Synthethic population generation

Data pre-processing, code report and results analysis.

Enrico Ubaldi

Introduction

This document reports the procedure of synthetic population generation using aggregated data from the Eurostat and a limited list of national surveys as well as the code structure and approximations made to generate the population.

The report is organized as follows: in Section 1 we quickly review the state of the art of synthetic populations generation and we provide arguments to explain why our procedure may be of interest and the improvements that it introduces with respect to alternative approaches. In Section 2 we outline the data fetching and pre-processing procedures to pass from the raw census data to the input of the synthetic population creation. The procedure to generate the latter is presented in Section 3, where we discuss the algorithm implemented to generate the population, the approximations made and the code structure. Finally, in Section 4 we review the accuracy of the generated population by comparing it to empirical data.

1 Motivations

in literature we can find two kinds of synthetic population generation procedures: Synthetic Reconstruction (SR) techniques and the Combinatorial Optimization (CO). The first set of procedure requires a micro-sample (a table reporting a survey on several socio-economic indicators of a small sample of households and individuals in a given area) to calculate the attribute table, usually implementing an Iterative Proportional Fitting (IPF) procedure to generate the joint distributions of the quantities one wants to reproduce in the population. On the other hand in the CO approach, which is rarely implemented, one divides an area in many sub areas for which the marginal distributions of the traits to be reproduced are given. Then a sub-sample of the general population is used to fit these marginals distributions and to represent the population of the sub area.

Both of these methods produces populations of individuals grouped in households and organized accordingly to administrative areas (of different granularity levels) reproducing with good accuracy the traits of the population. However, both of these procedure do not scale well when the number of traits to reproduce increase and when one wants to generate a population replicating not only the age structure of the agents but also their organization in households. In other words, if one wants to constrain both the agents and the household marginal distributions the procedures quickly get cumbersome and the amount of data required to carry out the generation procedure suddenly increases. Moreover, all of these

procedure requires a micro-sample to initialize the contingency table (joint distribution) of the traits to be replicated. These samples are not always available and they may be available only for a small part of the geographical region of which one wants to generate a population.

Work has been done to avoid the requirement of a large and detailed micro-sample. For example in [1] a procedure to generate a population of Belgium without a sample has been carried on. However, the procedure requires the data about joint distributions in the country that may not be always available and the procedure cannot then be generalized.

Moreover, in epidemics modelling one is interested not only in the household arrangements of the agents but also in which workplace or school they go and in which district of a given city they live, so that one can infer the set of people interacting with a specific agent during an ordinary day. Moreover, many Agent Based Models (ABMs) requires a hierarchical organization of the population and of the workplaces. This is especially true when dealing with multi-node ABMs, where many computing nodes are simulating the system under investigation. Within this approach it is therefore essential to be able to split the system among the different nodes at an arbitrary level of the hierarchy and to add a custom number of levels to group agents in local clusters, thus refining the most disaggregated level of the census data or administrative boundaries). For example, we may have a region where municipalities are the finest subdivision of the territory. Let us assume that we want to further split agents in smaller groups based on their household location within the municipality. For example, we may want to group them in districts of about 5000 people, then split in communities of about 800 people which are split in neighborhood groups of $\sim$90 people (about 30 households) in turn. This last step can be used to insert an additional layer of interaction between the agents in the system besides the household and workplace contexts.

To sum up, our procedure builds on previous works of synthetic population generation and addresses these two important issues:

- it provides a general, step-by-step procedure to generate a synthetic population (potentially of the whole Europe) using only open data coming from the Eurostat and from a limited array of national surveys;

- it generates a population with the agents organized in household and workplaces. All these entities are arranged in a hierarchical structure with an arbitrary number of levels and with the possibility to specify the size of each additional local level.

- it arranges the agents' household and workplace locations so as to reproduce realistic commuting data.

Given these features our generated synthetic populations are suitable to be used as input for epidemics-like ABMs or contact driven models where the daily contact patterns of one agent lead the dynamics of the underlying process under investigation.

2 Data pre-processing

2.1 Data sources

The data used in the synthetic population generation are mainly coming from the Eurostat database [2]. In particular we retrieve information on:

- fertility rates by age and NUTS 2 region (*demo_r_frate2* table) and the "life table" (*demo_r_mlife* table) for mortality rates by NUTS2 code to reproduce demography;
- participation rates by age group (*educ_uoe_enra14* table) and the education attainment level of the 25-64 age group (*edat_lfse_04* table) to establish the education level;
- the employment rate by sex, age and education level (*lfst_r_lfe2eedu* table) at the NUTS2 level to determine the employment status of each agent;
- the population by sex and age in the NUTS3 regions (*demo_r_pjanaggr3* table) to generate the correct amount of agents for each sex and age pair;
- families by type, size and NUTS3 region (*cens_11fts_r3* table), the private households by type, tenure status and NUTS 2 region (*cens_11htts_r2* table) and the population by family and marital status and NUTS 3 region (*cens_11fs_r3* and *cens_11ms_r3* tables, respectively) to get the marginal distribution of households.

