L.M. Geoinformatic Engineering

AA 2020/2021

POLITECNICO
MILANO 1863

Geoinformatic Project

Facebook Data for Good with Open Data Cube

Authors: Marianna Alghisi
Martina Pasturensi

Table of Contents

1. Introduction

1.1. Purpose

The purpose of this project is to provide an example of use, processing and analysis of the data distributed by Facebook Data for Good on the distribution of the population, by using Open Data Cube (ODC) software.

Since ODC was born as a satellite image management platform, the aim of our work is also to provide a sufficiently generic procedure to be able to extend its use to other types of data, in our case Facebook data.

The implementation work is divided in two different Jupyter Notebook, which aim to be a reference framework for the user who approach to data analysis through ODC. The first contains all the steps required for build the reference grid, association of the Facebook Data to the grid, transformation of the gridded data in the file format supported by ODC, production of the metadata required for the storage on ODC and upload on ODC of the produced Dataset. Therefore, it contains the code example that clarifies how to use ODC with data typologies different with respect to satellite images. The second notebook provide an example of use of the data stored in ODC, in particular, is highlighted the data selection and extraction process from the database, in conclusion functions for data analysis, plot and creation of interactive maps are provided. The provided function are sufficiently generic to be implemented with other datasets.

The notebooks have been developed on the Population Density Data of Milan, all the processes described can be extended and implemented for all other types of Facebook Data.

1.2. Facebook Data for Good

Data for Good is a Facebook database that collaborates with hundreds of organizations around the world, including universities, nonprofits and international organizations. Thanks to more than three billion subscribers, there is a lot of data collected, those data are aggregated, anonymized and resized, in such a way that no individuals or companies can be identified.

Some of the products are freely accessible, while others require data sharing agreements. Using information from this global community can help organizations deliver better services; in fact, the use of Facebook data for Good is useful for gathering information on the spread of the Covid-19 virus, climate change, environmental disasters, or for creating population density or business activity maps.

In our work, we used the data describing the population population density in the metropolitan city of Milan, which provide a daily historical series starting from 4 April 2020. The data were downloaded in .csv format and provide much information: data provides a

regular grid of points of known latitude and longitude, for each point different information is recorded.

In our work we will focus on the data obtained from the *'n_crisis'* field, that reports the average number of people registered in an eight-hour interval for the area represented by the point. The Bing Maps tile system was used to allocate users to tiles within the regular grid. According to the box in which a user spends the most time during the eight-hour period, population density measures have been derived.

Another key information related to each point is the '*quadkey'*: since each point represents a Bing tile of a specific zoom-level, the quadkey is the identification code of the tile. It is a string containing a binary numeric value, that is obtained by interleaving the bits of the row and column coordinates of a tile in the grid at the given zoom level, then converting the result to a base-4 number.

The data was downloaded in .csv format and then reworked as will be presented in the following chapters.

1.3. Open Data Cube and working environment

The Open Data Cube (ODC) is a non-profit, open-source project that was motivated by the need to better manage Satellite Data. It is s an Open-Source Geospatial Data Management and Analysis Software project and, at its core, it is a set of Python libraries and PostgreSQL database that helps in working with geospatial raster data.

ODC already provides numerous application tools regarding satellite data. Goal of our work is to understand if ODC is suitable also for other types of data.

The entire development process was carried out in a python environment created ad hoc for the installation of datacube library. The realization of notebooks instead relies on different libraries, in order to develop the work, it was necessary to install the following libraries:

- jupyter
- pandas
- shapely
- numpy
- datetime
- shutil
- geopandas
- os
- glob
- matplotlib
- folium
- branca
- ipywidgets

- seaborn
- contextily

1.4. Document Structure

I. Chapter 1 – Introduction
Definition of the purpose and main goals of the project, overview on the analyzed data and used tools (FB Data for Good and ODC), description of the python working environment and the required ibraries.

II. Chapter 2 – Grid Realization
Description of the routines and methods implemented for the realization of the grid.

III. Chapter 3 – Data Upload on ODC
Facebook data gridding routine and upload in ODC. Description in details of the fundamentals steps to work with ODC: initialization of the database, realization of the product and the database.

IV. Chapter 4 – Data Analysis
Description of data extraction from ODCs, structures used, definition of functions developed for data analysis and plot.

V. Chapter 5 – Conclusions
Conclusions of the work, possible future developments

2. Grid Realization

First step of our work was to create a regular point grid that would adapt to the one used to contain the Facebook data. This step is fundamental in the data grid process, required to upload the data in ODC, since the data belonging to the same ODC product must have the same geometry, furthermore, the data distributed by Facebook can be corrupted or have missing points in areas where no number of people has been registered.

The realization of the grid is divided in two steps: the first, realized in Python environment, consisted in the realization of the grid of regular points and the association of the points with the correct Quadkey, the second instead was done through QGIS and consists in the manual entry of the Quadkeys of the grid's points where they were missing.

2.1. Regular grid with Python

In order to realize the grid, a random CSV file (from '*./population/milan*' folder) is loaded as a Pandas DataFrame, since all data contained in the csv have the same geometry.

The loaded csv file is used to define the shape of the grid: that correspond to the values of maximum and minimum latitude and longitude retrieved by the example CSV:

$$Shape = \big[\,[\min(lon), \min(lat)], [\min(lon), \max(lat)], [\max(lon), \max(lat)], [\max(lon), \min(lat)]\big]$$

Then, once haveng defined the boundaries of the grid, an an empty Pandas DataFrame is introduced, that is filled with the points coordinates through two nested loops on longitude and latitude values. In particular, the distances in degrees between the points' latitude and longitude have been defined as $delta\_lat$ and $delta\_lon$ as follow:

$$delta_{lon} = \frac{lon_{max} - lon_{min}}{82}$$

$$delta_{lat} = \frac{lat_{max} - lat_{min}}{101}$$

The values used in denominator corresponds to the total number in horizontal/vertical -1.

