Experimenting with random-network generators

1. Comparing networks and their randomized versions.
   (a) Using networkx, load the airport dataset.

   It can be useful to work on a smaller network for experimenting (you can go back to the original graph when your functions are ready). To work on a subnetwork composed of the nodes of largest degree, you can do for instance:

   Listing 1: Filtering out low degree nodes

   ```
   to_keep = [n for n,v in g.degree() if v > 10]
   g_small = g.subgraph(to_keep)
   ```

   (b) Generate an ER random version of it. You can use `gnp_random_graph` and `gnm_random_graph` methods.

   (c) Generate also a configuration model (degree preserving) version of it, using `expected_degree_graph`, and the degrees observed for the real network (e.g., with `g.degree`)

   (d) Compare the network properties of the 3 different versions of the graph, in particular the average degree, clustering coefficient(`transitiviy`), average path length. Interpret in terms of small-worldness.

   (e) To have a statistically robust approach, generate 100 versions of the configuration model/gnp model, and compute their clustering coefficient. Plot the distribution of values obtained. Compare with the value on the real network to confirm that the observed transitivity is significantly higher (Bootstrapping confidence).

2. Experimenting with PageRank
   (a) Let's investigate a question relating to node centralities and random networks. We know that nodes with higher degrees tend to have a higher PageRank, there is a strong correlation between the two. We would like to know if the PageRank scores we observed are explained only be the degree of nodes, or if there is something more to it.

   (b) Compute and plot the distribution of PageRank scores in the original graph. You can use `seaborn` library and `displot` function, which simply requires a list of values. You can plot it with a log scale on `y` by adding the line `plt.yscale("log")` just after your seaborn command.

   (c) Compare with the distribution of PageRank in the gnp graph first. You should observe a completely different distribution. Make the relation with the degree distribution.

   (d) Compare now with the distribution of Pagerank in the configuration model. The distribution should be more similar to the real one. Run it 100 times, memorizing all Pagerank values observed, and plot the resulting distribution.

   (e) We could use a statistical test to check if both samples come from a same distribution, but let's focus on the highest degree nodes. Re-run 100 version of the configuration model, memorizing the highest value encountered. Plot the distribution of those highest values and compare with the highest value on the original network. What do you conclude ? Can you make an hypothesis on what is happening ?

3. Spatial graph

(a) We want to capture the spatial network aspect of our airport dataset. Extract first the coordinates of each node in a dictionary, for instance with
`coordinates = {n:(g.nodes[n]["lat"],g.nodes[n]["lon"]) for n in g.nodes}`

(b) For each edge in the network, compute the distance between its two endpoints. The *great circle distance*, i.e., the distance following the curvature of the earth, can be computed using function `haversine` from package `haversine`. In a google colab notebook, you can install it with
`!pip install haversine`

(c) Plot the distribution of edge distances using `sns.displot`. You can use the `binwidth` argument to make your graph more clear.

(d) Compute the distribution of distances between all pairs of nodes in the network. You can use for instance `itertools.combinations(LIST,r=2)`

(e) Plot the distribution. Observe that it is far from a bell curve, because it is driven by the position of the nodes, which is not random.

(f) To compute the actual influence of distance, you need to take into account both factors. A simple solution consists in using bins (e.g., 500km), and for each bin, to divide the number of observed links between nodes at this distance by the number of node pairs located at this distance. It can be interpreted as a notion of density, it is the probability to encounter an edge in a pair of nodes taken at random at a given distance. A simple way to do it is to use the `np.histogram(edge_distances, bins)` function to count the number of items in each bin.

(g) Plot the ad-hoc deterrence function computed that way.

(h) The next step would be to generate a random spatial graph by keeping the position of nodes and the deterrence function, and to study how much of the airport dataset properties is explained by it. Since it is a bit complex to code, just write the algorithm in natural language, and compare with your peers. If you want to try implementing it, you can use the `IntervalDict` function from package `python_intervals`. Remember that this will not preserve node degrees. If you want to preserve both, you need to use a *gravity random network model*.

4. Resilience

> When analyzing networks of infrastructure, transport or telecommunication, the *resilience* or *robustness* of the network is a popular investigation topic. Resilience can be defined as the capacity of the network to resist attacks and/or failures. The resistance can be measured in several ways, and attacks and failures can be modeled differently too. We will briefly investigate this question on the airport dataset.

(a) Let's define a simple measure of how *efficient* the network is as the fraction of nodes belonging to the largest connected component. Write a function computing this score, given a network.

(b) Write a function which, given an ordered list of nodes, remove them one by one from the network, compute the resilience measure at each step, and return the result.

(c) First, let's compute the resistance of the network to *failures*, i.e., random node removals. Generate a vector containing all nodes in a random order, and call the function defined above on it. Plot the results, i.e., resilience as a function of nodes removed. Repeat a few times to see how much it depends on the order of nodes.

(d) Let's now test the resilience to *attacks*, i.e., we will remove nodes in a specific order, and search for the most damaging strategy. Compare attacks based on the PageRank and the Betweeness.

(e) Compare the same process (Random attacks, attacks targeted on PageRank/Betweeness on gnp and configuration versions of the airport dataset.

(f) Try to interpret the resilience in term of the classic network descriptions, i.e., degree distribution, clustering...

(Going further exercises are provided as additional exercises you can do out of curiosity, you are not expected to do them as part of the training)

5. Going further : Generating Scale-Free networks with Preferential Attachment.

> Networkx has a function to create networks following the preferential attachment principle (`barabasi_albert_graph`), but we would like to study the dynamic of the model, so we will code our own version.

(a) Using networkx, generate an initial random ER network composed of a small number of nodes

(b) Write a `for` loop, such as each iteration adds a new node to the network, with a small number of edges, each of them connected to existing nodes with a probability proportional to their degree (*preferential attachment*). You can use, for instance, the method `random.choices`

(c) Plot the degree distribution, with and without a **log-log** scale.

(d) We want to observe how node degrees increase over time. For a few nodes (e.g., nodes 1, 2, 3, 9, 10, 11, 19, 20, 21), plot the evolution of their degree, for instance on a plot such as x=iteration, y=degree, one line per node.

(e) Compare the degree distribution after the first, last, and some intermediary steps.

> To plot several distributions on a same plot, you can either use `seaborn.scatterplot` , providing a *long form* pandas dataframe, i.e., a dataframe with 3 columns (x,y,label) such as each row correspond to one point (x,y) of the experiment represented by *label*. The plot is then done calling `scatterplot(x="x",y="y",hue="label",data=dataframe)` . Another solution is to call several time pyplot `plt.plot(x, y, 'color', label='label')` function.

(f) Vary the number of initial nodes, the number of nodes to add and the number of edges added by each node, and observe how the final degree distribution is affected.