While the Eurostat data cover the socio-economic traits and the household structures to be reproduced in the synthetic population we need to pair these data with additional sources reporting the size of workplaces and schools, the commuting, the spatial density of the population and the administrative boundaries of a country. These additional resources are:

- the SEDAC [3] UN-adjusted population count, reporting the population count for each cell of about 1 km squared in the whole world;
- the 2012 PISA [4] primary and secondary school size distribution;
- the Open Street Maps (OSM) database for LAU1 and LAU2 boundaries;
- the italian statistical office ISTAT (commuting and fraction of commuters).
- ISTAT and UK ONS data on workplace size distribution;
- still missing: income data (waiting for Andreas (GCF) input);

2.2 Data import

All of the pre-processing procedure is aimed at converting the raw tables downloaded from the Eurostat database to pandas dataframes that can later be used in the synthetic population generation procedure.

All of the steps done are reported in:

- `synPop_0*_eurostat_20**.ipynb` reduce the household composition data from the Eurostat data to a familiar data format. Specifically, it aggregates the redundant family and household types to the six types most representative (for example, it merges together married, consensual and registered couples to be a generic couples). These six types of households are:

  - `CPL_NCH` couple with no children;
  - `CPL_WCH` couple with one or more children;
  - `M1_CH` lone father with one or more children;
  - `F1_CH` lone mother with one or more children;
  - `A1_HH` single adult living alone;
  - `MULTI_HH` multiple adults household (e.g., a house composed by students or workers living together);

  In the notebook we also manipulate the family status per age, i.e. the age distribution of a given member of an household covering a specific age. For example, we compute the age distribution of the children living with a couple of parents. Another statistics converted to the dataframe format is the household size distribution by household kind. All of these data are converted to a table whose rows are the NUTS code to which the statistics belong and the columns are the observables to be reproduced. We create one table for the PDF, CDF and the RAW count of each quantity. For example the dataframe containing the household size distribution looks like this:

HH kind	A1_HH			CPL_WCH				
NUTS / size	1	2	3	1	2	3	4	5
IT1A	1.0	0.0	0.0	0.0	0.0	0.45	0.35	0.15
IT1B	1.0	0.0	0.0	0.0	0.0	0.45	0.35	0.15
IT1C	1.0	0.0	0.0	0.0	0.0	0.45	0.35	0.15

  The overall results of this procedure are:

  - the $H_c(k)$, the household kind distrubtion, i.e. the probability for an household falling within the c code to be of kind k (where k is one of the six types of household defined above);
  - the $S_c(s|k)$ size distribution, i.e. the probability for an household falling in the NUTS code c and of kind k to have size s;
  - the $A(a|r,k)$ age distribution, i.e., the probability for a member of that lives in an household of kind k belonging to the geographical area c and therein playing the role r (i.e., adult/parent or son) to have age a.

- `synPop_01a_schoolsSize_PISA-PIRLS.ipynb` in this notebook we read the school size distribution from the questionnaire with a custom parser of the `SPSS` data format found in the PISA dataset. Then we save the PDF and CDF of the schools of kind k (being k either 0 for kindergartens, 1 for primary schools and 2 for secondary schools) size s distribution $W_c(s|k)$ in the dataframe for each NUTS code c.

- `synPop_01b_wpSize.ipynb` here we import the data from the ISTAT [5] and ONS of UK [6] regarding the size distribution of the activities (workplaces) found in the two countries. Since we find no significant differences in the size distributions of the two countries we project the Italian data to the whole European area as in [7]. The results of this notebook is a dataframe containing for each NUTS2 code c the PDF and CDF of the $W_c(s|k)$ workplace size distribution, i.e. the probability for a workplace of type k to have size s (by now we set a unique id for all the industrial and business activities to be $k = 10$).

- `synPop_02a_eurostat_demography.ipynb` covers the birth and death rates in EU, starting from the `tsv` dataset of the life table of Eurostat. Here we end up with a table reporting for each NUTS code c, year y, sex s and age a the death probability $D_c(y, s, a)$ (i.e., the probability for a person of age a and sex s living in the NUTS area to die before her next birthday) and birth rate $B_c(y, s, a)$ (i.e., the probability for a person of age a and sex s to give birth to a child in one year). We set $B_c(y, s = \text{male}, a) = 0$ for all the ages a.

- `synPop_02b_eurostat_educationActivity.ipynb` in this notebook we handle the data on the education level among the population, the student participation rate in schools and the joint distribution of the education level and employment rate per age group. Specifically, we start by computing the attendance rate $R_c(a)$ of pupils and young adults to school, i.e. the fraction of pupils of a given age a actually attending school. Then, we work on the data of the education attainment level in the population by NUTS2 code. We define three education levels: primary, secondary and tertiary, meaning that a person completed a cycle of a primary, secondary or tertiary (university studies) school, respectively. We report for each education level $e \in [0, 1, 2]$ and for an age group a the percentage of population $E_c(e, a)$ that attained that education level in the NUTS code c. Finally, from the *lfsa_ergaed_lfst_r_lfe2eedu* table we extract the information about the conditional probability $O_c(a, e)$ for an agent with age a and living in the NUTS code c to be employed given his education level e.

- `synPop_03a_NUTS_LAU_fetchProcedure.ipynb` presents the procedure to fetch the local boundaries (the LAU1 and LAU2 levels) from the OSM database [8] using the Overpass API. The OSM data are then converted to the more convenient `geoJson` data format to be later used in the creation of the hierarchical levels of the administrative boundaries. The result of this procedure is a collection of boundaries (geoJson Polygon or MultiPolygon) for each local administrative unit level in the desired region.