The code for making the grid is shown below:

```
grid = pd.DataFrame(columns = ['id_grid','latitude_grid', 'longitude_grid', 'geometry'])
delta_lat = (lat_max - lat_min)/82
delta_lon = (lon_max - lon_min)/101
i = 0

for lon in np.arange(lon_min, lon_max+delta_lon, delta_lon):
    for lat in np.arange(lat_min, lat_max + delta_lat, delta_lat):
        p = Point(lon, lat)
        row = pd.DataFrame([[i, lat, lon, p]], columns = ['id_grid','latitude_grid', 'longitude_grid', 'geometry'])
        grid = grid.append(row)
        i = i + 1
```

It consists in the simple iteration on all the possible combinations of coordinates that go to make up the grid. The process can be implemented for other types of grids only by changing the border points and the distance between the points.

Once having defined latitude and longitude of the points belonging to the grid, the resulting DataFrame is transformed in a Geopandas GeoDataFrame by adding the column 'geometry', containing all the points of the grid, where each point is a Shapely object.

$$POINT_i = (lon_i, lat_i)$$

The GeoDataFrame's csv is set to '*EPSG:4326*'.

At this point we have a regular grid of points, it remains to insert the '*quadkey*' field, that corresponds to the identification number of each point. This number must respect the conventions used in Facebook's CSVs, in order to create a final generic grid that can be used to collect population data from all CSVs. This field is crucial because allows to create the relationship between the CSVs and the grid.

Most of the quadkey values were obtained by making a spatial join between the created grid and the example csv. The join is performed according to the nearest neighbour algorithm, that joins every point of the CSV file to the closest point of the grid. This procedure is carried out by two external functios: nearest_neighbour(***) and get_nearest(***). More info about these fuctions at:

- https://scikit-learn.org/stable/install.html
- https://automating-gis-processes.github.io/site/notebooks/L3/nearest-neighbor-faster.html

Result of the of the nearest neighbour join is a DataFrame containing all the points of the csv associated to the ID of the respective grid point:

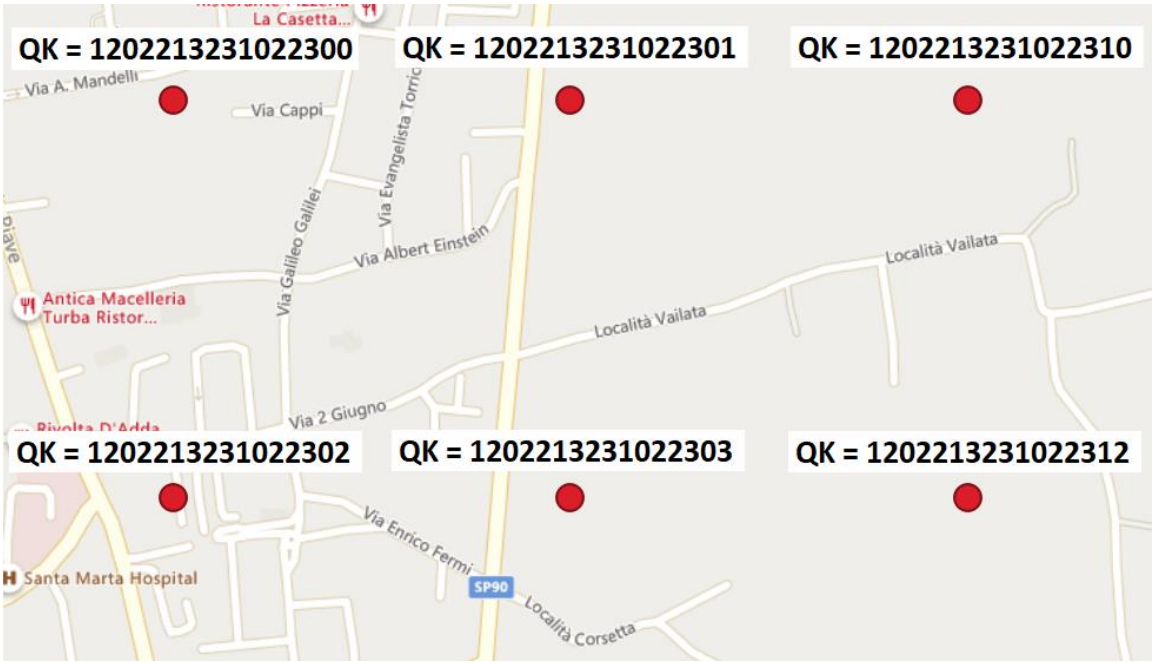
ID_CSV	Quadkey	Latitude_CSV	Longitude_CSV	ID__grid

The above product is merged with the grid itself on the '*ID_grid*' field, non-necessary fields are discharged, only the fields corresponding to the latitude and longitude of the grid points and the joined quadkeys are kept.

