lab_08

February 28, 2019

1 Clustering, spatial clustering, and geodemographics

This session covers statistical clustering of spatial observations. Many questions and topics are complex phenomena that involve several dimensions and are hard to summarize into a single variable. In statistical terms, we call this family of problems *multivariate*, as oposed to *univariate* cases where only a single variable is considered in the analysis. Clustering tackles this kind of questions by reducing their dimensionality -the number of relevant variables the analyst needs to look at- and converting it into a more intuitive set of classes that even non-technical audiences can look at and make sense of. For this reason, it is widely use in applied contexts such as policymaking or marketting. In addition, since these methods do not require many preliminar assumptions about the structure of the data, it is a commonly used exploratory tool, as it can quickly give clues about the shape, form and content of a dataset.

The basic idea of statistical clustering is to summarize the information contained in several variables by creating a relatively small number of categories. Each observation in the dataset is then assigned to one, and only one, category depending on its values for the variables originally considered in the classification. If done correctly, the exercise reduces the complexity of a multidimensional problem while retaining all the meaningful information contained in the original dataset. This is because, once classified, the analyst only needs to look at in which category every observation falls into, instead of considering the multiple values associated with each of the variables and trying to figure out how to put them together in a coherent sense. When the clustering is performed on observations that represent areas, the technique is often called geodemographic analysis.

Although there exist many techniques to statistically group observations in a dataset, all of them are based on the premise of using a set of attributes to define classes or categories of observations that are similar *within* each of them, but differ *between* groups. How similarity within groups and dissimilarity between them is defined and how the classification algorithm is operationalized is what makes techniques differ and also what makes each of them particularly well suited for specific problems or types of data. As an illustration, we will only dip our toes into one of these methods, K-means, which is probably the most commonly used technique for statistical clustering.

In the case of analysing spatial data, there is a subset of methods that are of particular interest for many common cases in Geographic Data Science. These are the so-called *regionalization* techniques. Regionalization methods can take also many forms and faces but, at their core, they all involve statistical clustering of observations with the additional constraint that observations need to be geographical neighbors to be in the same category. Because of this, rather than category, we will use the term *area* for each observation and *region* for each category, hence regionalization, the construction of regions from smaller areas.

```
In [1]: %matplotlib inline

        import seaborn as sns
        import pandas as pd
        import pysal as ps
        import geopandas as gpd
        import numpy as np
        import matplotlib.pyplot as plt
        from sklearn import cluster
```

```
/Users/dani/anaconda/envs/gds/lib/python3.6/site-packages/pysal/__init__.py:65: VisibleDeprecati
  ), VisibleDeprecationWarning)
```

1.1 Data

The dataset we will use in this occasion is an extract from the online website AirBnb. AirBnb is a company that provides a meeting point for people looking for an alternative to a hotel when they visit a city, and locals who want to rent (part of) their house to make some extra money. The website has a continuously updated listing of all the available properties in a given city that customers can check and book through. In addition, the website also provides a feedback mechanism by which both ends, hosts and guests, can rate their experience. Aggregating ratings from guests about the properties where they have stayed, AirBnb provides additional information for every property, such as an overall cleanliness score or an index of how good the host is at communicating with the guests.

The original data are provided at the property level and for the entire London. However, since the total number of properties is very large for the purposes of this notebook, they have been aggregated at the Middle Super Output Area (MSOA), a geographical unit created by the Office of National Statistics. Although the original source contains information for the Greater London, the vast majority of properties are located in Inner London, so the data we will use is restricted to that extent. Even in this case, not every polygon has at least one property. To avoid cases of missing values, the final dataset only contains those MSOAs with at least one property, so there can be average ratings associated with them.

Our goal in this notebook is to create a classification of areas (MSOAs) in Inner London based on the ratings of the AirBnb locations. This will allow us to create a typology for the geography of AirBnb in London and, to the extent the AirBnb locations can say something about the areas where they are located, the classification will help us understand the geography of residential London a bit better. One general caveat about the conclusions we can draw from an analysis like this one that derives from the nature of AirBnb data. On the one hand, this dataset is a good example of the kind of analyses that the data revolution is making possible as, only a few years ago, it would have been very hard to obtain a similarly large survey of properties with ratings like this one. On the other hand, it is important to keep in mind the kinds of biases that these data are subject to and thus the limitations in terms of generalizing findings to the general population. At any rate, this dataset is a great example to learn about statistical clustering of spatial observations, both in a geodemographic as well as in a regionalization.

As usual, before anything, let us set the paths to where we have downloaded the data:

```
In [2]: # This will be different on your computer and will depend on where
```

```
        # you have downloaded the files
        path = '../../../gds17_data/airbnb/'
```

IMPORTANT: the paths above might have look different in your computer. See this introductory notebook for more details about how to set your paths.

Note that, in this case, the data are provided as two separate files, so you will have to create a folder (for the example above, named `airbnb`) and place both there.

The main bulk of data is stored in `ilm_abb.geojson` (`ilm` for Inner London MSOAs, `abb` for AirBnb). Let us load it first:

In [3]:
```
        # Read GeoJSON file
        abb = gpd.read_file(path+'ilm_abb.geojson')
        # Manually set CRS (it might work without depending on
        # machine, but just in case)
        abb.crs = {'init': u'epsg:27700'}
        abb.info()
```

```
<class 'geopandas.geodataframe.GeoDataFrame'>
RangeIndex: 320 entries, 0 to 319
Data columns (total 16 columns):
accommodates                  320 non-null float64
bathrooms                     320 non-null float64
bedrooms                      320 non-null float64
beds                          320 non-null float64
number_of_reviews             320 non-null float64
reviews_per_month             320 non-null float64
review_scores_rating          320 non-null float64
review_scores_accuracy        320 non-null float64
review_scores_cleanliness     320 non-null float64
review_scores_checkin         320 non-null float64
review_scores_communication   320 non-null float64
review_scores_location        320 non-null float64
review_scores_value           320 non-null float64
property_count                320 non-null int64
MSOA_id                       320 non-null object
geometry                      320 non-null object
dtypes: float64(13), int64(1), object(2)
memory usage: 40.1+ KB
```

Note that, in comparison to previous datasets, this one is provided in a new format, `.geojson`. GeoJSON files are a plain text file (you can open it on any text editor and see its contents) that follows the structure of the JSON format, widely used to exchange information over the web, adapted for geographic data, hence the `geo` at the front. GeoJSON files have gained much popularity with the rise of web mapping and are quickly becoming a de-facto standard for small datasets because they are simple and easy to read in many different platforms. As you can see above, reading them in Python is exactly the same as reading a shapefile, for example.