- `synPop_03b_NUTS_LAU_aggregation.ipynb`: in this notebook we combine the NUTS3 codes and areas (retrieved from the geographical database, see Section 2.3 for details) with the shapes of the LAU1 and LAU2 codes manually collected in the previous step, ending up with a geo-dataframe containing the information about the hierarchical code (NUTS3+LAU1 and 2) of an area, its population and the shape of the area represented by a Shapely object. This dataframe will then be used by the syn-

thetic population generator to iterate over all the geographical areas that have to be populated. The idea is to produce three objects:

- `NUTS_LAU_hierarchy`, a list whose index is the hierarchical level and whose corresponding item is the geopandas dataframe containing the boundaries of one area, the list of the SEDAC cells overlapping with it and the list of the overlap fraction of the cells with the shape; the lists of cells and their overlap then defines $O_c(i)$, the overlap of the i-th cell with the boundary s; in the same way we can estimate the number of people living in cell i and within the NUTS code c with $P_c(i) = O_c(i) \cdot C(i)$, where $C(i)$ is the population count of the SEDAC for cell i. By summing the population living in each cell c overlapping with area c we get $P_c = \sum_{i \in c} P_c(i)$ and we can define the population density for each cell i in the area as $D_c(i) = P_c(i)/P_c$. These density will be used to arrange the household and workplaces in space proportionally to the population density of the NUTS code area c.
- `index2NUTS` and `NUTS2index`, the mapping between the NUTS code and the index used in the hierarchical representation.

The projection procedure is implemented as follows: starting from the LAU2 codes (districts) we set each of them as the children of the LAU1 boundary overlapping the most with it. We then repeat the procedure for the LAU1 areas (municipalities) with the NUTS3 codes (provinces), thus constructing the hierarchical structure of the population. Finally, we "fill" the holes (e.g., the areas of a LAU1 municipality where no LAU2 districts are present) creating new areas and inserting them into the hierarchy. Most of the functions used to generate the hierarchy of the codes are found in the `synpopUtils.py` module that will be described later.

- `synPop_04_commutingDataISTAT.ipynb`: here we show how to import the ISTAT commuting data [9] to generate the commuting patterns in the synthetic population. In particular, we load the tables containing the information about the fraction $M_c(w)$ of students/workers commuting (movers) to the same municipality of residence within the NUTS code c and for the reason w (being w either to go to work or to attend school); since these are the most detailed statistics on commuting that we have, we will project them to all the Eurostat NUTS codes by sampling from the Italian data (that are given at a regional level). Then, the fraction of workers/students commuting to another municipality to work/attend school will then read $1 - M_c(w)$.

2.3 The geo-database

`MongoDB` is a non-relational, document-oriented database program using JSON-like entries. The database runs on a server and can be replicated and/or mirrored in slaves instances that synchronize for safety.

2.4 mongoDB structure

The program allows for different databases to exist on a single instance and each database may contain different **collections** (corresponding to the tables of a SQL-like database). Each collection then contains the **documents** in a *JSON-like* format, which can be thought as a Python dictionary or a C hash table (or associative array). The document is composed by a list of {`key: value`} couples, binding each key to a corresponding value. While the key are bound to be of string type, the value can be anything from a float, to an array of strings or even another JSON. This allows for nested data storage, for example one can save the population time-series of a given area as:

```
document = {'data': {'census': {'population':
              {'2014': 1230, '2015': 1255, '2016': 1260}
         }}}
```

Every document inserted in a collection is given a unique identifier which is stored as the value corresponding to the `'_id'` key. Every collection is automatically indexed against this field which lets the search and updates query asking for document with a given `'_id'` to run faster. One can also define other indexes on different fields of the documents.

2.5 mongoDB functionalities

Indeed, one of the most interesting and useful features of MongoDB is that it naturally deals with and understands spatial relationships in terms of GeoJSON-like documents. GeoJSON is a format to encode geographic entities such as points, polygons and lines in a given space (which is usually the `(longitude, latitude)` coordinates system) [1]. One can then define a document like:

```
document = {'_id': 'XX000', 'type': 'Feature',
  'geometry': {
    'type': 'Polygon',
    'coordinates': [ [ [125.6, 10.1], [125.6, 10.5],
                       [125.1, 10.5], [125.1, 10.1], [125.6, 10.1] ] ]
  }, 'properties': {
        'name': 'Dinagat Islands', 'data': {'census': {'population': {
                       '2014': 1230, '2015': 1255, '2016': 1260} } }
  }
}
```

where we gave the `'_id': 'XX000'` identifier to the document and loaded it with geographical information in the `'geometry'` key. Also, the GeoJSON format allows a geographical feature to store within the `'properties'` key the values associated with this region to be later retrieved.

[1] See `http://geojson.org`.

2.5.1 Space-based queries

It is now possible to do spatial-based searches using, for example, the `Python` interface provided by the `pymongo` module. For example, let us say that we want to fetch all the geographical entities falling within the $[[120, 5], [120, 15], [130, 15], [130, 5], [120, 5]]$ polygon. In order to do that we connect to the database creating a client, select the database and collection where we stored our geographical entities and perform a geo-spatial query from the `pymongo` interface using the database's `'$geoWithin'` filter like this:

```
import pymongo
client = pymongo.MongoClient('mongodb://user:password@databaseurl.mongo.net')
geoCollection = client['databaseName']['collectionName']
results = geoCollection.find({'geometry': {'\$geoWithin': {
                        'geometry': {
                            'type': 'Polygon',
                            'coordinates':
                                [[[120, 5], [120, 15], [130, 15], [130, 5],
                                [120, 5] ]]
                        }
                    }}})
```

The `results` is now an iterator over all the documents that falls into the polygon specified in the query.

