COVID19lib

Introduction

In this documentation we introduce the main function used to produce the results showed in "Superspreading k-cores at the center of pandemic persistence". While the code is freely available for those who wants to reproduce these results or want to perform the same analyses on similar data, the raw data we used are only available under reasonable request to the corresponding author of the reference paper.

The is a *beta*-version of the code (which hopefully, will be improved in a second moment), as well as this documentation, are provided by Matteo Serafino (matteo.serafino@imtlucca.it) & Higor Monteiro (higor.monteiro@fisica.ufc.br). For any doubts the readers may directly refer to one of the authors.

For a better understanding of the code, as well as for the interpretation of the results, we warmly suggest to read the reference paper.

Since the codes of this project deal with sensible GPS data coming from several different mobile APPs, localized in all Latim America, we cannot provide at first any sample data. However, we consider of extreme importance the distribution of the main source code responsible for the data analysis to complement the methodology described in the paper and, of course, for authorized use in new implementations. In future versions, we may create an online workspace where user can to interact with the raw data, without having a direct access to them.

The COVID-19 class

We provide two different, independent classes. `COVID-19` and `Mapcore`. Such package is based on several common libraries of python. In the next section we list the external packages (which are of common use with python and requires little effort for installation) the user must have installed on his own machine if she/he would like to work with with `COVID-19` and/or `Mapcore`.

`COVID-19` is structured in sub-classes. Here we list them and give a general overview of their functionalities, while in the next sections we explain in depth how each of them works.

the main sub-classes are:

1. Contact network generation.
2. SIR dynamics.
3. Percolation/Attack strategies.
4. Mobility (MSD).

`MapCore` is a single class with the goal of producing folium maps containing the spatial information of the disconnected components of a selected k-core of the contact networks. In a future version, extra functionalities, concerning spatial features of the networks, will be added to this class.

Sub-classes Overview

Contact network

This function builds transmission networks for different time windows. The input is a list of timestamped and geolocalized contacts between mobile IDs. Nodes are unique mobile IDs and two nodes are a link if they have at least one interaction (only nodes which have at least one contact with a probability $P \geq p_c$ are used). The list of contacts is generated according to the probabilistic model described in the paper and the user should refers to it for further details.

Given a window we firstly select all the infected people (layer 0) together with their contacts (1 layers infected). We then proceed to find the contact between first layers infected and the other IDs (2° layer infected). In this fashion, we keep adding n layers, with n input variable of the function. The networks generated in this class are used for the main analysis showed in the project paper. They are essential inputs for all the other sub-classes of `COVID-19`, but the *Mobility* sub-class.

SIR

SIR can be used to perform a susceptible-infected- recovered model on a network which has the same vertex properties of the networks generated above. The model we implement here

has recover probability equals to 1. Given a spreader, we compute the number of nodes it infects given an infectious probability β.

Mobility (MSD)

In order to understand the mobility patterns of a population, this sub-class can be used to compute the daily root mean square displacement (RMSD or MSD) of the users in the GRANDATA data. Note that this function needs the original raw data in order to work. Also, this class counts with extra functionalities to associate mobility patterns with the k-core structure of the contact networks generated.

Percolation/Attack strategies

Percolation theory refers to important measures and techniques to achieve the global disruption in the network connectivity of complex systems. To address the problem of finding the optimal strategy to destroy the global connectivity of the contact networks, this class makes use of several node centrality measures.

K-core spatial visualization

Since the networks built are generated from geolocalized contacts data, it is also very important to visualize the spatial distribution of these contacts. In the project's paper, we gain great insights by looking the places where contacts happened. More specifically, it is even more important to check where the more risky people, called superspreaders, meet. For that, this class makes use of folium maps to visualize important features of the networks through spatial distribution of contacts and components of the highest k-cores.

This class also provides extra features to work with the network layouts and HTML movies corresponding to the trajectories of specific superspreaders. (more likely in another version of the code)

Note about input specifications

It is worth to stress that all these functions work with some specific input types. These inputs generally are formatted according to the ElasticSearch framework developed specifically for this project (add a doc for the ES framework). Thus, the user should have in mind the narrow applicability of these classes for general settings.

Dependencies

In this section we simply list the packages we do use in our libraries. Not all of them are used at the same moment, however, in interested reader must have all of them in order to proceed in the analyses' flow.