Before we jump into exploring the data, one additional step that will come in handy down the line. Not every variable in the table is an attribute that we will want for the clustering. In

particular, we are interested in review ratings, so we will only consider those. Hence, let us first manually write them so they are easier to subset:

```
In [4]: ratings = ['review_scores_rating', 'review_scores_accuracy',
                   'review_scores_cleanliness', 'review_scores_checkin',
                   'review_scores_communication', 'review_scores_location',
                   'review_scores_value']
```

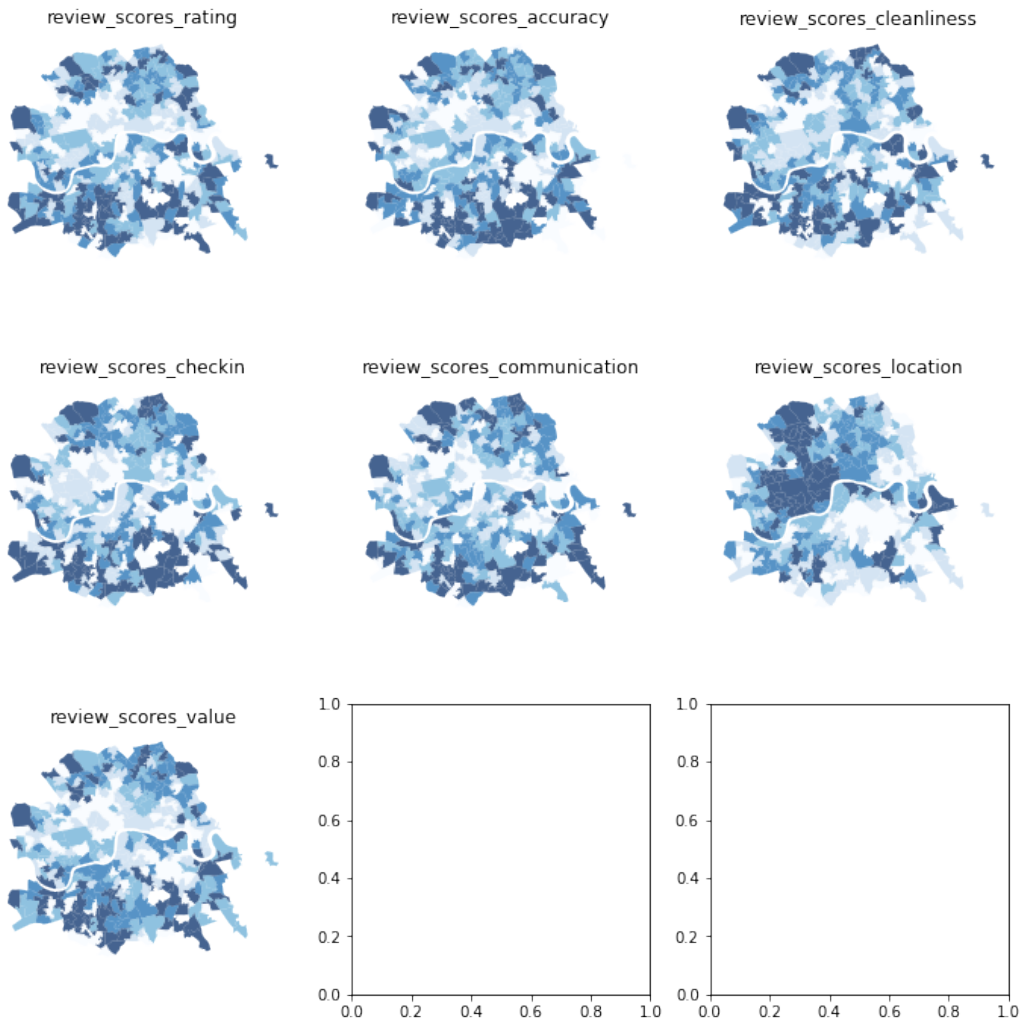
1.2 Getting to know the data

The best way to start exploring the geography of AirBnb ratings is by plotting each of them into a different map. This will give us a univariate perspective on each of the variables we are interested in.

Since we have many columns to plot, we will create a loop that generates each map for us and places it on a "subplot" of the main figure:

```
In [5]: # Create figure and axes (this time it's 9, arranged 3 by 3)
        f, axs = plt.subplots(nrows=3, ncols=3, figsize=(12, 12))
        # Make the axes accessible with single indexing
        axs = axs.flatten()
        # Start the loop over all the variables of interest
        for i, col in enumerate(ratings):
            # select the axis where the map will go
            ax = axs[i]
            # Plot the map
            abb.plot(column=col, ax=ax, scheme='Quantiles', \
                     linewidth=0, cmap='Blues', alpha=0.75)
            # Remove axis clutter
            ax.set_axis_off()
            # Set the axis title to the name of variable being plotted
            ax.set_title(col)
        # Display the figure
        plt.show()
```

```
/Users/dani/anaconda/envs/gds/lib/python3.6/site-packages/scipy/stats/stats.py:1713: FutureWarni
  return np.add.reduce(sorted[indexer] * weights, axis=axis) / sumval
```



Before we delve into the substantive interpretation of the map, let us walk through the process of creating the figure above, which involves several subplots inside the same figure:

- First (L. 2) we set the number of rows and columns we want for the grid of subplots.
- Then we *unpack* the grid into a flat list (array) for the axes of each subplot that we can loop over (L. 4).
- At this point, we set up a `for` loop (L. 6) to plot a map in each of the subplots.
- Within the loop (L. 6-14), we extract the axis (L. 8), plot the choropleth on it (L. 10) and style the map (L. 11-14).
- Display the figure (L. 16).