The health habits pilot created two collections in a MongoDB database storing the cells of the 2015 SEDAC population count raster [2] and the European countries' boundaries as found in the NUTS scheme [3].

The two collections are organized so as to provide the following features:

- hierarchical structure of the boundaries to replicate the European Commission's `NUTS` geographical division;
- store census and national health agencies data for each region;
- fast access to the SEDAC raster cells falling within a region to generate the simulation input rasters for the observable under investigation.
- fast individuation of the boundary containing a specific cell of the raster.

The cells collection stores the information about the number of people living in a $\sim$1x1Km area around the world and are stored as GeoJSON polygons (rectangles in the latitude-longitude coordinates system) whose `'properties'` field reports the population count.

The boundaries collection also comprehends polygons delimiting the NUTS at their different levels. The `'_id'` value of these entries is set to the NUTS code which naturally

[2] See `http://sedac.ciesin.columbia.edu/data/set/gpw-v4-population-count-adjusted-to-2015-unwpp-country-total`
[3] See `http://ec.europa.eu/eurostat/web/gisco` for details.

provides a hierarchical organisation of the documents, as the code structure reads `CC123`, where `CC` are two characters identifying the country, while 1, 2, 3 are three alphanumeric characters that are present in the first, second and third NUTS level, respectively (i.e., `CC12` is a code of level 2 for country `CC` which is a children of the `CC1` level 1 NUTS).

2.6 Database interface

Besides the database, which is intended as a permanent and consistent storer of data, the health habit pilot also developed a high-level interface to insert (retrieve) data to (from) each region, aggregate them using different schemas, import the simulation output, compare simulations results with empirical time series and easily visualize results on a map.

The interface leverages on all the database features we presented in the previous section and allows for a quick interaction with data. We provide here three paradigmatic essential examples covering the insertion of data from a **c**omma **s**eparated **v**alues (csv) file in the database, the aggregation of such data from an higher, more refined level to a coarser level (from example from level 3 to 0) and the visualization of such data on a map.

Insertion We start from a csv whose columns store the NUTS codes of the regions to which data are referring and the different values we want to include in the dataset. For example we may have:

NUTScode	CurrentSmokers	ExSmokers	NeverSmokers
UKF11	25%	35%	40%
UKF12	20%	40%	40%
UKF13	10%	40%	50%
...	...	...	...

Suppose that the data refer to the 2012 year. We can insert them using our interface by simply importing the csv as a data-frame (for example using the `pandas` module in `Python`) and then calling the `insertFromPdDf` method from our library instance `geoClient`:

```
smokingData = pandas.read_csv('smokingPrevalence.csv')
fieldsToInsert = {
                'CurrentSmokers':  'properties.data.health.smoking.2012.CurrSmok
                'ExSmokers':       'properties.data.health.smoking.2012.ExSmok',
                'NeverSmokers':    'properties.data.health.smoking.2012.NeverSmo
            }
geoClient.insertFromPdDf(dataFrame=smokingData, keyDF="NUTScode", keyDB="_id",
```

Where `fieldsToInsert` is a dictionary specifying where each column (key) value should be saved in the database record. The latter is given using the *dot-notation*, i.e.

properties.data.health.smoking.2012.ExSmok

will be saved in

document['properties']['data']['health']['smoking']['2012']['ExSmok'].

Aggregation In the previous insertion point we loaded the database with data on smoking prevalence at the higher and most refined level of NUTS3 in UK (as the NUTS codes feature three alphanumeric characters after the country code). Now we want to aggregate these results for LAD 2,1, and, 0 so as to have the smoking prevalence at the district, regional and national level, respectively. In doing so we want to perform a weighted average of the smoking prevalence for all the children codes of a given district (for example all the 'UKF1*' are child of 'UKF1') using as weights the population of the child regions, and then repeat the procedure up to teh 0-th level.

Using our interface this is done by calling the `aggregateCountryLevels` method as:

```
geoClient.aggregateCountryLevels(countryCode="UK", mode="wmean",
    field='properties.data.health.smoking.2012.CurrSmok',
    on="properties.data.census.population.total.2015", levelStart=3, levelStop=
```

where we specified the aggregation `mode='wmean'` meaning weighted-mean (other available methods are *sum, mean, wsum*), and the `on` field specifies the document entry to be used as weight. We also instruct the method to start aggregating at level 3 and stop at level 0.

After issuing the previous command, the collection stores the computed smoking prevalence for all the administrative boundaries between level 0 and 3.

2.7 Input preparation

The input preparation basically consist in downloading the data from the Eurostat, ONS UK and ISTAT resources and put them within the path of the notebooks just outlined. The result is a series of dataframes and python onjects serialized (saved) to disk. Once this procedure is completed we can pass to the actual synthetic population generation.

3 Procedure

The population can be generated by editing a custom configuration file. One can start from the default configuration file `synpopGenerator_config.py` that reports all the configuration options available for the user. We review these options in the following sections also reviewing the overall code structure.

