}^{N \times N}$ where $|\mathcal{V}| = N$. Graphs can be created using the *graphs* module in **PyGSP** (`https://pygsp.readthedocs.io/en/stable/reference/graphs.html`).

We can define a graph as a real world network, such as the road network in Minnesota, USA.

```
G = graphs.Minnesota()
G.coords.shape % coordinates are already set for nodes for visualisation
fig, axes = plt.subplots(1, 2)
_ = axes[0].spy(G.W, markersize=0.5) % visualise the adjacency matrix in a spy plot
G.plot(ax=axes[1]) % visualise the graph in 2D coordinates
```

Alternatively, we can define a graph as an instance of a random graph model, e.g., an Erdős-Rényi graph.

```
G = graphs.ErdosRenyi(N=30, p=0.2)
G.set_coordinates(kind='spring') % set coordinates for nodes for visualisation
fig, axes = plt.subplots(1, 2)
_ = axes[0].spy(G.W, markersize=2)
G.plot(ax=axes[1])
```

[1]This lab session is largely inspired by the exercises built by Michaél Defferrard, Effrosyni Simou, and Hermina Petric Maretić for the course *A Network Tour of Data Science* at EPFL and that by Michaél Defferrard and Nicolas Tremblay for the GraphSiP Summer School in 2018.

Graph Laplacian. The unnormalised (combinatorial) graph Laplacian matrix is defined as:

$$\mathbf{L} = \mathbf{D} - \mathbf{W},$$

where $\mathbf{D}$ is the matrix that contains the degree of the nodes along diagonal.

Let us compute and visualise the graph Laplacian matrix $\mathbf{L}$.

```
G.compute_laplacian('combinatorial')
fig, axes = plt.subplots(1, 2)
axes[0].spy(G.L, markersize=5)
axes[1].hist(G.L.data, bins=50, log=True);
```

A spy plot is a visualisation of a matrix in which there is a dot for each non-zero entry in the matrix.

- Can you observe the difference between $\mathbf{W}$ and $\mathbf{L}$ from their spy plots?

Graph signal. A graph signal $\mathbf{f} : \mathcal{V} \to \mathbb{R}^N$ assigns a scalar value to each node of the graph. Let us plot a random graph signal on a ring graph.

```
G = graphs.Ring(N=60)
f = np.random.normal(size=G.N)
G.plot_signal(f, vertex_size=50)
```

The smoothness of a signal $\mathbf{f}$ on the graph can be evaluated by the Laplacian quadratic form:

$$\mathbf{f}^T \mathbf{L} \mathbf{f} = \frac{1}{2} \sum_{i,j=1}^{N} \mathbf{W}_{ij} (\mathbf{f}(i) - \mathbf{f}(j))^2,$$

i.e., the smaller the quantity of $\mathbf{f}^T \mathbf{L} \mathbf{f}$, the smoother the signal on the graph (less variation of signal values across edges).

- Compute the Laplacian quadratic form for $\mathbf{f}$. Now define a new signal $\mathbf{g}$ which is the same as $\mathbf{f}$ but with values sorted in increasing order from the first to the last entry. Compute the Laplacian quadratic form for $\mathbf{g}$. How does it compare to that for $\mathbf{f}$, and why?

Graph Fourier transform (GFT). The eigenvectors of the graph Laplacian matrix provides a Fourier-like basis for signals on graphs. Plot the first 5 eigenvectors of the 2D grid graph and the Minnesota road network. (Note: The Minnesota road network consists of many vertices, so it is better to set vertex_size $= 5$.)

```
G = graphs.Grid2d(10, 10)
G = graphs.Minnesota()
```

- Compute the Laplacian quadratic form for the first 5 eigenvectors of the 2D grid graph. How do the quantities vary, and why?

Similar to the classical Fourier transform, the GFT transforms the graph signal from the vertex domain into the graph spectral domain. Let us examine the Fourier coefficients of a random signal $\mathbf{f}$ on a community graph, i.e., a graph in which nodes form a number of communities. (Note: You can change the connectivity pattern of the graph by adjusting the parameters comm_density and world_density.)

```
communities = [40, 80, 60]
G = graphs.Community(N=180, Nc=3, comm_sizes=communities, comm_density=0.2, world_density=0.01)
f = np.random.normal(size=G.N)
G.compute_fourier_basis()
f_hat = G.gft(f)

fig, axes = plt.subplots(1, 2, figsize=(10, 4))
G.plot_signal(f, vertex_size=30, ax=axes[0])
axes[1].plot(G.e, np.abs(f_hat), '.-')
```

- From its Fourier coefficients, can you comment on the smoothness of $\mathbf{f}$ on the graph?

