

## 1. Create interpretable graphs

- (a) Get the data for two successive days of Bitcoin activity. To avoid computation difficulties, I recommend to start with some days around 2014 or 2015 to begin with. When the code is ready, you can try on more recent dates.
- (b) We will write a function that takes one day of data and create a graph from it. To create a function, use `def my_function_name(df):`. You can test each line independently and then add it to the function when you're confident about its result.
- (c) write a line to compute the sum of transactions between any two pair of nodes. You can use `groupby`, `sum()` and `reset_index()` to obtain a simple to manipulate dataframe from the groupby result.
- (d) Filter out all actors (sources, destination) that have interactions with less than  $k$  different actors (for instance using  $k = 5$ ). You can use `value_counts()` to count how many times an element appear in a column, and something like `.isin(list[list>=threshold].index)` to get the elements appearing more than a threshold. If unsure, *google* for a way to do it.
- (e) Write a line to remove self-spending, i.e., lines where the source and the destination are equal
- (f) Write a line to remove all lines in which the sum of the value of the transaction is below a threshold, for instance, less than 1 BTC (you can lower this threshold later if you want)
- (g) Transform the resulting dataframe into a graph, typically using `from_pandas_edgelist`.
- (h) You might want to check your graph, for instance using *Gephi*.

## 2. Link Prediction

- (a) First, apply your function on the two days, to get 2 graphs constructed according to the same process.
- (b) Compute some *heuristics*, for instance *Common neighbors*, *Adamic Adar* and *Preferential attachment*. You can use the networkx functions of the same name ([https://networkx.org/documentation/networkx-1.10/reference/algorithms.link\\_prediction.html](https://networkx.org/documentation/networkx-1.10/reference/algorithms.link_prediction.html)).
- (c) A simple way to use the resulting prediction is first to build a dataframe out of it with `prediction = pd.DataFrame.from_records(list(AA), columns=["n1", "n2", "score"])` where `AA` is the result of the Adamic Adar function.
- (d) Now, you can iterate the rows of this dataframe using `iterrows()`
- (e) To make a meaningful link prediction from the first graph, we need to: keep only predictions between two nodes that appear also in the second graph. You can easily test this for each prediction by using a condition such as `if n1 in g2 and n2 in g2`.
- (f) Finally, we want to evaluate the quality of the prediction using the *auc* score, typically from `sklearn` library: `from sklearn.metrics import roc_auc_score`. This score takes 2 ordered lists: One contains a list of scores (e.g., `AA`) for each pair of node, and the other contains the value of the class, i.e., 0 if the pair of node is not connected in `g2` and 1 if an edge exists between the two nodes in `g2`. You can test if an edge exist using `g2.has_edge(u,v)`.
- (g) If the *auc* score is above 0.5, then the prediction is better than random. The closer to 1, the better.
- (h) You can compare different heuristics, different days, and different thresholds for your graph construction

## 3. Embeddings

- (a) 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`.
- (b) `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.
- (c) Extract the highest connected component. You can use `G=G.subgraph(max(nx.connected_components(G), key=len)).copy()`
- (d) 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
- (e) Using `karateclub` library, initialize a DeepWalk embedding model with `model= DeepWalk(dimensions=8,window_size=4)`. `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.
- (f) With `model.fit(G)`, you can compute the embedding on graph `G`. It can take a few minutes on a large graph.
- (g) 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.
- (h) Now, you can use your embedding in different ways. As an example, you can search what are the actors considered the most similar to a particular actor, e.g. `"Kraken.com"`. To do so, you should compute the similarity between vectors. For instance, you can use `spatial.distance.cosine(vect1, vect2)`, with `spatial` from `from scipy import spatial`
- (i) Make a similar test using `Role2Vec`. You simply need to change the name of the method when initializing the model.