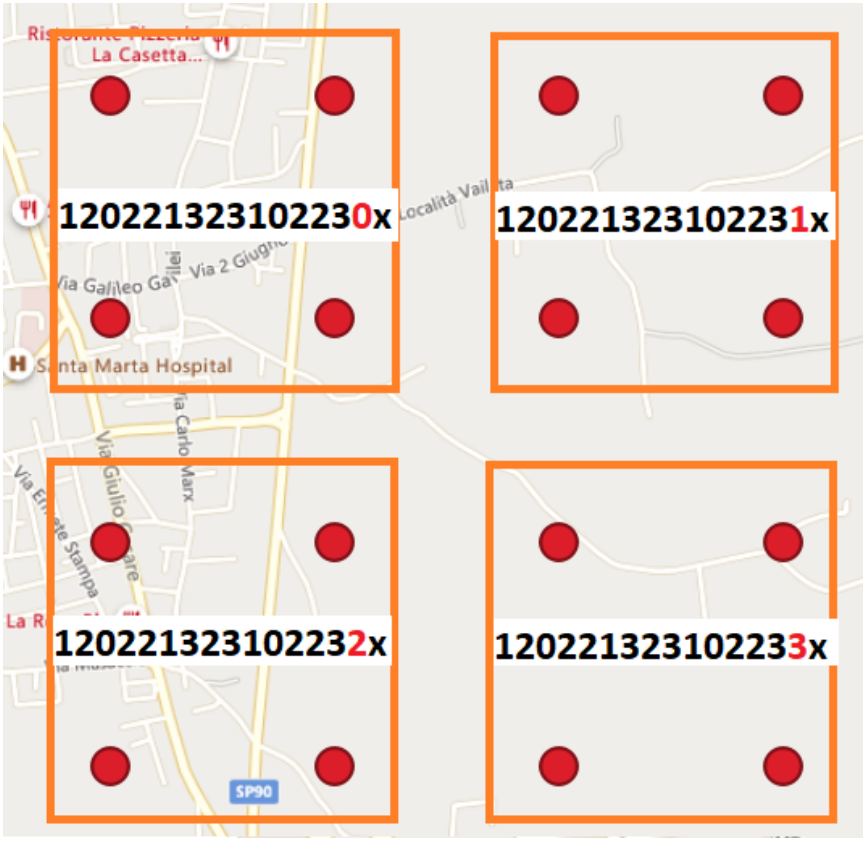
Since the number of points of the grid is greater than the ones contained in the CSV, some points present a missing quadkey value. This happens because in the CSV, according to the data availability, there are some missing points. The final DataFrame is saved as a CSV and imported in QGIS to manually insert the missing quadkeys.

2.2. QGIS work on quadkeys

The final product discussed above is imported in QGIS, all the points having missing quadkeys are evaluated and the quadkey value is manually inserted according to the values of the surrounding points, following the process shown in figure below.



Since the quadkey is a number in base 4, it is composed only of 4 digits that repeat a regular pattern: if we focus only on the last digit, it is evident that, taking a row of the grid, if the last digit alternates 0 and 1, therefore, the previous and the following lines alternates 2 an 3. The same schema is applied vertically to the columns, but in the vertical case, the values alternated will be (0 and 2) and (1 and 3).



It is evident that the penultimate digit identifies the higher-level tile and changes every 4 (2^2)points. Consequently, the third to last digit will change every sixteen points (2^4) and so on, up to the firs level tile.

Since the points with missing quadkey were few, we chose to use this solution, as it is fast and intuitive. since the points with missing quadkey are too many and the manual procedure is too time wasteful, the implementation of an ad hoc function is recommended.

3. Data Upload on Open Data Cube

The loading of the csv on ODC is completely carried out by the gridData function, present in the first notebook. A routine is proposed that automates the data uploading process: the names of the csvs contained in the '/milan' folder are compared with the list of names already loaded on ODC, present in the loaded_CSV.txt file. In this way, it is possible to check if there are new csvs to load. The obtained list of names of the csvs to upload is passed to the gridData function, which applies the gridding and uploading routine to all the new csvs. In conclusion, the list of uploaded csvs is updated.

3.1. Data gridding

The data gridding is performed through a merge between the reference grid and the csv: both files are loaded as pandas.DataFrame and the merge is performed on the 'quadkey' field. The result of the merge is cleaned: all not necessary columns are dropped and all nan values (*not a number*) are set to zero. The final Dataframe contains the following columns: ['time', 'latitude', 'longitude', 'n_crisis']; the fields ['time', 'latitude', 'longitude'] are set as index of the structure, the field 'n_crisis' contains the information regarding the population density obtained from the csv.

```
grid = pd.read_csv('path_to_grid.csv')
csv_df = pd.read_csv('path_to_FB_csv.csv')

gridded_csv = grid.merge(temp_df, on = 'quadkey', how = 'outer')
gridded_csv['measurement_name'].fillna(0, inplace=True)
gridded_csv.crs = '+proj=longlat +ellps=WGS84 +datum=WGS84 +no_defs'
temp_gridded = temp_gridded.set_index(['time', 'latitude', 'longitude'])
```

The above piece of code shows the key functions implemented to perform the join, clean the data by setting to 0 all the NaN values, set the CRS and the MultiIndex required for the realization of the xarray.DataSet.

Next, the dataframe of the gridded data is converted into xarray and subsequently saved locally in NetCDF format, this is a key step for the uploading in ODC, since the NetCDF format is supported by the platform (csv not yet). This procedure is performed with the function provided by pandas and xarray libraries shown below.

```
csv_xarray = gridded_csv.to_xarray()
csv_xarray.to_netcdf(path_to_netcdf_folder/name_of_netcdf.nc)
```

3.2. Indexing data in ODC

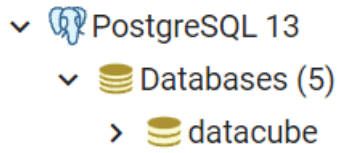
Once the csv has been grided and transformed into the correct file format, the gridData function proceeds with the preparation of the metadata Dataset document required for upload in ODC. The metadata document is stored in .yaml format and the document is made

up of a fixed structure to which some specific parameters are substituted by the function gridData, those parameters depend on that depend on the NetCDF file specification that we are going to upload. The produced metadata are stored in '/cubenv/dataset' folder and uploaded in ODC through an OS command launched directly by the function.