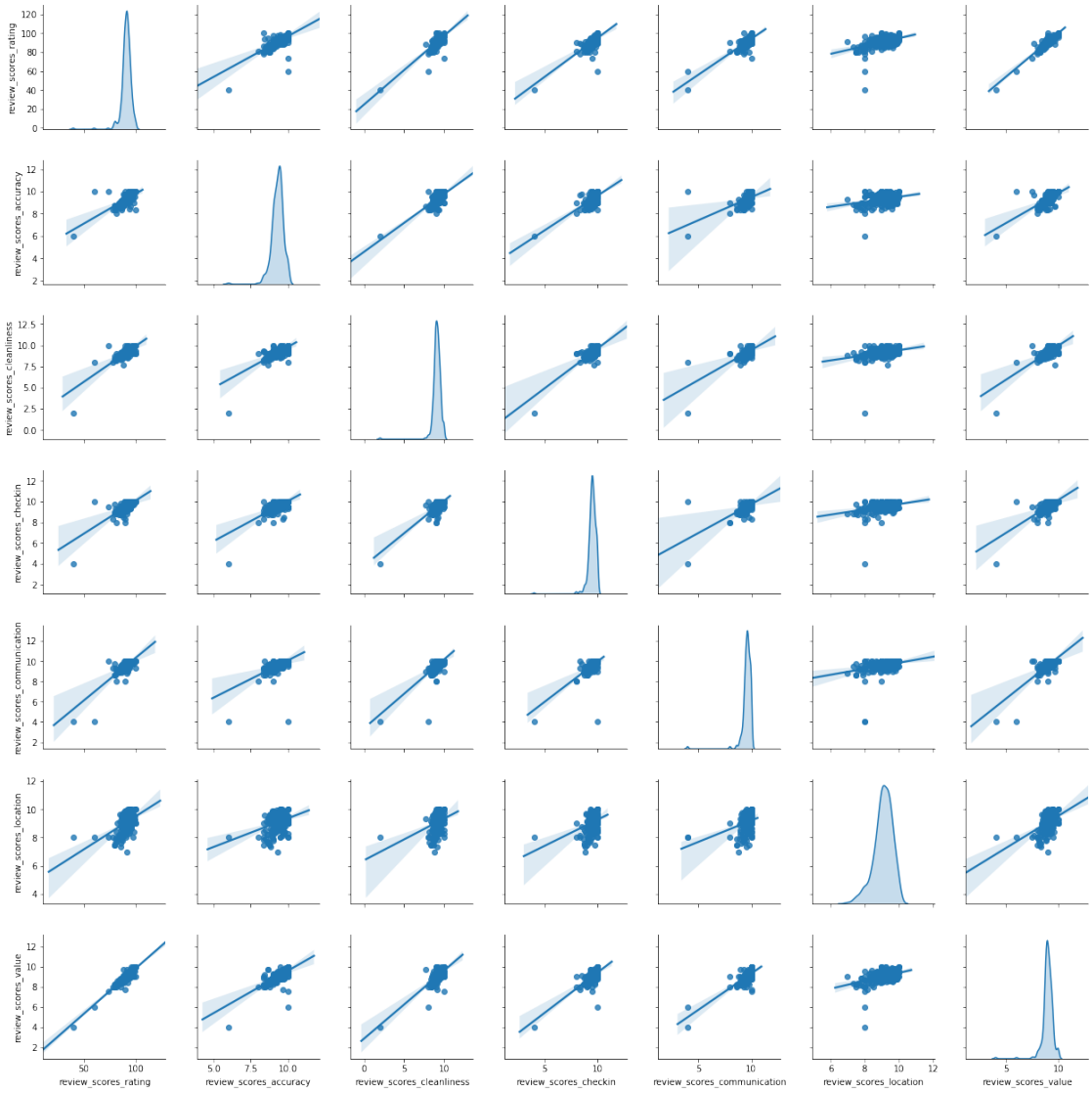
As we can see, there is substantial variation in how the ratings for different aspects are distributed over space. While variables like the overall value (`review_scores_value`) or the communication (`review_scores_communication`) tend to higher in peripheral areas, others like the location score (`review_scores_location`) are heavily concentrated in the city centre.

Even though we only have seven variables, it is very hard to "mentally overlay" all of them to

come up with an overall assessment of the nature of each part of London. For bivariate correlations, a useful tool is the correlation matrix plot, available in `seaborn`:

```
In [6]: _ = sns.pairplot(abb[ratings], kind='reg', diag_kind='kde')
```

```
/Users/dani/anaconda/envs/gds/lib/python3.6/site-packages/scipy/stats/stats.py:1713: FutureWarni
  return np.add.reduce(sorted[indexer] * weights, axis=axis) / sumval
```



[Optional exercise]

Explore the help and the `seaborn` tutorial (find it on Google) for the function `pairplot` and experiment with some of the parameters. For example, recreate the figure above replacing the KDE plots with histograms.

This is helpful to consider uni and bivariate questions such as: *what is the relationship between the overall (`rating`) and location scores?* (Positive) *Are the overall ratings more correlated with location or with cleanliness?* (Cleanliness) However, sometimes, this is not enough and we are interested in more sophisticated questions that are truly multivariate and, in these cases, the figure above cannot help us. For example, it is not straightforward to answer questions like: *what are the main characteristics of the South of London? What areas are similar to the core of the city? Are the East and West of London similar in terms of the kind of AirBnb properties you can find in them?* For these kinds of multi-dimensional questions -involving multiple variables at the same time- we require a truly multidimensional method like statistical clustering.

1.3 An AirBnb geodemographic classification of Inner London using K-means

A geodemographic analysis involves the classification of the areas that make up a greographical map into groups or categories of observations that are similar within each other but different between them. The classification is carried out using a statistical clustering algorithm that takes as input a set of attributes and returns the group ("labels" in the terminology) each observation belongs to. Depending on the particular algorithm employed, additional parameters, such as the desired number of clusters employed or more advanced tuning parameters (e.g. bandwith, radius, etc.), also need to be entered as inputs. For our geodemographic classification of AirBnb ratings in Inner London, we will use one of the most popular clustering algorithms: K-means. This technique only requires as input the observation attributes and the final number of groups that we want it to cluster the observations into. In our case, we will use five to begin with as this will allow us to have a closer look into each of them.

