Experimenting with randomized versions of networks

1. Comparing networks and their randomized versions.

   > I recommend to use the airport dataset for interpreting results, but if it is too slow, you can write your code on a smaller network, such as the TV series dataset.

   (a) Using networkx, load the airport dataset.

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

   (c) Generate also a configuration model 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, at least the average degree, clustering coefficient, average path length (takes <3 min. on a google colab notebook). Interpret in terms of small-worldness

   (e) Compute the (approximate) betweenness, closeness, and PageRank of nodes in the three networks.

   (f) Compare the values for nodes of higher degrees between the configuration model and the real graph. Are they similar or different ?

   (g) Plot the distribution of degrees, betweenness and closeness for each network, and compare them.

   > Plotting properly power-law distributions can be tricky. A simple way to do it is to use `collections.Counter` to count occurences of each degree, and plot the resulting keys and values as a *scatter plot* (x=degree,y=occurences(frequencies)). With `seaborn` you can also use `ecdfplot` to plot cumulative distributions.You can plot with loglog scales (`ax.set(xscale="log", yscale="log")`).

2. 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.

3. Going further : fitting exponents

(a) We have seen that a power law distribution is defined by its exponent. We would like to find the exponent of our distribution. First, we try to find it manually. The exponent is the slope of the line on a log-log plot. Can you find it *graphically*? (e.g., if you move one unit to the right on the x axis, how many units are you going down on the y axis to stay on the line)

(b) Let's try to fit by trial and error. Draw lines corresponding to power law distributions of intersect C and exponent α, using the formula of the power law distribution.

> A simple way to draw distribution is to generate series of values for x, e.g., `x = np.arange(1,100, dtype=float)`, and then to compute the y value for each of those x, e.g., `y = a*x+b)`

(c) A naive way to fit the exponent would be to use a least-square regression on log values, i.e., find the slope of the line that we can observe on a log-log plot. You can use for instance the `LinearRegression` method of package sklearn, `model = LinearRegression.fit(log_x,log_y)`, with log_x and log_y being the log values of observed x and y. Intercept and exponent can be obtained with `model.intercept_` and `model.coef_`.

(d) Plot the line and check how well it fits the model. If you're not satisfied, try to impose min and max values of degree to consider.

(e) Fitting power laws with least square is known to be tricky. The `powerlaw` package has been developed to help doing it properly. Using the documentation `https://pythonhosted.org/powerlaw/`, use the package to find the exponent of your distribution. How close is it from your previous experiments? Which one corresponds the most to what is known in theory about the exponent of the prefential attachment model?