The following paragraph aims to clarify in detail the indexing of data in ODC, it is possible to divide the procedure in three fundamental steps: the database setup, the product definition and the dataset indexing.

3.2.1. Database initialization

Fundamental requirement for using datacube is to have correctly configured a PostgreSQL installation, that is a graphical interface that allows to manage PostgreSQL databases in a simplified way. First step of the process il to access pgAdmin and create a new database named 'datacube':



In our configuration the owner of the database is the super-user postgres.

Next step is the creation of a database configuration file that must be named 'datacube.conf' and must be stored in ~/.datacube.conf.

```
[datacube]
db_database: datacube

# A blank host will use a local socket. Specify a hostname (such as localhost) to use TCP.
db_hostname: localhost

# Credentials are optional: you might have other Postgres authentication configured.
# The default username otherwise is the current user id.
db_username: postgres
db_password: postgres

```

The fields 'db_username' and 'db_password' must be correctly filled with the user credentials.

Once having create the configuration file the user must navigate through the terminal to the file location:

```
cd path/to/folder
```

And launch the following command:

```
datacube -v system init
```

The datacube system init tool tool can create and populate the Data Cube database schema.

This step is done only once, afterwards you will only need to navigate to the folder containing the configuration file in order to be able to launch the commands.

3.2.2. Product definition

Once ODC is correctly connected to the database, it is possible to start to load in some data. Loading data in ODC means recording the existence of and detailed metadata about the data into the index. None of the data itself is copied, moved or transformed. This is therefore a safe and fast process, the limitation of the process is that the data must be saved locally in the working environment.

Before the data itself can be added, a product describing the data must be created. A product consists of a .yaml file that provides a short name, a description, some basic source metadata and a list of measurements describing the type of data that will be contained in the Datasets of its type.

Following is shown the product realized for Milan Facebook population data:

```yaml
name: MILAN
description: FB for good data about population
metadata_type: eo3

metadata:
    product:
        name: MILAN
        format: NetCDF

storage:
    crs: epsg:4326
    resolution:
        latitude: 0.0038522769335609763
        longitude: 0.005493164062499015
    dimension_order: [time, latitude, longitude]

measurements:
    - name: 'n_crisis'
      dtype: float64
      nodata: -9999
      units: 'people'
```

Relevant fields:

- metadata: – format: specifies the format of stored files pointed by the Dataset, in our case the NetCDF produced according to the process described in paragraph 3.1.
- storage: - resolution: this field is not necessary, we decided to introduce it to speed up the dataset loading process through the load function, limiting the number of input parameters to be passed to the function.

- storage: - dimesion_order: specifies the dimensions of the dataset, must match with the names of the index fields of the NetCDF files.

- measurements: measurements is an ordered list of data, which specify a name and some aliases, a data type or dtype, and some options extras including what type of units the measurement is in, a nodata. The names given to the measurements must match with the names of the fields of the NetCDF containing the corresponding information, in our case the field containing the required information is 'n_crisis' that specifies the number of people for each point.

To load the product in ODC, it is necessary to launch this command:

```
datacube product add path_to_product/product_name.yaml
```

Once having created the product we can start to indexing the datasets. Since this process must be done only once, it is not carried out by gridData function. The function only creates and uploads the datasets in ODC.

3.2.3. Dataset definition

Every dataset produced by gridData function requires a metadata document describing what the data represents and where it has come from, as well has what format it is stored in. At a minimum, you need the dimensions or fields you want to search by, such as lat, lon and time, but you can include any information you deem useful.

Dataset metadata are stored in .yaml format and are produced by gridData function. Following, an example of metadata is given, then, the metadata fields modified by the function are specified.

```yaml
$schema: https://schemas.opendatacube.org/dataset

id: 00000000-0000-0000-0000-202004050800

product:
  name: MILAN
  href: https://dataforgood.fb.com/
  format: NetCDF

crs: epsg:4326

geometry:
  type: Polygon
  coordinates: [[[ 8.995056152343800, 45.311597470877999],
                 [8.995056152343800, 45.627484179430269],
                 [9.549865722656120, 45.627484179430269],
                 [9.549865722656120, 45.311597470877999],
                 [ 8.995056152343800, 45.311597470877999]]]
  crs: epsg::4326

grids:
  default:
    shape: [102,83]
    transform: [0.005493164062498224, 0.0, 8.99230957031255,
                0.0, -0.0038522769335641825, 45.62941031789704,
                0.0, 0.0, 1.0]

lineage: {}

measurements:
  n_crisis:
    layer: n_crisis
    path: C:/git/FB_ODC_2021/netcdf_files/2020-04-050800.nc
    nodata: -9999

properties:
  odc:file_format: NetCDF
  datetime: 2020-04-05T08:00:00.000Z
```

Fields modified by the function:

- id: UUID of the dataset, consists in a hexadecimal number composed by 32 digits. It changes in each dataset, it can be randomply generated, but in our case we have decided to create it through gridData function and is composed by a fixed part (the first 20 digits are always 0) and the last 12 digits consists of the date and the time of the measurements.
- measurements: - path: path of the NetCDF file containing the measurements.
- properties: - datetime: timestamp of the measurements.

Constants values:

- product: -name: name of the product which the datasets belong to, see paragraph 3.2.2.