`Numpy` , `pandas` , `random` , `glob` , `collections` , `datetime` , `igraph` , `geopy` , `joblib` , `tqdm` , `pickle` , `folium` , `geohash` , `graph_tool` , `matplotlib,` `shapely` , `pytz` .

Sub-classes

Contact network

The aim of this sub-class is to create a contact network starting from the raw data we receive from the ElasticSearch framework. The function can be initialized as:

```
CN=COVID_19.contact_network(in_folder,out_folder,chunks=4,rho=0.9,layers=5,c
```

Where:

- `in_folder` : path where the raw data (chunks) are located.
- `out_folder` : path of the folder where the output (series of *GML* networks and a numpy array) will be saved.
- `chunks` : is the number of chunks used to build the network. By default `chunks=4` .
- `rho` : is the value of the p_c .
- `layers` : is the number of layers used to build the network. By default, `layers=5` .
- `cores` : number of cores used in the computation. If bigger than 1 a parallel version of the algorithm is used. By defaults `cores=1` .

This sub-class is composed of several functions. Each of them can be used alone. In particular we have:

```
CN.remove_grid()
```

This function does not take any input. It groups all the chunks together, remove (if any) point belonging to the grid, and save the full list of contact in a unique `filtered_risky_contacts.csv` file in `out_folder` . Notice that this function must be used only once. Once the file is saved, you can load it by means of the function:

```
CN.load(save=True)
```

This function load the `filtered_risky_contacts.csv` and than filter it, by considering all those nodes that have at least one connection with $P > pc$. *If* `save=True`, then the contact list filtered in such a way is saved as `filtered\risky_contacts_p_c.csv`. Differently from the previous function, the latest function should be used every time we change value of p_c. We can finally generate the contact network by means of the following main function:

```
_=CN.contacts_network(start,final,days,window=7)
```

where:

- `start`: left boundary of the time window. The network is build by using all the contacts between `start` and `start` + `window`. `start` should be a datetime (or date) object from datetime package, following the format YY-MM-DD (example: `start = datetime.date(2020, 3, 1)`).
- `final`: last date point to consider. The format is the same as `start`.
- `days`: Step size between the dates. It must be an integer. By default its value is 2.
- `window`: length of the time window. It must be an integer. By default its value is 7.

Pipeline Flow

The pipeline flow of this function is the following:

Step 1: Load the file `filtered_risky_contacts_p_c.csv`.

Step 2: Compute all the contacts between `start` and `start` + `window`. The network is built in a hierarchical way. We first check the contacts between infected and non-infected (layer=1). This non-infected that have been in contact with the infectd becomes part of the first layer of the network. We then look at the contact between first-layer infected and non infected. We repeat the procedure `layers` time.

Step 3: We save (in `out_folder`) the network as `g_index.gml`, where index goes from zero to the number of windows we decided to use. We also save the Giant Connected

Component (GCC) of the network as `gc_index.gml`. For each network, we also save specific vertex properties, such as the layer and the ids of all vertices.

Step 4: We repeat the previous steps with `start` + `days` until `start` + `days` < `final`.

Notice that, the networks generated by mean of this sub-class are the ones used in SIR or Percolation. If you would like to use your own graph, please verify it meets all the characteristic (as for example the vertex properties) of the graph produced above.

SIR

SIR models are generally used to simulate the spreading of a disease in a network starting from some initial spreaders. The function below perform a basic SIR model with recovering probability equals to 1 (i.e. after being infected each person recover) and infectious probability equals to β. The function needs to be initialized as follow:

```
SIR = COVID_19.SIR(beta_max,out_folder,name,sampling=100,layer=0.0,cores=1)
```

with

- `beta_max` : Maximum value for the contagious probability. The values of β considered will be the ones in [0, `beta_max`], with step of 0.02.
- `out_folder` : Path of the folder where the output will be saved.
- `name` : String the will be used as name of the *numpy* array that will be saved.
- `Sampling` : Number of iterations for each node. The default is 100.
- `layers` : Users to consider as a spreaders. By default we only consider as spreaders the users belonging to the layer zero.
- `cores` : Number of cores used in the computation. If bigger than 1 a parallel version of the algorithm is used.

