Experimenting with Graph embeddings

As in previous classes, if your computer has limited amount of memory or just if you want to save time when experimenting, you can create a subgraph of the airport dataset, for instance only with the most important nodes, or only nodes in a region of the world.

For this class, we will use the `karateclub` library, which contains implementation of various graph embedding methods. As usual, you can install it with `pip install karateclub`.

1. Preparing the network
   (a) `karateclub` library requires graph to respect some specific properties: the graph must be composed of a single connected component, and node names must be integers from 0 to n. First, load the airport graph.
   (b) Extract the highest connected component. You can use `G=G.subgraph(max(nx.connected_components(G), key=len)).copy()`
   (c) Rename nodes from 0 to n, using `nx.relabel_nodes`. To easily retrieve names later, you should keep a dictionary associating node numbers to names, or add the original name as an attribute to the graph (`nx.set_node_attributes`)
2. Computing your first node embedding
   (a) Using `karateclub` library, initialize a DeepWalk embedding model with `model= DeepWalk(dimensions=16,window_size=5)`. `dimensions` corresponds the number of dimensions in the resulting embedding, and `window_size` corresponds to how far away in a random walk 2 nodes can be and still considered in the context of one another.
   (b) With `model.fit(G)`, you can compute the embedding on graph `G`. On the airport dataset, it should take less than a minute or two.
   (c) With `X = model.get_embedding()`, you can now retrieve the embedding of all nodes as a matrix. `X[0]` returns a vector with d elements corresponding to the vector of node 0 in the embedded space.
3. Making sense of the embedding
   (a) A good way to check that the embedding makes sense is to plot it. To do so, the first step is to convert it from its original number of dimensions to 2 dimensions. You can use the TSNE method, with `from sklearn.manifold import TSNE` and `X_2 = TSNE(n_components=2).fit_transform(X)`.
   (b) You can now use `draw_networkx` (passing the 2d embedding to the `pos=` parameter) or Gephi to plot the graph (with the geolayout plug-in and dimensions as latitudes and longitudes). You can either assign colors corresponding to countries to the nodes, or just look at node labels to check that nodes that are close in the graph (i.e., usually, geographically close) are close in the embedding. A quick and dirty way is simply to plot a large graph with `plt.figure(1,figsize=(30,30))` before calling `draw_networkx`.
4. Computing distances
   (a) Compute distances/similarities between all pairs of different nodes according to the original embedding. You can use for instance `sklearn.metrics.pairwise.cosine_similarity`.
   (b) What are the nodes closest to Paris (`PAR_Paris`) in the embedding? Does it seem relevant for link prediction? What about the closest nodes overall? Remember that we have not used any geographical information to obtain this embedding.

(c) Using the code written for heuristics in the previous class, evaluate the quality of link prediction based on the node pair ordering. (If you do not have such a code ready, you can skip this question.)

5. Comparing embeddings
   (a) Answer the same questions as previously, using Role2Vec. Observe the difference between locational and role embedding.
   (b) Answer the same questions as previously, on DeepWalk computed with a significantly different number of dimensions. Compare the results.
   (c) Answer the same questions as previously, on DeepWalk computed with a significantly different context size. Compare the results.
6. Going Further
   (a) Use DeepWalk embedding to do supervised link prediction.