Although the underlying algorithm is not trivial, running K-means in Python is fairly straightforward thanks to `scikit-learn`. Similar to the extensive set of available algorithms in the library, its computation is a matter of two lines of code. First, we need to specify the parameters in the `KMeans` method (which is part of `scikit-learn`'s `cluster` submodule). Note that, at this point, we do not even need to pass the data:

```
In [7]: kmeans5 = cluster.KMeans(n_clusters=5)
```

This sets up an object that holds all the parameters required to run the algorithm. In our case, we only passed the number of clusters, but there are several other ones set by default:

```
In [8]: kmeans5
```

```
Out[8]: KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
            n_clusters=5, n_init=10, n_jobs=1, precompute_distances='auto',
            random_state=None, tol=0.0001, verbose=0)
```

To actually run the algorithm on the attributes, we need to call the `fit` method in `kmeans13`:

```
In [9]: # This line is required to obtain the same results always
        np.random.seed(1234)
        # Run the clustering algorithm
        k5cls = kmeans5.fit(abb[ratings])
```

The `k5cls` object we have just created contains several components that can be useful for an analysis. For now, we will use the labels, which represent the different categories in which we have grouped the data. Remember, in Python, life starts at zero, so the group labels go from zero to four. Labels can be extracted as follows:

```
In [10]: k5cls.labels_
```

```
Out[10]: array([3, 0, 4, 0, 3, 0, 4, 4, 0, 0, 3, 0, 4, 3, 0, 0, 4, 4, 0, 3, 3, 3,
                3, 3, 3, 3, 3, 3, 3, 4, 0, 3, 0, 0, 4, 0, 3, 4, 0, 0, 0, 0, 4, 0,
                0, 0, 0, 0, 3, 4, 0, 0, 0, 4, 0, 0, 0, 4, 2, 2, 3, 3, 3, 3, 4, 3,
                4, 0, 0, 0, 0, 0, 3, 3, 0, 4, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4,
                4, 4, 0, 0, 0, 4, 0, 0, 3, 0, 3, 0, 3, 4, 3, 0, 0, 0, 0, 0, 0, 0,
                0, 0, 0, 3, 0, 3, 3, 3, 3, 0, 3, 3, 0, 0, 3, 4, 3, 0, 0, 3, 3, 0,
                3, 0, 0, 3, 4, 4, 4, 4, 3, 4, 3, 4, 4, 0, 4, 4, 3, 0, 0, 0, 0, 0,
                4, 3, 4, 2, 0, 1, 0, 3, 2, 2, 2, 4, 3, 0, 0, 3, 0, 4, 4, 3, 0, 4,
                2, 4, 0, 0, 0, 3, 4, 4, 3, 0, 1, 4, 0, 0, 0, 0, 0, 4, 4, 3, 0, 3,
                3, 3, 0, 3, 3, 3, 3, 0, 3, 0, 2, 3, 4, 0, 4, 3, 4, 4, 4, 0, 4, 0,
                4, 0, 0, 0, 3, 0, 3, 3, 3, 4, 3, 3, 3, 3, 2, 0, 0, 0, 3, 2, 3, 3,
                3, 3, 3, 3, 0, 4, 0, 0, 3, 0, 4, 0, 3, 4, 0, 4, 0, 0, 4, 4, 4, 0,
                4, 4, 0, 0, 2, 0, 4, 4, 0, 4, 4, 4, 4, 0, 4, 3, 4, 2, 0, 3, 4, 0,
                3, 3, 2, 3, 0, 0, 3, 2, 3, 3, 3, 0, 3, 3, 3, 3, 3, 0, 0, 0, 3, 0,
                3, 0, 3, 0, 0, 3, 4, 4, 4, 3, 0, 4], dtype=int32)
```

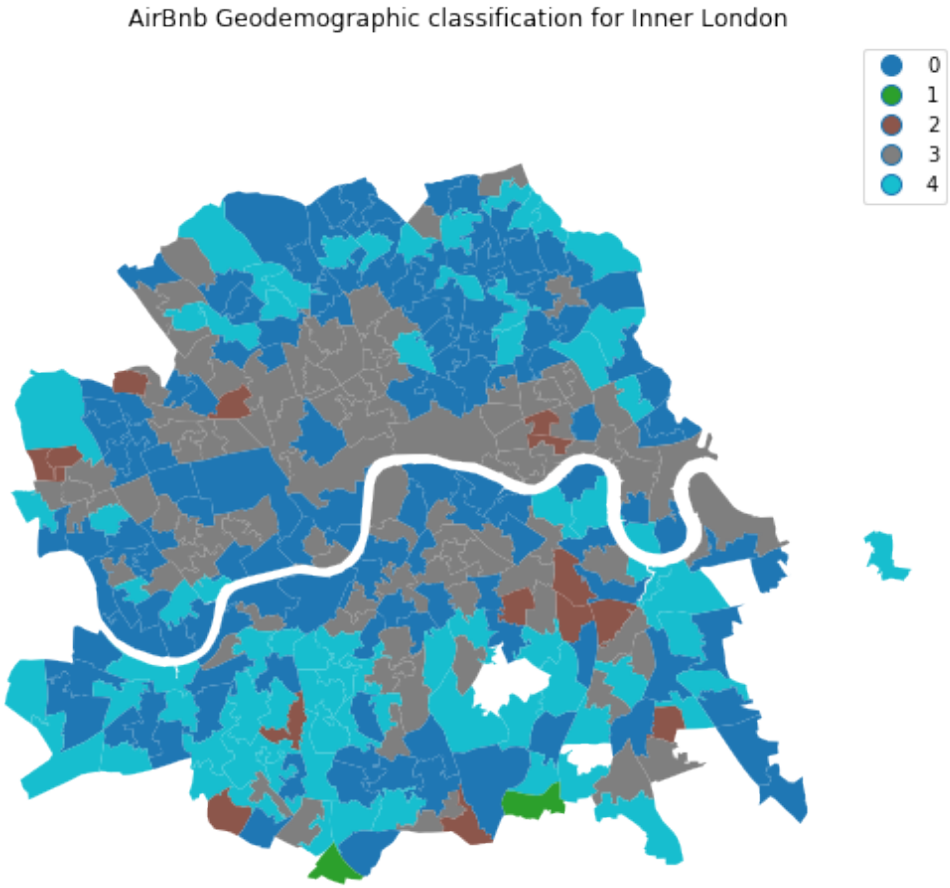
Each number represents a different category, so two observations with the same number belong to same group. The labels are returned in the same order as the input attributes were passed in, which means we can append them to the original table of data as an additional column:

```
In [11]: abb['k5cls'] = k5cls.labels_
```

1.3.1 Mapping the categories

To get a better understanding of the classification we have just performed, it is useful to display the categories created on a map. For this, we will use a unique values choropleth, which will automatically assign a different color to each category:

```
In [12]: # Setup figure and ax
         f, ax = plt.subplots(1, figsize=(9, 9))
         # Plot unique values choropleth including a legend and with no boundary lines
         abb.plot(column='k5cls', categorical=True, legend=True, linewidth=0, ax=ax)
         # Remove axis
         ax.set_axis_off()
         # Keep axes proportionate
         plt.axis('equal')
         # Add title
         plt.title('AirBnb Geodemographic classification for Inner London')
         # Display the map
         plt.show()
```



The map above represents the geographical distribution of the five categories created by the K-means algorithm. A quick glance shows a strong spatial structure in the distribution of the colors: group three (brown) is mostly found in the city centre and barely in the periphery, while group two (orange) is the opposite. Group zero (red) is an intermediate one, while group three (brown) and one (green) are much smaller, containing only a small number of observations.