Once the the function is initialized, the algorithm ca be simply call by:

```
_ = SIR.run_SIR(g)
```

This function will simulate a SIR(susceptible-infected-recovered) spreading no the network `g` , which must be a gml network, with the same vertex properties of the networks created by by the `CN` function. The function returns to -1, while the result will we save in a numpy array, in the simulations_results folder. For more details, interested reader may refer to the **COVID_19_notebook**, where outputs of the functions are handled.

Pipeline flow

The flow of the algorithm is the following:

- Step 1: Select a spreader (node)
- Step 2: Find its neighbors and infect each of them with probability β . Change the state of the spreader to recovered.
- Step 3: For each of the infected nodes, repeat Step 2, until no more infected are found. Storage the number of infected people m .
- Step 4 Repeat Step 2-3 `sampling` times. Compute , i.e. $M = \langle m \rangle$ the average number of infected people.
- Step 5: Repeat the previous steps for each spreaders.
- Step 6: Repeat the previous steps for each value of β .

In order to understand the contribute of each k-shell to the spreading, we compute the average of the infected population for the nodes belonging to same shell shell. This is done by means of the function `g.coreness()` , provide by igraph library. Interested readers may refer to the code.

Percolation

This sub-class performs different percolation (attack) strategies on a network. The main idea of the percolation methods is to find those nodes that, if removed, would break the network structure. We implemented a function that does so. The function needs to be initialized as:

```
attack = COVID_19.Percolation(g, radius ,layers=None)
```

where `g` is a *gml* version of the network to investigate. Radius is the ball radius that will be used to compute the CI of the nodes. If `layers` is different from `None`, than only nodes belonging to a layer $\leq$ `layers` will be removed.

Once the function is initialized the percolation can be performed by means of

```
GCC,q = Attack,percolation(modes=string)
```

The options for `string` are

- `'random'` : Nodes will be removed randomly.
- `'degree'` : Nodes will be removed according to the degree (starting from the node with the highest degree).
- `'closeness'` : Nodes will be removed according to their closeness centrality(starting from the node with the higher closeness).
- `'CI'` : Nodes will be removed accord to their value of the collective influence computed on a ball of radius equals to `radius` (starting from the node with the higher value).
- `'KC+BC':` This is a mix strategies. We first select the nodes in the highest k-core and then we remove them according to their betweenness centrality.
- `'KC+HB':` This strategy it's really similar to the previous one, but here we use the degree instead of the betweenness centrality.

- `'betweenness'`: if `layers` equals to *None*, nodes are removed according to their betweenness centrality (BC). Otherwise only nodes belonging to a layer $\leq$ `layers` are removed according to their BC. As an example, if `layers` =1, than the percolation will be made by deleting the nodes in the layers 0/1 according to their BC. Nodes belonging tho the other layers,, are not considered, even if they have an higher value if BC.

The percolation proceed until the side of giant connected component of the remaining graph after the removal is bigger than 0.0005 times of the original side of the network. The function return to two numpy arrays, that is the size of the GCC after each removal and the number of removed nodes q.

Pipeline flow

The flow of the algorithm is quite simple. Given a data strategy :

- Step 1: We remove the node with the highest value of the chosen centrality.
- Step 2: We compute the Giant connected component of the graph after the removal.
- Step 3: We recompute the centrality measure (adaptive attack) we have chosen.

We keep removing nodes until the size of the network is bigger than 0.0005 times its original size.

Mobility (MSD)

This class can be used to compute the daily (root) mean square displacement of the users in the GRANDATA datasets. Notice that this function needs the original raw data in order to work. While we release the function, the data are available under reasonable request to the correspondent authors. The function is initialized in the following way:

```
MSD = COVID_19.msrd(in_directory,out_directory)
```

where `in_directory` is the path where the raw data are saved, while `out_directory` is the path of the folder where the simulations_results folder is located. Raw data are stored in daily files named as *Month_day.csv.* Two more commands are required in order to complete the task. In particular:

```
_ = MSD.generate_pickle()
_ = MSD.daily_msrd()
```

Both function do not require any inputs. The first function generate daily pickle files, in which we store a dictionary containing for each user, her/his position in time. The second function works on this files. The daily MSD is computed as: $\frac{1}{\sum_i^N 1}\sum_i^N MSD_i$ where the index i moves over the N unique IDs in each day and MSD_i is the mean square root displacement of the i *-th* id. The final result consist in a file (*daily_msrd.csv* and saved in the folder **simulations_results**) containing the mean square displacement over time.