3.1 Code structure

Once the configuration file is complete (it must be a valid python module file from which import the different configurations), one can call in sequence:

```
$ python2 synpopGenerateEntities.py cfg_file_name.py
```

These procedure will generate the workplaces, households and agents and serialize them to disk. However, these entities must still be arranged in space. To this end one can call:

```
$ python2 synpopClusterizeEntities.py cfg_file_name.py
```

That will save the synthetic population in the desired h5 output file. Finally, the population can be checked by calling the

```
$ python2 synpopCheckpopulation.py cfg_file_name.py
```

that will save all the plots and analysis in the `./figures`.

Another way to create the population is to run the `synPop_10_generationDemo.ipynb` notebook that shows, using the `ipywidgets` module, the interface to all the three steps of the population creation and check procedure.

Apart from the script just outlined, the source files involved in the population creation and check are the following:

- `synpopStructures.py` contains the declarations and definitions of the classes involved in the population creation. Specifically, we define:
  - the enumerators indicating the codes assigned to the categorical variables of the population, e.g. the sex and the household role correspondence table;
  - all the functions and classes involved in the creation of the population for a specific are. In particular:
    * `householdType` and `workplace`, the principal groups of the synthetic population. Each of them provides the methods to generate an entity complying with the size and age distributions of one particular NUTS code. Once an household or work kind has been declared and initialized one can change dynamically the size and age distribution of the components via the methods to set the PDF or the CDF of the distribution to update. In this way the household or workplaces kind can be defined once and then used in all the NUTS code of the regions to populate.
    * `generatePopulation` method that, given the household types distribution and the household size distribution for each household kind, returns a table containing the details about the agents and the households they belong to. These data will be used in the synthetic population generation process enriching them with data about the agents employment, education status and school/workplace location.
- `synpopUtils.py` contains the functions exploited in the pre-process part, especially the functions building the hierarchy of the codes. Apart from these pre-processing functions the module contains:
  - `assignToCell` function that, given the cells CDF and their shapes returns one *(lon, lat)* tuple of a point inside the boundary. This function is used to spatially arrange the households and workplaces inside the geographical boundary during their generation.

  - **clustering tools** functions used to compute and create the local clusters starting from the groups of households and workplaces belonging to the same LAU code. The clustering procedure is implemented using either the *kmeans* method (slower but more memory performant) and the *hierarchical clustering* (faster but scaling badly in memory and practically unusable for medium-large populations). The advantages of the kmeans approach are various and make it favourable over the hierarchical clustering procedure. For example, in the kmeans process it is possible to fit the model on a sub-sample of the data and then generalize the results to all the points in the LAU code, thus enabling the clustering of densely populated areas, such as the central districts of the largest cities.

- `synpopGenerator_tools.py` contains all the functions to simplify the access to the dataframes containing the statistics about the socio-economic data previously produced in the pre-processing part. Moreover it contains the functions used to assign the education level to the agents not attending school, the employment status and the school kind one agent is attending, if she goes to school.

- `synpopGenerateEntities.py` contains the core of the generation procedure in the `generateEntities` function that will be detailed in Section 3.2. The result of this procedure is then passed to the clustering process described in the next point.

- `synpopClusterizeEntities.py` starts from the agents and locations generated in the previous step and clusterize them in space, finally organizing all the data in a single hdf5 file. All the steps are contained in the `clusterizeEntities` function. Basically for each LAU2 code we define the local clusters of households and workplaces so as to add the local cluster codes to the NUTS+LAU hierarchical code.

3.2 Algorithm

Starting from the pre-processing data we start calling the `generateEntities` functions and passing the configuration to be used in the generation process. The first step is to load from disk all the statistics and information about the system. We then proceed generating all the entities to be part of the system starting from the households.

3.2.1 Households

The first thing to do is to commit the kind of household that will be part of the population and specify the age constraints of their members (e.g., the maximum age distance between parents and sons). Then we have to specify in which columns of the dataframes we will find the age distribution of the parents and sons for each household kind. The household types are then arranged in an array and a corresponding probability distribution over the types of household is created, thus representing the $H_c(k)$ distribution. Then the procedure is as follows:

1 for each LAU2 code retrieve the statistics on the household composition and update the age and size distribution of each household size. Also, update the statistics on

education level, employment rate and commuting;

2 compute the number of agents to be generated in this LAU2 code c as $P_c \cdot \phi$, where ϕ is the fraction of the population to be generated;

3 create the households and agents of this code c and compute their education level and employment status.

3.2.2 Workplaces and schools

Once the agents and households of each LAU code c have been generated, we pass to the creation of the workplaces they will work into. To this end, we first compute the commuting volume matrix $V_{ij}(w)$ for each commuting cause w (school or work). This tells us how many workers will travel from code i to j for work/study. We set $V_{ii}(w)$ to be $W_i(w) \cdot M_i(w)$, i.e. the number of agents going to the workplace/school kind w within LAU i times the fraction of commuters that commutes to the same municipality accordingly to the ISTAT figures. We then set the off the diagonal elements $V_{i \neq j}(w)$ proportionally to a gravitation model that reads:

$$V_{ij} \propto \frac{W_i(w)^{\tau_t} W_j(w)^{\tau_f}}{d_{ij}^{\rho}}, \tag{1}$$

with $\rho = 2.95$, $\tau_f = 0.28$, and, $\tau_t = 0.66$ as in [7].