- geometry: - boundaries: border points of the reference grid realized for the data gridding, see chapter 2. All the points are inside this Polygon.
- grids: - shape: number of horizontal and vertical points of the grid.
- grids: - transformation: transformation matrix, required for the computation of the points. It is possible to retrieve it by launching the following commands on a random NetCDF file produced by the gridData function:

  ```
  src = rasterio.open(NetCDF)
  src.transform
  ```

  These commands return the first two rows of the matrix, the third row is set to [0, 0, 1]. The transformation matrix is the same for all the datasets belonging to the same product.
- measurements: information contained about measurements must match with the one provided in the product yaml file.

gridData produce for each dataset the corresponding metadata and stores it in './cubeenv/dataset/', then metadata are loaded in ODC with the following system command:

```
datacube dataset add path/cubeenv/dataset/name.yaml
```

The function executes the system command using the os library, this library works fine for Windows and Linux systems.

4. Data Analysis

Purpose of this last chapter is to clarify how it is possible to use the data present in the datacube, showing the methods for the selection and extraction of data, the structures used by datacube to manage them and describe the routines implemented in the notebook to perform data analysis, data visualization and producing meaningful plots.

The content of the following paragraph describes the methods, functions and routine implemented in Notebook-2.

4.1. Loading Data with datacube

Fundamental tool of datacube library is the function load, that allows to load data as an xarray.DataSet. Each measurement will be a data variable in the xarray.Dataset. In our study case, the obtained dataset has as dimension the values of time, latitude and longitude; the measurements carried as variable are the values corresponding to the average number of people present in a certain point (touple of latitude and longitude) in a specific epoch (time).

```
<xarray.Dataset>
Dimensions:      (time: 51, latitude: 83, longitude: 102)
Coordinates:
  * time         (time) datetime64[ns] 2020-04-05 2020-04-06 ... 2020-05-25
  * latitude     (latitude) float64 45.31 45.32 45.32 ... 45.62 45.62 45.63
  * longitude    (longitude) float64 8.995 9.001 9.006 ... 9.539 9.544 9.55
    spatial_ref  int32 4326
Data variables:
    n_crisis     (time, latitude, longitude) float64 0.0 20.65 ... -9.999e+03
Attributes:
    crs:           epsg:4326
    grid_mapping:  spatial_ref
<class 'xarray.core.dataset.Dataset'>
```

Arguments of the function are the values that will be used to query the database:

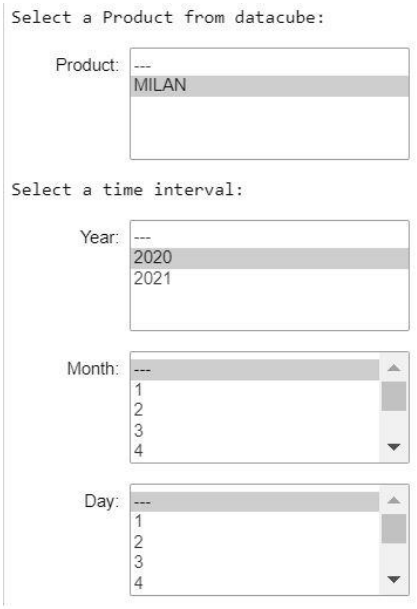
- Product name and measurements list.
- Dimensions of the data: we can specify spatial and time dimensions, according to the characteristics of the data we're interested in.
  Spatial dimension: can be specified using the longitude/latitude and x/y fields. The user can pass the boundaries of the area of interest, the CRS of this query is assumed to be WGS84/EPSG:4326 unless the crs field is supplied, even if the stored data is in another projection or the output_crs is specified. The dimensions longitude/latitude and x/y can be used interchangeably.
  Time dimension: specified using a tuple of datetime objects or strings with YYYY-MM-DD hh:mm:ss format.

In the presented notebook, the user can specify the time boundaries through a widget (thanks to the ipywidgets library), it was decided to neglect the spatial boundaries as a query variable, since the loaded data covers a limited space. The user has also the possibility to choose the product he's interested in between all the available products in the Database (in our case we have only one product e.g. MILAN).

The code used to create the widget for choosing the year is shown as an example. The key element is still the pandas DataFrame. The same procedure is applied for each variable.

```
year_options = ['2020', '2021']
year_options = pd.DataFrame(year_options, columns=['year'])
year = widgets.SelectMultiple(options=['---']+list(year_options['year'].unique()), description='Year:')
display(year)
```

The following figure shows how the widgets appear to the user.



Other relevant parameters of the function that can be specified are:

- *crs:* used to define the CRS of the coordinates in the query.
- *output_crs:* CRS of the returned data (inside the DataSet). If no CRS is supplied, the CRS of the stored data is used if available.
- *resolution:* tuple of spatial resolution of output data, expressed in coherence with the output CRS.

Once the data has been successfully loaded with the parameters specified by the user, the notebook provides an example of use the DataSet for statistical analysis. The following paragraphs show the methods implemented and the results produced.

