Lexical ressources: Wikipedia, Exercise 1

Wikipedia - Exercise 1

This exercise is meant to familiarize you with manipulating data that has been extracted from wikipedia.

This exercise will guide you in using a preparsed wikipedia dump step-by-step. The dump has been extracted and converted into the JSON format using the media-wiki parser from hell, a wiki-markup parser. While this parser does have some issues, it's a good tool to keep in mind. The code canvas contains the instructions to produce the JSON pre-parse from the XML dump, if you're interested in *how* exactly you should do that.

Parsing and retrieving dumps can also be done using other python packages, such as the `wikipedia` python package, the `wptools` python package or gensim's submodule `corpora.wikicorpus`. If you're done early with this exercise, try modifying your code to work with either of these other options. This might require you to download a wikipedia dump from `dumps.wikimedia.org`. Be careful of potential timeouts if you query data directly from wikipedia itself.

The exercise will have you compute some simple graph statistics. Namely, we are going to focus on small-world network stats: local clustering coefficient and minimum shortest path. The slides from today's lecture should already contain all the mathematical formulae required, but they more often describe brute force approaches and might not be the most suitable here. You may be required to find less naive solutions at times!

1.1 Write a function to load the JSON dump

Implement the function `def load_jsons(jsondump_filename)` from the code canvas.

For your convenience, the XML English Wikipedia dump has been preparsed into a `enwiki-dump.json` file. Each line in this file corresponds to JSON string, which you can convert into a python dictionary using the `json.loads` function.

To avoid overusing memory, you might want to to produce a generator rather than a list.

1.2 Write a function to convert JSONs into a graph

Implement the function `def compute_graph(jsons)` from the code canvas.

In this graph, each vertex is to be represented by a specific page title, and each edge is to be derived from links. Links from a given page are already listed in the JSON objects, under the `links` key.

You can tackle this exercise much like you did last week with the wiktionary, excepted that you don't have to convert the graph into a matrix format afterwards.

1.3 Write a function to compute the average local clustering coefficient

In the provided code canvas, implement the function `def compute_average_local_clustering_coefficient(graph)`

You may want to break down this function into subfunctions, as the algorithm itself is easily decomposable:

- First, retrieve for a vertex v its out-neighbors, ie. all the vertices n such that there is an edge (v, n) in the graph.

- Second, compute the local clustering coefficient for a neighborhood: count the number of edges connecting two neighbors, and divide that by the number of possible edges.
- Lastly, compute the average local clustering coefficient: sum the local clustering coefficients of all vertices in the graph, and divide the total by the number of vertices in the graph.

1.4 Write a function to compute the average shortest path length

Lastly, in the provided code canvas, implement the function `def compute_average_shortest_path_length(graph)`.

This function is somewhat tricky to implement in an efficient way. To proceed, you should consider the shortest paths vertex by vertex, and compute the shortest path from a vertex v to any other vertex u reachable from v.

- First, select all the out-neighbors n of v, and add a path of length 1 between v and n.
- Then, for each node n, select all its out-neighbors m that have yet to be selected, and a path of length 2 between v and m.
- Within the set of remaining unselected vertices, select all the out-neighbors of the last picks (viz. the previous m vertices), and add a path of length one more that what you had previously.
- *And so on, and so on...* The process should stop when there are no longer any reachable unselected vertices.
- This process can be neatly written as a recursive algorithm. Keep in mind that we are interested only in the average shortest path length: hence you only need to keep track of the sum of shortest path lengths and the number of shortest paths (there is no need to keep in memroy what the shortest path between any two nodes is).

Once you have a function that returns you for any given vertex v the number of vertices reachable from v, and the sum of shortest path lengths starting from v, you can iterate over all the vertices in the graph, apply your function to each of them, and then compute the average (sum all the sums, and divide it by the sum of all the number of reachable nodes).

1.5 Put everything together

Write a `main` block (the bit that goes under a `if __name__ == '__main__':` instruction) to execute all your functions in order, and print out the average local clustering coefficient and the average shortest path length of the json dump.

There are many ways you can improve this code. A very good complementary workout would be comparing your extracted statistics to what a null-hypothesis random graph would look like. You could for instance:

- generate a random graph with as many vertices as your wikipedia graph,
- statistically compare the local clustering coefficients of the random graph with those of the Wikipedia graph,
- and likewise compare the average shortest path lengths per vertex in the random graph and in the Wikipedia graph

The statistical comparison can technically be done using a Mann-Whitney U test (that's not the best practice though). Have a look at the scipy function for that!