Pipeline flow

This function works basically in three steps:

- Step 1: create the pickle file for all the available data. As we already said each of this files contains a dictionary that associate for each id its position through time.
- Step 2: Compute the daily mean square root displacement for each id.

- Step 3: Compute the average daily displacement.

While the generation of pickle files may in principle be avoided, it allow us ti improve significantly the speediness of the computation.

K-core visualization

MapCore class

The `MapCore` class uses the filtered contacts after the `remove_grid()` and the thresholding by p_c in the contact network class. This way, the class loads the table containing all the filtered contacts from the `contact_folder` folder and it should be initialized by passing a contact network `g` , `date` together with `contact_folder` path string to indicate the location of the filtered table of contacts generated over the results of the ES framework:

```
from lib.MapCore import MapCore
objmap = MapCore(g, date, contact_folder, rho=0.9)
```

where:

- `g` : the network that should be used for the k-core visualization. It must contains two vertex properties: *ids* (string) and *layers* (int or float).
- `date` : the start date of this network as defined in the `contact network` subclass. `date` should be a datetime/date object.
- `contact_folder` : The folder where the `filtered_risky_contacts_X.csv` table are stored. `X` is given by the `rho` argument below.
- `rho` : value used to generate the filtered contacts table in the `contact network` subclass.

This way, the `objmap` holds all the contacts information and the network structure of `g` . At this point, no map and aesthestic parameters was set. For that, we must initialize a folium map outside the scope of the class and pass the latter to the former. Following the example of the project's paper, we initialize a map using coordinates for the city of Fortaleza and pass it to `objmap` :

```
from folium import Map

```

```
loc = (-3.775811, -38.530402) # FORTALEZA
m = Map(location=loc, zoom_start=12, max_zoom=17, min_zoom=7, tiles='opens

objmap.initialize_map(m)
```

Visualizing the components of the k-core

The main spatial information we are interested when plotting the map of contacts is not only to see the places of contacts, but also to grasp the distribution of contacts for different components of the given k-core (a k-core can be composed by disconnected components). For that, we only have to run the following line:

```
m, place_info = objmap.set_cores_on_map(start_core=5,
                                        weight_filter=0,
                                        geo_precision=9)
```

where `m` is the final map with the proper visualization, and `place_info` is the table of places, given as geohashes with all the information necessary for plotting any modified version (if needed) of `m`.

`start_core` is the core that we want to visualize the contacts on the map. This core is obtained by the k-core decomposition on the `g` network given at the `MapCore` object initialization.

`geo_precision` refers to the geocoding resolution used to aggregate the contacts on the map. In this project we use geohashes to encode the spatial coordinates of contacts between unique IDs. The length of a geohash reflects its resolution. Here, the default value is `9` for the geohash lengths.

Setting the colors

Different values of `start_core` can result in different number of disconnected components, and each one should receive a different color for proper visualization on the folium map. For that, the user can choose a predefined colormap (according to matplotlib's colormaps), which is independent on the number of components of the current k-core, or

the user can provide a custom colormap by passing a list of hex color codes. For the second case, the user is responsible to guarantee that the passed list has a length of, at least, the number of components of the current k-core.

To use a matplotlib's colormap, only one line is required before calling the `set_cores_on_map()` function:

```
objmap.set_colormap(cmap_name='gist_rainbow')
```

Where, by default, `gist_rainbow` is the chosen colormap. The user can look the list of colormaps available here.

To set a custom colormap, the user must pass a list of hex colors strings. For example:

```
colorlist = ['#fcba03', '#31fc03', '#03f8fc', '#fc03e3']
objmap.set_custom_palette(colorlist, repeat=0)
```

where `repeat` corresponds to the number of times the function repeats the given list in order to make it larger. In the example above, if `repeat=1` the color list will have size 8, not 4. In this case, the final color list will have the conditional `colorlist[:4]==colorlist[4:]` as True.

Manipulating spatial points

The maps produced by the `set_cores_on_map()` can be very noisy containing a lot of circle markers very close to their neighbors. However, sometimes for visualization and for sharing results it is better a cleaner map with fewer circles. To achieve that, one approach is by only change the `weight_filter` parameter. With this parameter, all markers with number of contacts less than `weight_filter` will be filtered out from the map, letting only the larger markers on it. This allows a less messy map while keeping the main massage of the map. Another approach is to produce a new map by directly manipulating the points given in `place_info` table.