4.2. Time Series of a statistical operator

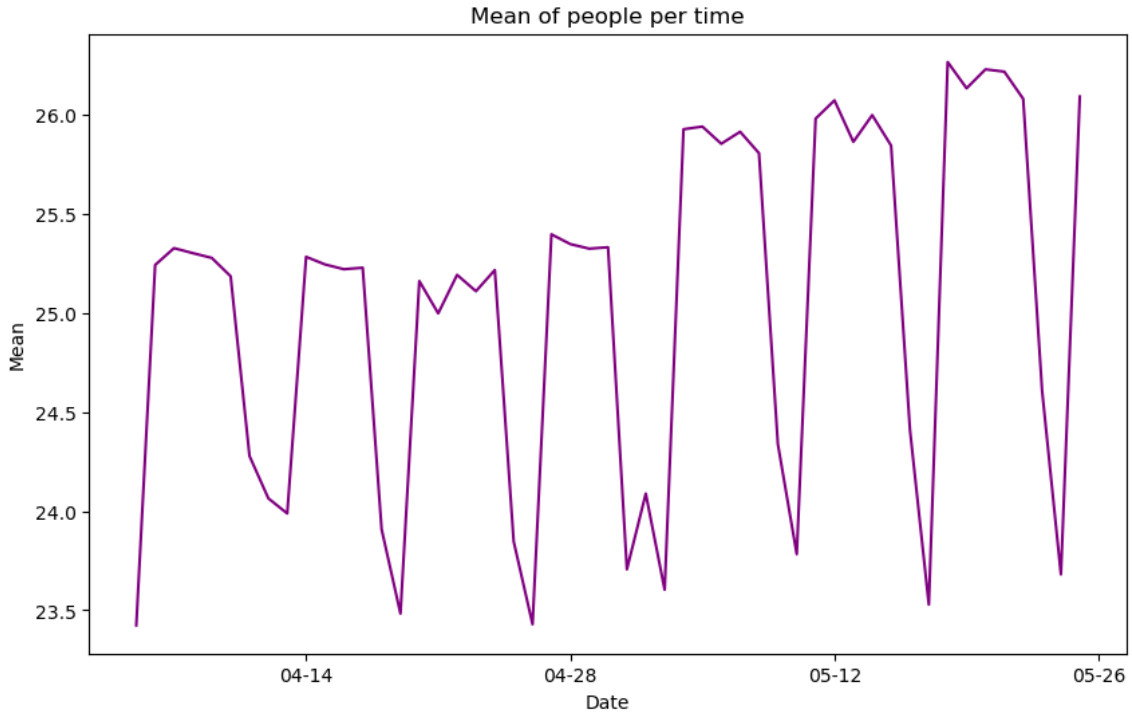
The first statistical analysis proposed is general and helps the user to get an idea of the data contained in the DataSet. The user, through a widget, is able to select an operator between mean, variance and median. The selected operator will be applied to all measurements in the DataSet for each available epoch in time dimension.

An ad-hoc function has been developed: takes as input the DataSet and the selected operator; it returns the time range of the dataset (all the epochs available in the DataSet) and, for each epoch, the mean or the variance or the median of the measurements in the same epoch.

```
def plot_time_op(operator, ds):
    time_list = ds.time.to_dataframe()
    time_list = time_list['time'].tolist()
    y = []
    for i in time_list:
        df_i = ds.sel(time=i).to_dataframe()
        df_i = df_i.reset_index().replace(to_replace=-9999, value=0)
        if operator == 'Mean':
            mean_i = df_i['n_crisis'].mean()
            y.append(mean_i)
        elif operator == 'Variance':
            std_i = df_i['n_crisis'].std()
            y.append(std_i)
        elif operator == 'Median':
            med_i = df_i['n_crisis'].median()
            y.append(med_i)
    return [time_list, y]
```

Fundamental element of the function is the tool *DataSet.sel (…)* which allows to quickly select the data of interest within the DataSet. Selection methods passed as arguments of the function are the dimensions of the dataset (in our case: *time, latitude* and *longitude*).

The obtained results are then plotted, in the following an example of plot is available: time ranges between April 4th and May 26th of 2020 and, for each day is evaluated the mean number of people per point.



4.3. Time Series of a selected point

The second solution proposed focuses on the plot of the measurements over time for a selected point. The user can specify the coordinates of the point he is interested in in two ways:

- Type manually the coordinates in a form.
- Choose a location from a widget, that propose a list ok key point for the city of Milano. Each location is related to a specific latitude and longitude, information about the key points are stored in a csv file and have been produced manually. The widget for the Milan's landmark selection appear to the user as follow:

Then, the DataSet and the coordinates are passed as argument to a developed function, that evaluates the nearest point of the grid to the passed coordinates and extract the value of the measurement for each epoch. The results are stored in a DataFrame that is returned by the function. The DataFrame has two columns: 'time' and 'measurement', these columns are then plotted in graph. An example id shown below, the time ranges between April 4th and May 26th of 2020 and the point selected corresponds to Duomo di Milano (selected through landmarks widget).