1.3.2 Exploring the nature of the categories

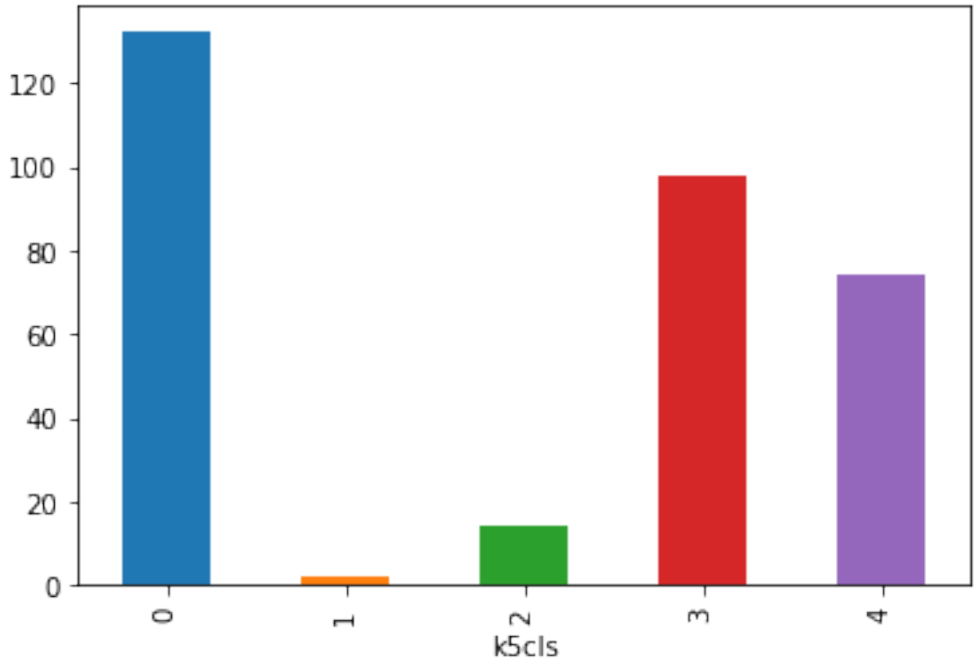
Once we have a sense of where and how the categories are distributed over space, it is also useful to explore them statistically. This will allow us to characterize them, giving us an idea of the kind of observations subsumed into each of them. As a first step, let us find how many observations are in each category. To do that, we will make use of the `groupby` operator introduced before, combined with the function `size`, which returns the number of elements in a subgroup:

```
In [13]: k5sizes = abb.groupby('k5cls').size()
         k5sizes
```

```
Out[13]: k5cls
         0    132
         1      2
         2     14
         3     98
         4     74
         dtype: int64
```

The `groupby` operator groups a table (`DataFrame`) using the values in the column provided (`k5cls`) and passes them onto the function provided aftwerards, which in this case is `size`. Effectively, what this does is to groupby the observations by the categories created and count how many of them each contains. For a more visual representation of the output, a bar plot is a good alternative:

```
In [14]: _ = k5sizes.plot.bar()
```



As we suspected from the map, groups varying sizes, with groups zero, three and four being over 75 observations each, and one and two being under twenty.

In order to describe the nature of each category, we can look at the values of each of the attributes we have used to create them in the first place. Remember we used the average ratings on many aspects (cleanliness, communication of the host, etc.) to create the classification, so we can begin by checking the average value of each. To do that in Python, we will rely again on the `groupby` operator but, in this case, we will combine it with the function `mean`:

```
In [15]: # Calculate the mean by group
         k5means = abb.groupby('k5cls')[ratings].mean()
```

```
# Show the table transposed (so it's not too wide)
k5means.T
```

```
Out[15]: k5cls                                0     1          2          3          4
         review_scores_rating         91.959366  50.0  80.693227  88.381725  95.878575
         review_scores_accuracy        9.394764   8.0   8.688363   9.099590   9.619290
         review_scores_cleanliness     9.228806   5.0   8.634376   8.927268   9.483036
         review_scores_checkin         9.575746   7.0   8.997941   9.364621   9.813744
         review_scores_communication   9.664774   4.0   9.034868   9.466938   9.856405
         review_scores_location        9.175812   8.0   7.944803   8.995572   9.300086
         review_scores_value           9.084736   5.0   8.178255   8.784636   9.428881
```

Or we can try to get a more comprehensive description and include also the quartiles and the standard deviation by calling the function `describe` instead of simply `mean`:

```
In [16]: # Calculate the summary by group
         k5desc = abb.groupby('k5cls')[ratings].describe()
         # Show the table
         k5desc
```

```
Out[16]:       review_scores_rating                                                  \
                              count       mean        std        min        25%
         k5cls
         0                    132.0  91.959366   1.014482  90.000000  91.165000
         1                      2.0  50.000000  14.142136  40.000000  45.000000
         2                     14.0  80.693227   2.670005  73.500000  79.796875
         3                     98.0  88.381725   1.367979  84.967742  87.666379
         4                     74.0  95.878575   1.718813  93.960000  94.462500

                                                  review_scores_accuracy            \
                      50%        75%         max                  count       mean
         k5cls
         0      91.972503  92.651961   93.750000                  132.0  9.394764
         1      50.000000  55.000000   60.000000                    2.0  8.000000
         2      80.760823  82.459524   83.900000                   14.0  8.688363
         3      88.666667  89.495968   90.096774                   98.0  9.099590
         4      95.306818  96.972222  100.000000                   74.0  9.619290

                  ...       review_scores_location             review_scores_value   \
                  ...                          75%        max                count
         k5cls    ...
         0        ...                     9.500000  10.000000                132.0
         1        ...                     8.000000   8.000000                  2.0
         2        ...                     8.000000   9.000000                 14.0
         3        ...                     9.369028   9.914286                 98.0
         4        ...                     9.526316  10.000000                 74.0


                    mean        std        min        25%        50%        75%         max
```

```
k5cls
0      9.084736  0.201922  8.428571  8.952238  9.089572  9.215587   9.666667
1      5.000000  1.414214  4.000000  4.500000  5.000000  5.500000   6.000000
2      8.178255  0.278528  7.500000  8.022727  8.185714  8.276786   8.642857
3      8.784636  0.250596  7.666667  8.685854  8.800000  8.909434   9.666667
4      9.428881  0.288177  8.857143  9.238971  9.410526  9.553031  10.000000

[5 rows x 56 columns]
```