Once we know the total number of agents working in each municipality $W_c(w)$ we start sampling workplaces accordingly to their size distribution $W_c(s|w)$ and assign s randomly sampled workers to each of them, until all the workers are assigned. Each workplace, as each household in the previous step, is randomly placed in the LAU area within a SEDAC cell i selected proportionally to the $D_c(i)$ population density.

3.2.3 Local clustering of agents and workplaces

The three tables containing the agents, the households they are living in and the workplaces/schools they are attending are then passed to the `clusterizeEntities` function. This function, given the number S and the target sizes s_i (with $i \in 1, \ldots, S$)of the additional local codes to be added to the Eurostat territorial nomenclature. Starting from the broader group of size s_1 we group all the workplaces and households of the area in $g_1 = \text{int}(P_c/s_1)$ clusters, using a kmeans clustering method on their (lon, lat) coordinates. We then repeat the procedure, within each cluster further organizing the households and workplaces in each one of the g_1 groups in $g_2 = \text{int}(s_1/s_2)$ groups. Then, the hierarchical code of each entity will be extended with $(i_1, i_2, \ldots, i_S)$, being i_j the index of the cluster in which the entity falls at the level j.

3.2.4 Output structure

Once the clustering procedure has completed, the three tables are inserted in three different datasets within an hdf5 file. In each dataset we insert one column for each traits of the entity we want to serialize. Since hdf5 accepts only tables of a single kind we create a

compound type in case we have more than one kind of data for each entity (for example the age is an integer and the income is a float so that we have to define compound types to be inserted in the dataset). We also save the demography table to be later used in the simulations.

The default structure for the three tables is as follow:

Agent

Name	id	hh	role	sex	age	edu	employed	income	wp
Type	int	int	int	int	int	int	int	float	int

Where *hh* is the household id where the agent lives, *wp* the id of the workplaces she is attending (-1 means that the agent is unemployed and not attending any school), *edu* is the education level and *employed* sets the kind of workplace he is working at.

Household and workplaces

Name	id	kind	size	lon	lat	l0	l1	...	lS
Type	int	int	int	float	float	int	int	...	int

Demography

Name	l0	date	sex	age	natality	mortality
Type	int	str8	int	int	float	float

Where *date* is a string reporting the date from which the statistic about the natality and mortality annual probability are valid (i.e., in the simulations we will have to look for the most recent date before the simulated time to inform the demography module with the correct birth and death probabilities).

4 Results

The results can be checked using the `synpopCheckpopulation` function that produces the following plots (the following examples refer to the 100% population of Piedmont in 2011):

References

[1] Johan Barthelemy and Philippe L Toint. Synthetic population generation without a sample. *Transportation Science*, 47(2):266–279, 2013.

[2] E. U. Commission. Eurostat, your key to European statistics. `http://ec.europa.eu/eurostat/data/database`, 2016. Accessed 05-April-2018.

[3] NY: NASA Socioeconomic Data Center for International Earth Science Information Network CIESIN Columbia University, Palisades and Applications Center (SEDAC). Gridded population of the world, version 4 (gpwv4): Population count adjusted to match 2015 revision of un wpp country totals. `http://dx.doi.org/10.7927/H4SF2T42`, 2016. Accessed 21 11 2016.

[4] OECD. PISA: programme for international student assessment. `http://www.oecd.org/pisa/data/2012database`, 2013. Accessed 01-February-2018.