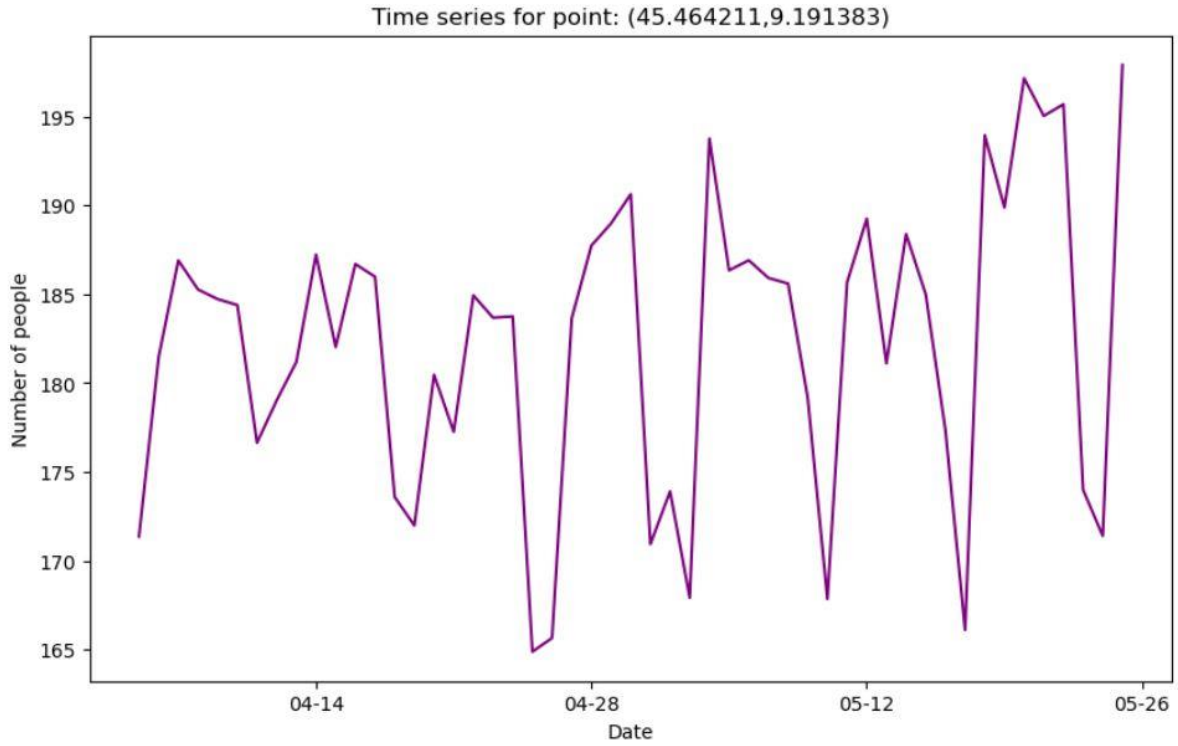
```
def temporal_series(ds, lat, lon):

    # Search for the closest latitude in the grid:
    all_lat = ds.latitude.to_dataframe()
    all_lat['delta_lat'] = abs(all_lat['latitude']-lat)
    mask_lat = all_lat[all_lat['delta_lat'] == all_lat['delta_lat'].min()]
    final_lat = mask_lat.latitude
    final_lat = final_lat.to_list()[0]

    # Search for the closest longitude in the grid
    all_lon = ds.longitude.to_dataframe()
    all_lon['delta_lon'] = abs(all_lon['longitude']-lon)
    mask_lon = all_lon[all_lon['delta_lon'] == all_lon['delta_lon'].min()]
    final_lon = mask_lon.longitude
    final_lon = final_lon.to_list()[0]

    df = ds.sel(latitude=final_lat, longitude=final_lon).to_dataframe().reset_index()
    df = df[['time', 'n_crisis']]
    df = df.replace(to_replace=-9999, value=0)
    return df
```

The first two pieces of code search for the closest point in the grid to the one passed as argument. The last one implements the *DataSet.sel(…)* function to extract the values of the measurement for the closest point.



4.4. Data visualization

Final part of the notebook focuses on the production of maps and graphical representation of the analysed data.

The user, through a widget, can select an operator between mean, variance and median. Once the operator has been selected, the DataSet and the operator are passed to a function that produce a grid, whose points matches with the grid of the DataSet. Each point of the grid contains the computation of the selected operator on the point measurements over the whole time-range.

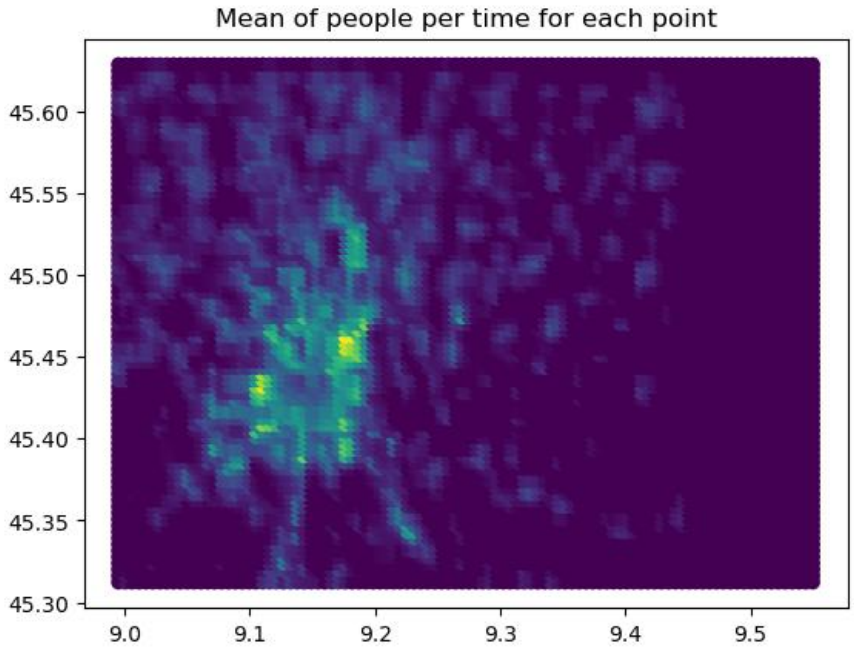
```
def grid_of_op(operator, ds):
    lon_list = ds.longitude.to_dataframe()
    lon_list = lon_list['longitude'].tolist()
    lat_list = ds.latitude.to_dataframe()
    lat_list = lat_list['latitude'].tolist()

    grid_op = pd.DataFrame(columns = ['latitude', 'longitude', 'value'])
    for i in lat_list:
        for j in lon_list:
            df_i = ds.sel(longitude=j, latitude=i).to_dataframe().reset_index().replace(to_replace=-9999, value=0)
            if operator == 'Mean':
                val = df_i['n_crisis'].mean()
            elif operator == 'Variance':
                val = df_i['n_crisis'].std()
            elif operator == 'Median':
                val = df_i['n_crisis'].median()
            row = pd.DataFrame([[i, j, val]], columns = ['latitude', 'longitude', 'value'])
            grid_op = grid_op.append(row)
    return grid_op
```

By iterating over all the points of the grid, the value of the operator selected for the same point in time is computed.

The function returns a DataFrame containing the following fields: [‘latitude’, ‘longitude’, ‘value’], where value is the computed mean/variance/median of the point’s measurements over time.