However this approach quickly grows out of hand and the tables become very large to easily communicate any pattern. A good alternative is to visualize the distribution of values by category. The following optional extension shows how to transform the data so it is easy to create a fairly sophisticated plot that summarizes the table above.

[Optional extension]

To do this conveniently, we need to "tidy up" the table of values. Recall the meaning of *tidy* in the context of data: a dataset is tidy if every row represents an individual observation and every column a single variable. The table we want to plot to replace the summary above contains the following data:

```
In [17]: # Name (index) the rows after the category they belong
         to_plot = abb.set_index('k5cls')
         # Subset to keep only variables used in K-means clustering
         to_plot = to_plot[ratings]
         # Display top of the table
         to_plot.head()
```

```
Out[17]:        review_scores_rating  review_scores_accuracy  \
         k5cls
         3                 90.000000                9.244681
         0                 92.000000                9.500000
         4                 95.526316                9.684211
         0                 92.000000                9.400000
         3                 88.857143                9.000000

                review_scores_cleanliness  review_scores_checkin  \
         k5cls
         3                       9.265957               9.531915
         0                       9.500000              10.000000
         4                       9.526316               9.894737
         0                       9.000000               9.800000
         3                       9.071429               9.571429

                review_scores_communication  review_scores_location  \
         k5cls
         3                         9.542553                9.521277
         0                        10.000000               10.000000
```

```
4                      10.000000              9.947368
0                       9.600000              9.000000
3                       9.428571              9.500000

       review_scores_value
k5cls
3                 8.914894
0                 9.000000
4                 9.421053
0                 9.200000
3                 8.714286
```

Following the definition of "tidy data", the table above does not quality as tidy: the names of the columns are a variable in itself, the type of rating that the value represents. If we want to tidy up the table, the column names need to be squeezed into a single column -type of rating. This operation, in pandas is called to "stack" a table, and can easily be accomplished as follows:

```
In [18]: to_plot = to_plot.stack()
         to_plot.head()

Out[18]: k5cls
         3      review_scores_rating           90.000000
                review_scores_accuracy          9.244681
                review_scores_cleanliness       9.265957
                review_scores_checkin           9.531915
                review_scores_communication     9.542553
         dtype: float64
```

This returns a multi-indexed object. To keep things simple, we can convert it into a `DataFrame` by treating the index as additional columns:

```
In [19]: to_plot = to_plot.reset_index()
         to_plot.head()

Out[19]:    k5cls                      level_1          0
         0      3         review_scores_rating  90.000000
         1      3       review_scores_accuracy   9.244681
         2      3    review_scores_cleanliness   9.265957
         3      3        review_scores_checkin   9.531915
         4      3  review_scores_communication   9.542553
```

Finally, we can rename the columns to give them more meaningful names:

```
In [20]: to_plot = to_plot.rename(columns={'level_1': 'Rating', 0: 'Values'})
         to_plot.head()

Out[20]:    k5cls                       Rating     Values
         0      3         review_scores_rating  90.000000
         1      3       review_scores_accuracy   9.244681
         2      3    review_scores_cleanliness   9.265957
         3      3        review_scores_checkin   9.531915
         4      3  review_scores_communication   9.542553
```

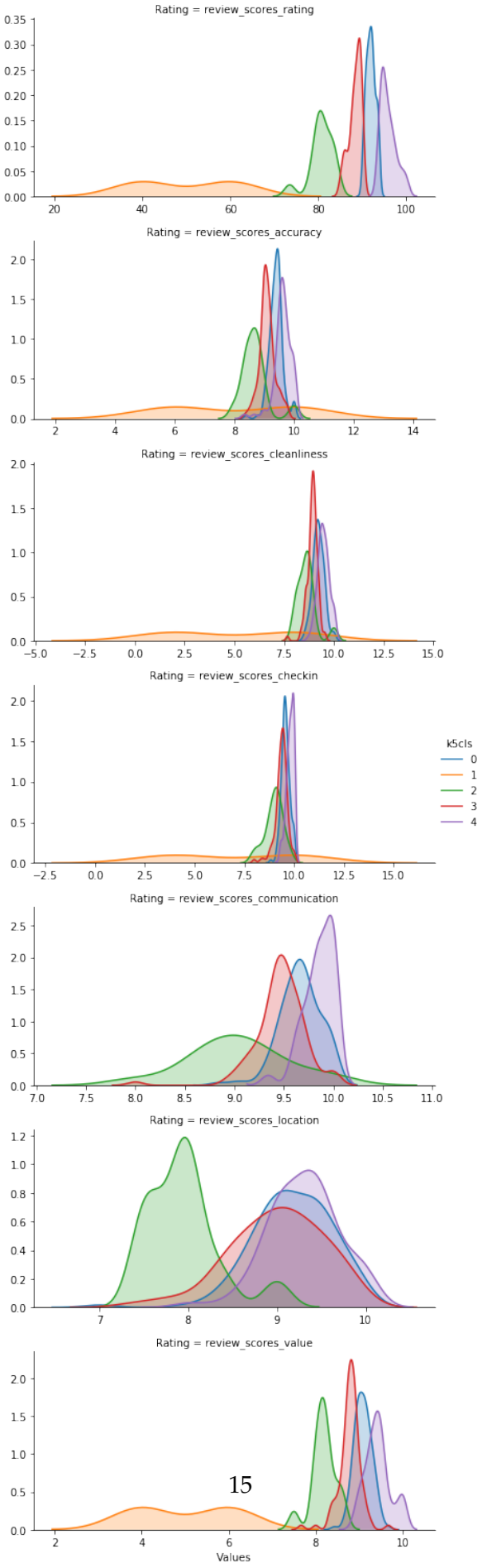
At this point, we are ready to visualize the distribution of values by type of rating by category. This is done in two steps:

1. Set up of the axis ("facet") to plot by variable.
2. Building of the plot

Let us show the code first and we will explain it afterwards:

```
In [21]: # Setup the facets
         facets = sns.FacetGrid(data=to_plot, row='Rating', hue='k5cls', \
                                sharey=False, sharex=False, aspect=2)
         # Build the plot as a `sns.kdeplot`
         _ = facets.map(sns.kdeplot, 'Values', shade=True).add_legend()
```

```
/Users/dani/anaconda/envs/gds/lib/python3.6/site-packages/scipy/stats/stats.py:1713: FutureWarni
  return np.add.reduce(sorted[indexer] * weights, axis=axis) / sumval
/Users/dani/anaconda/envs/gds/lib/python3.6/site-packages/statsmodels/nonparametric/kde.py:488:
  binned = fast_linbin(X, a, b, gridsize) / (delta * nobs)
/Users/dani/anaconda/envs/gds/lib/python3.6/site-packages/statsmodels/nonparametric/kdetools.py:
  FAC1 = 2*(np.pi*bw/RANGE)**2
/Users/dani/anaconda/envs/gds/lib/python3.6/site-packages/numpy/core/fromnumeric.py:83: RuntimeW
  return ufunc.reduce(obj, axis, dtype, out, **passkwargs)
```


Rating = review_scores_rating
Rating = review_scores_accuracy
Rating = review_scores_cleanliness
Rating = review_scores_checkin
Rating = review_scores_communication
Rating = review_scores_location
Rating = review_scores_value
k5cls
0
1
2
3
4
Values

Let us work through the logic of the lines of code above. First, the setup of facets (lines 2-3):

- What we are doing here is creating the "frames" onto which we will later plot. This part is crucial because here is where we specify the variables that we want to split the dataset into.
- We first pass the tidy dataset (`tidy`) to the argument `data`.
- Then we specify the argument `row` which controls how we will split the table `tidy` into plots that will take a different row each. In this case, we use the `Rating` column because we want a single plot for each type of rating (one for the overall rating, one for cleanliness, etc.).
- The next argument is `hue` which is another one to split the data into but, instead of defining differnt plots (as was the case for `row`), `hue` will split the data to create different elements in the same plot. In our case, this means that the split we specify here, by category (`k5cls`), will create different kernel densities within the same plot.
- The arguments `sharex` and `sharey`, which are set to `False`, imply that the scale of the horizontal and vertical axes respectively is not shared across plots. This makes different plots not directly comparable with each other, but generates graphs that make the best use of the available space, resulting in clearer visualizations.
- Finally, `aspect` controls the ratio of height and width of each plot. By setting it to 2, we create plots double long than high.

Once the "shell" of the figure is set up, the second part (line 5) provides the visual characteristics of each of the plots to be created. This is done by calling the function `map` on the `facets` object and specifying the following arguments:

- The first one is the function that we will use to plot the data. Because we want to visualize the distribution of values for each subgroup, we will use `sns.kdeplot`, which creates a kernel density estimation.
- The second one represents, on the data table specified before (`tidy` in our case), which values we want to plot. We pass the variable `Values`.
- Then is additional style arguments for `sns.kdeplot`. To color the area under the line, we set `shade` to `True`.
- Finally, we add an additional call, outside the `map` function to include a global legend with `add_legend`.

In substantive terms, the visualization shows the differences in each of the ratings by clustering group. Although in some cases, as in the checkin, these are not very large, others are more useful in stablishing differences across categories. For example, it singles out group one (green in the plots) as notably worse than the rest which, if we refer to the map we created above, corresponds with areas in green. Both the overall rating and the rating in terms of value (`review_scores_value`) establishes a hierarchy of categories by which the best is group four (grey areas in the map), then goes group zero (red in the map), followed by group three (brown areas in the map), and then group two (orange in the map). The visualization also makes clear that group one (green lines in the plots, green in the map) contains too little observations and too much noise to provide any meaningful information.

This concludes the section on geodemographics. As we have seen, the essence of this approach is to group areas based on a purely statistical basis: *where* each area is located is irrelevant for the label it receives from the clustering algorithm. In many contexts, this is not only permissible but even desirable, as the interest is to see if particular combinations of values are distributed over space in any discernible way. However, in other context, we may be interested in created groups of observations that follow certain spatial constraints. For that, we now turn into regionalization techniques.

[Optional exercise]

Create a new classification with 10 groups. Compare the output maps between the two solutions.

1.4 Regionalization algorithms

Regionalization is the subset of clustering techniques that impose a spatial constraint on the classification. In other words, the result of a regionalization algorithm contains areas that are spatially contiguous. Efectively, what this means is that these techniques aggregate areas into a smaller set of larger ones, called regions. In this context then, areas are *nested* within regions. Real world examples of this phenomenon includes counties within states or, in the UK, local super output areas (LSOAs) into middle super output areas (MSOAs). The difference between those examples and the output of a regionalization algorithm is that while the former are aggregated based on administrative principles, the latter follows a statistical technique that, very much the same as in the standard statistical clustering, groups together areas that are similar on the basis of a set of attributes. Only that now, such statistical clustering is spatially constrained.

As in the non-spatial case, there are many different algorithms to perform regionalization, and they all differ on details relating to the way they measure (dis)similarity, the process to regionalize, etc. However, same as above too, they all share a few common aspects. In particular, they all take a set of input attributes *and* a representation of space in the form of a binary spatial weights matrix. Depending on the algorithm, they also require the desired number of output regions into which the areas are aggregated.

To illustrate these concepts, we will run a regionalization algorithm on the AirBnb data we have been using. In this case, the goal will be to re-delineate the boundary lines of the Inner London boroughs following a rationale based on the different average ratings on AirBnb proeperties, instead of the administrative reasons behind the existing boundary lines. In this way, the resulting regions will represent a consistent set of areas that are similar with each other in terms of the ratings received.