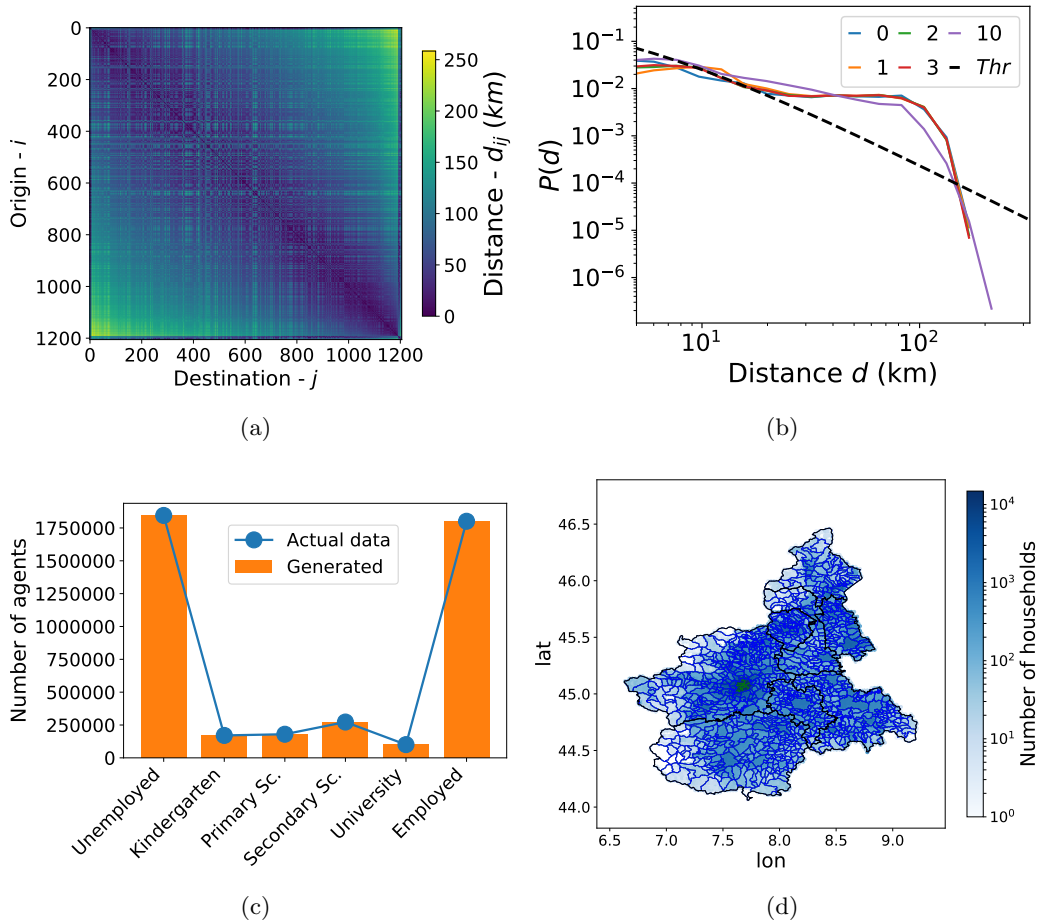
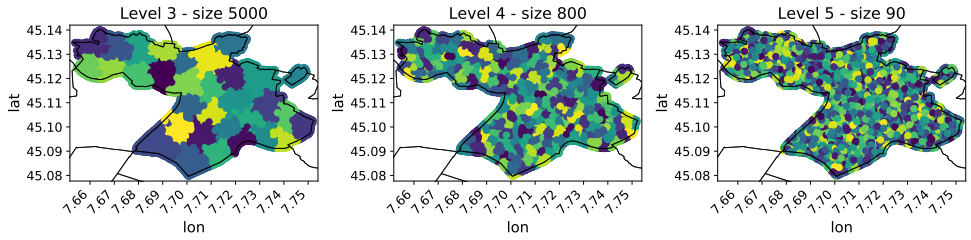
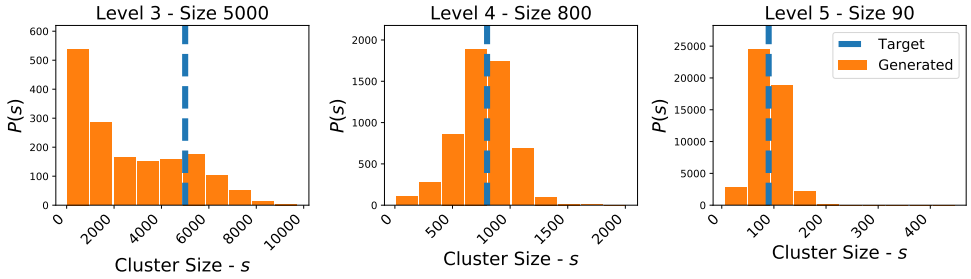


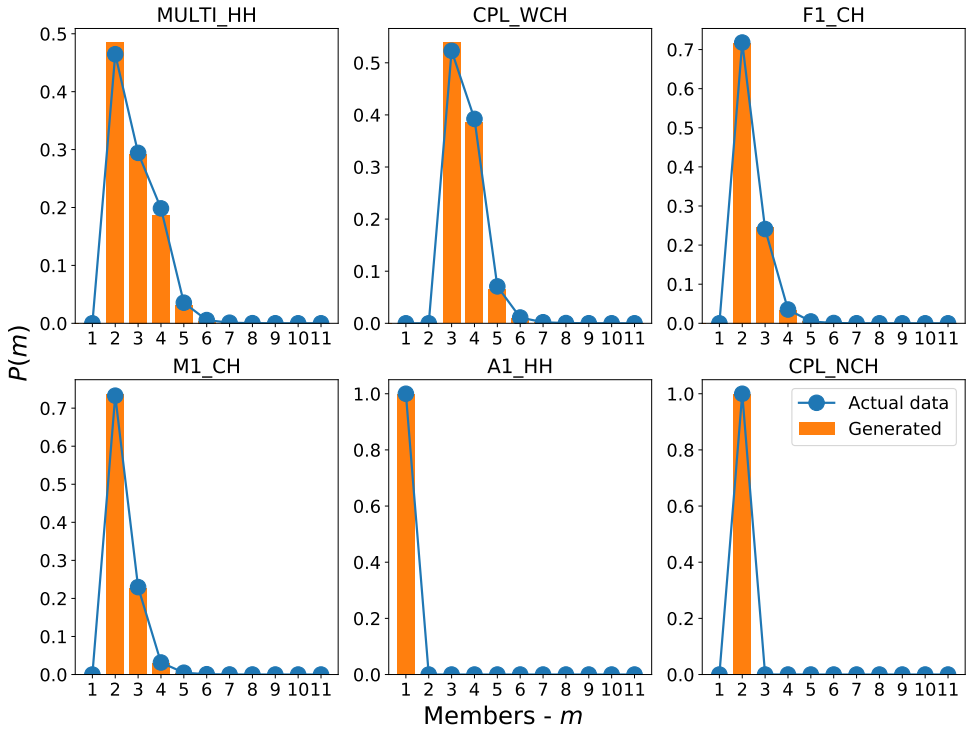
Fig. 1: (a) The distance matrix between the LAU codes in the regions in km. (b) The commuting distance distribution for each workplace type (coloured solid lines) and the theoretical prediction of Eq. (1). (c) The employment kind of the agents. (d) The spatial density of the generated households.