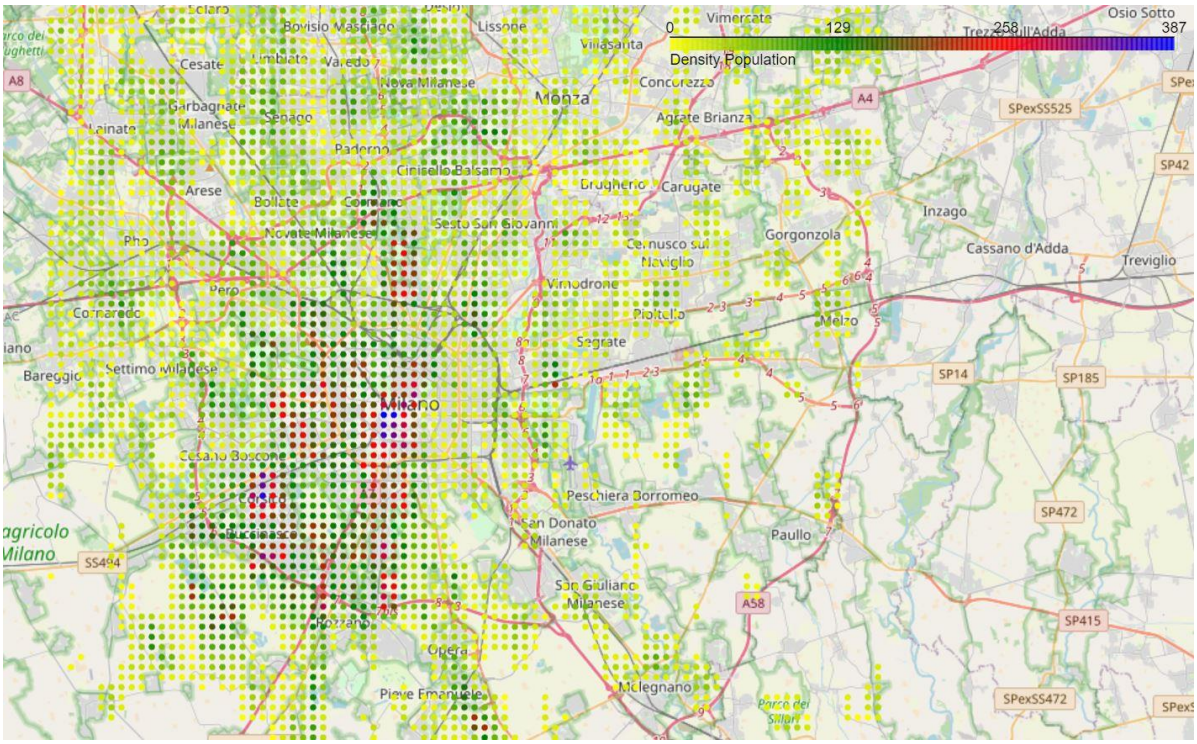
A first visualization of the data obtained is provided by matplotlib scatter tool:



It’s evident that the data visualization is poor and difficult to be read and understand. In order to give a better representation, we decided to use the *folium* library to build an interactive map of the obtained DataFrame.

Folium makes it easy to visualize data that’s been manipulated in Python on an interactive leaflet map. In our implementation we created a map with *folium.Map* tool, then, we added each point in the DataFrame to the map as *folium.Circle* object. Each point, according to the carried value of measurements, has a different colour, according to a colour scale defined with *branca* library.

The obtained result is shown below, the user can zoom in and out an move on the map. It is also possible to save the map as *.html* file.



In particular, in the above map, all the points with null values (that means absence of people) are discarded from the map.

4.4.1. Focus on Milan districts

The last example of possible use of data focuses on the city of Milan and on the distribution of the population among the different urban districts. The analysis is carried out on the DataFrame containing, for each point, the average number of people over a time range, produced with the function implemented for Data Visualization, but the implemented methods are sufficiently generic to be extended also to other spatial DataFrames.

In the example provided in the notebook, we worked with the geopandas library: a shapefile containing the boundaries of the municipalities of Milan was loaded as a geodataframe, the dataframe containing the points was also transformed into a geodataframe of points. Then, a spatial join is performed, between the polygons representing the municipalities and the points of the grid present within them: therefore, each point is associated with the municipality to which it belongs. In order to perform the spatial join, the two GeoDataFrame must have the same CRS (in our case we chose EPSG:4326). In the following we report the geopands' tool used to perform the special join:

```
join = gpd.sjoin(gdf, mun, how="inner", op="within").reset_index()
```

Where '*gdf*' is the geodataframe containing the points and '*mun*' is the geodataframe of Milan's municipalities.

In conclusion, for each municipality, the mean value of people for each district is evaluated. Then, the GeoDataFrame of the municipalities is plotted, the color of the municipality follows a color scale depending on the average number of people present. A basemap is added though *contextily* library.

5. Conclusions

The work carried out in the previous months shows that ODC is a suitable tool not only for the management of satellite data, but also for the collection and analysis of rasters containing other types of measurements. The most complicated part was the grid realization and the transformation of the CSVs into a format suitable to be managed by the ODC and the production of the metadata necessary for uploading. In our case we chose the NetCDF format, since, since the measurements are linked to points, it was the most suitable for our purpose, as well as the fastest to implement in a python environment.

We proposed to provide, in chapter 3, an in-depth analysis on the use of ODC through the datacube library, defining in a short guide how to initialize the database, create products and related datasets. The procedures used can be extended to create other types of metadata for other data as well.

Possible future continuations may be the extension of the work to the population data of greater extension (e.g. regional scale of Lombardy) or the development of functions for the analysis of movement range data.