Now let us consider another signal, i.e., the partition function:

$$\mathbf{f}_{\text{part}}(i) = \begin{cases} -1, & \text{if } i \in C_1, \\ 0, & \text{if } i \in C_2, \\ 1, & \text{if } i \in C_3, \end{cases}$$

where C_1, C_2, and C_3 denote the three communities.

```
fp = np.zeros(G.N)
fp[:communities[0]] = -1 * np.ones(communities[0])
fp[-communities[-1]:] = 1 * np.ones(communities[-1])
```

- Compute the Fourier coefficients of $\mathbf{f}_{\text{part}}$ and compare with that of $\mathbf{f}$. What do you observe, and why?

[Checkpoint 1] You are now done with the first part of the exercises. Please ask a lab demonstrator to evaluate your work in this part.

Filtering. Just like in classical signal processing, we can also apply filtering to graph signals. The main idea is to transform the graph signal into the graph spectral domain, attenuate unwanted frequencies or amplify desired frequencies of the signal by modifying the corresponding Fourier coefficients, and convert the signal back to the vertex domain. This modification is done via a filter function $g(\lambda)$ in the graph spectral domain, where λ is the eigenvalue of the graph Laplacian. Let us start with the following low-pass filtering example, using the random signal $\mathbf{f}$ and the community graph defined before.

```
import copy
fb_hat = copy.deepcopy(f_hat)
fb_hat[10:] = 0 % this corresponds to an ideal brick-wall low-pass filtering
fb = G.igft(fb_hat)
```

- Plot both the original signal $\mathbf{f}$ and the filtered signal $\mathbf{f}_{\text{brick}}$ on the graph. What do you observe?

We can define our own filters by designing the function $g(\lambda)$. For example, the following filter

$$g(\lambda) = \frac{1}{1+\lambda}$$

can be defined as follows.

```
g = filters.Filter(G, lambda x: 1. / (1. + x))
```

As discussed in the lecture, filtering of graph signals is equivalent to convolution on graphs, which leads to the development of convolutional neural networks on graphs. We will look into these advanced machine learning models in Lab 2. In the meantime, many practical signal processing tasks can be formulated as a filtering problem. One such example is denoising.

Denoising. Denoising is a classical signal processing problem where we wish to remove noise from the observed signal. Denoising for a graph signal can be formulated as the following optimisation problem:

$$\min_{\mathbf{x} \in \mathbb{R}^N} ||\mathbf{x} - \mathbf{y}||_2^2 + \alpha\, \mathbf{x}^T \mathbf{L} \mathbf{x}, \tag{1}$$

where $\mathbf{y}$ is the observed noisy graph signal. The optimisation problem in Eq. (1) tries to find an $\mathbf{x}$ that is close to $\mathbf{y}$, and at the same time being smooth on the graph. The regularisation parameter α controls the trade-off between the data fidelity term and the smoothness prior. The problem in Eq. (1) has the following closed-form solution:

$$\mathbf{x} = (\mathbf{I} + \alpha\, \mathbf{L})^{-1} \mathbf{y}. \tag{2}$$

This solution can be interpreted as applying the following low-pass filter to the observed signal $\mathbf{y}$:

$$g(\lambda) = \frac{1}{1 + \alpha\,\lambda}$$

- Implement the solution to Eq. (1) via the following steps:
  1. Construct a sensor network using the class *pygsp.graphs.Sensor* in **PyGSP**, and compute the graph Laplacian;
  2. Construct a smooth signal on the graph, e.g., a linear combination of the first few eigenvectors of the Laplacian;
  3. Add Gaussian random noise (with appropriate variance) to the smooth signal, to obtain the noisy signal;
  4. Define the low-pass filter with a suitable value of α, and apply it to the noisy signal, to obtain the denoised signal; compare the clean, noisy, and denoised signals on the graph.
  5. Repeat with a different α. What is the impact of the value of α on the denoised signal?

Now you can apply this low-pass filtering technique to denoise your own graph signal observation, e.g., estimated user preference or sentiment as a noisy signal on a social network.

[Checkpoint 2] You are now done with the second part of the exercises. Please ask a lab demonstrator to evaluate your work in this part.