(a)

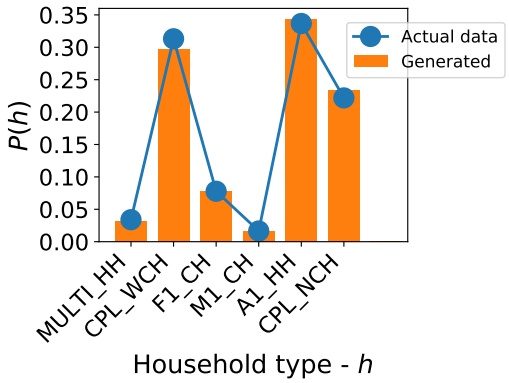


(b)

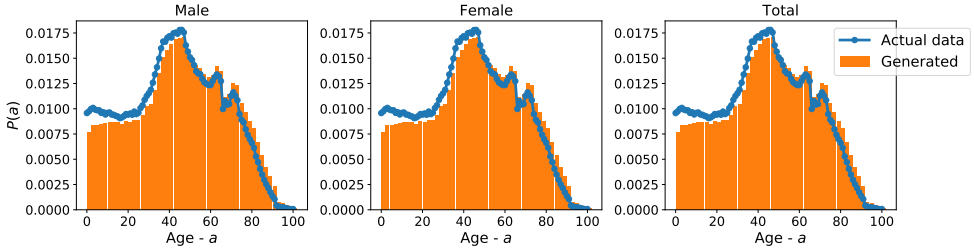


(c)

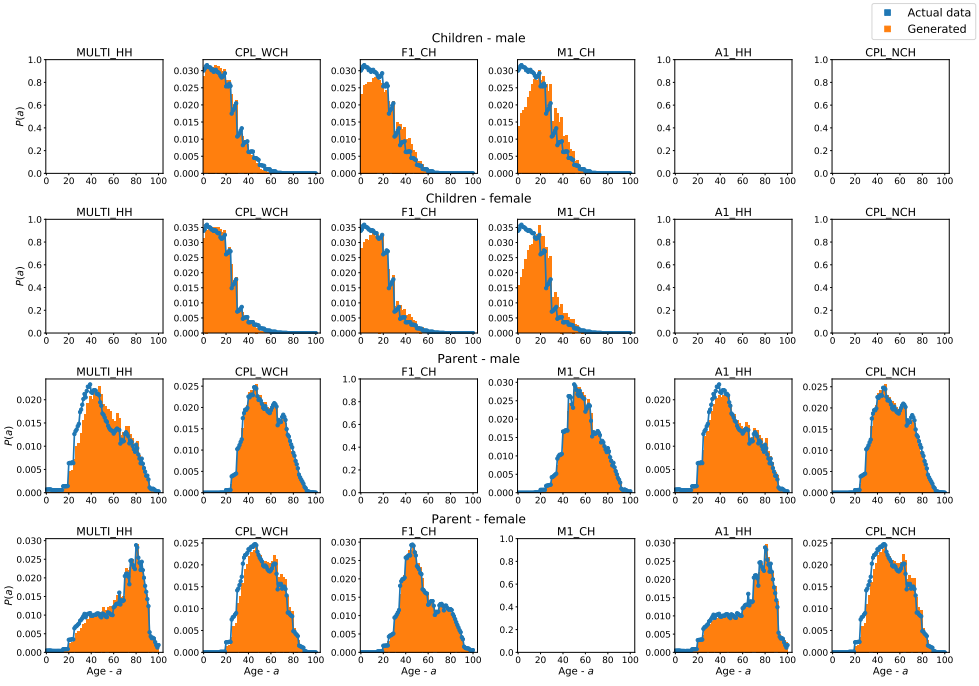
Fig. 2: (a) The local clustering results. (b) The check on the size of the local clustering levels. (c) The size distribution of the actual data and of the generated population for all the household kinds.



(a)



(b)



(c)

Fig. 3: (a) The distribution of the household kind. (b) The overall age distribution of the population per sex. (c) The age distribution by household role and household kind.

[5] ISTAT. Censimento Industria e Servizi. http://dati-censimentoindustriaeservizi.istat.it, 2011. Accessed 10-February-2018.

[6] ONS UK. UK Business - activity, size and location, 2017 data, table 18. https://www.ons.gov.uk/businessindustryandtrade/business/activitysizeandlocation/datasets/ukbusinessactivitysizeandlocation, 2017. Accessed 10-February-2018.

[7] Stefano Merler and Marco Ajelli. The role of population heterogeneity and human mobility in the spread of pandemic influenza. *Proceedings of the Royal Society of London B: Biological Sciences*, 277(1681):557–565, 2010.

[8] OpenStreetMap Community. The Free Wiki World Map. http://www.openstreetmap.org, 2018. Online, accessed 15-February-2018.

[9] ISTAT. Censimento Popolazione. http://dati-censimentopopolazione.istat.it, 2015. Accessed 20-February-2018.