1.4.1 Defining space formally

Very much in the same way as with ESDA techniques, regionalization methods require a formal representation of space that is statistics-friendly. In practice, this means that we will need to create a spatial weights matrix for the areas to be aggregated.

Technically speaking, this is the same process as we have seen before, thanks to `PySAL`. The difference in this case is that we did not begin with a shapefile, but with a GeoJSON. Fortunately,

PySAL supports the construction of spatial weights matrices "on-the-fly", that is from a table. This is a one-liner:

```
In [22]: w = ps.weights.Queen.from_dataframe(abb)
```

```
/Users/dani/anaconda/envs/gds/lib/python3.6/site-packages/pysal/weights/weights.py:186: UserWarn
  warnings.warn("There is one disconnected observation (no neighbors)")
/Users/dani/anaconda/envs/gds/lib/python3.6/site-packages/pysal/weights/weights.py:187: UserWarn
  warnings.warn("Island id: %s" % str(self.islands[0]))
```

1.4.2 Creating regions from areas

At this point, we have all the pieces needed to run a regionalization algorithm. For this example, we will use a spatially-constrained version of the agglomerative algorithm. This is a similar approach to that used above (the inner-workings of the algorithm are different however) with the difference that, in this case, observations can only be labelled in the same group if they are spatial neighbors, as defined by our spatial weights matrix `w`. The way to interact with the algorithm is very similar to that above. We first set the parameters:

```
In [23]: sagg13 = cluster.AgglomerativeClustering(n_clusters=13, connectivity=w.sparse)
         sagg13
```

```
Out[23]: AgglomerativeClustering(affinity='euclidean', compute_full_tree='auto',
                     connectivity=<320x320 sparse matrix of type '<class 'numpy.float64'>'
                  with 1688 stored elements in Compressed Sparse Row format>,
                     linkage='ward', memory=None, n_clusters=13,
                     pooling_func=<function mean at 0x111f6a048>)
```

And we can run the algorithm by calling `fit`:

```
In [24]: # This line is required to obtain the same results always
         np.random.seed(1234)
         # Run the clustering algorithm
         sagg13cls = sagg13.fit(abb[ratings])
```

```
/Users/dani/anaconda/envs/gds/lib/python3.6/site-packages/sklearn/cluster/hierarchical.py:193: U
  affinity='euclidean')
```

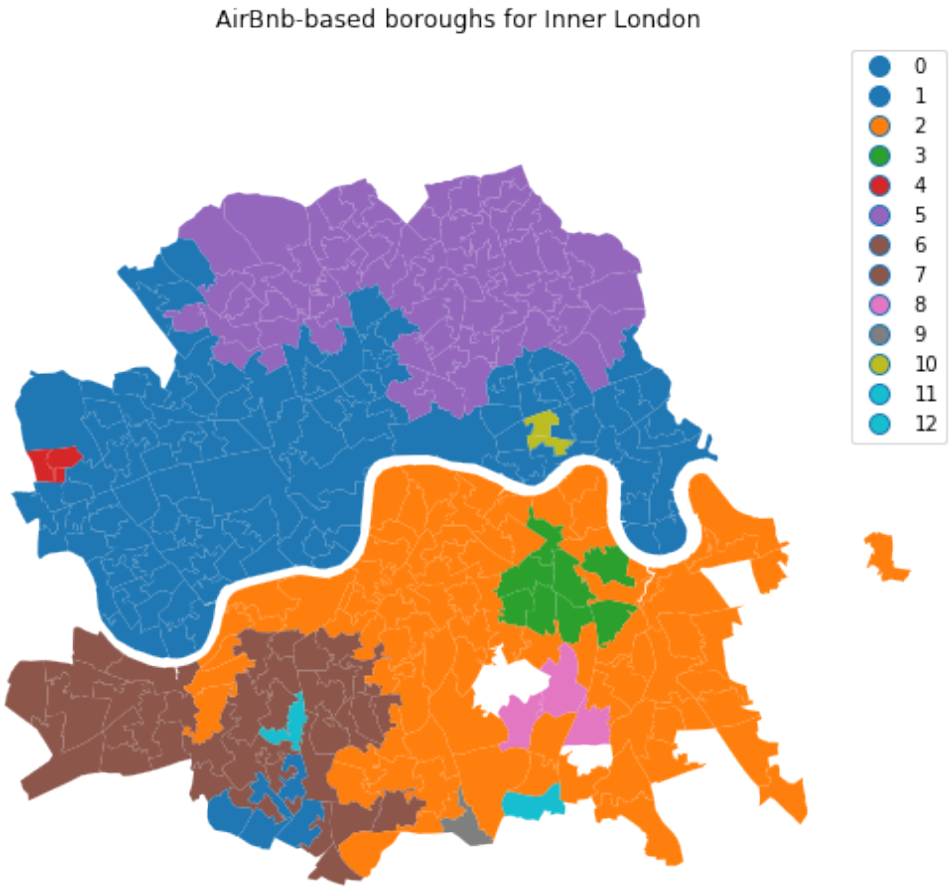
And then we append the labels to the table:

```
In [25]: abb['sagg13cls'] = sagg13cls.labels_
```

1.4.3 Mapping the resulting regions

At this point, the column `sagg13cls` is no different than `k5cls`: a categorical variable that can be mapped into a unique values choropleth. In fact the following code snippett is exactly the same as before, only replacing the name of the variable to be mapped and the title:

In [26]:
```
# Setup figure and ax
f, ax = plt.subplots(1, figsize=(9, 9))
# Plot unique values choropleth including a legend and with no boundary lines
abb.plot(column='sagg13cls', categorical=True, legend=True, linewidth=0, ax=ax)
# Remove axis
ax.set_axis_off()
# Keep axes proportionate
plt.axis('equal')
# Add title
plt.title('AirBnb-based boroughs for Inner London')
# Display the map
plt.show()
```



[Optional extension]

The map above gives a very clear impression of the boundary delineation of the algorithm. However, it is still based on the small area polygons. To create the new boroughs "properly", we need to dissolve all the polygons in each category into a single one. This is a standard GIS operation that is supported by `geopandas` and that can be easily actioned with the same `groupby` operator we used before. The only additional complication is that we need to wrap it into a separate function to be able to pass it on to `groupby`. We first the define the function `dissolve`:

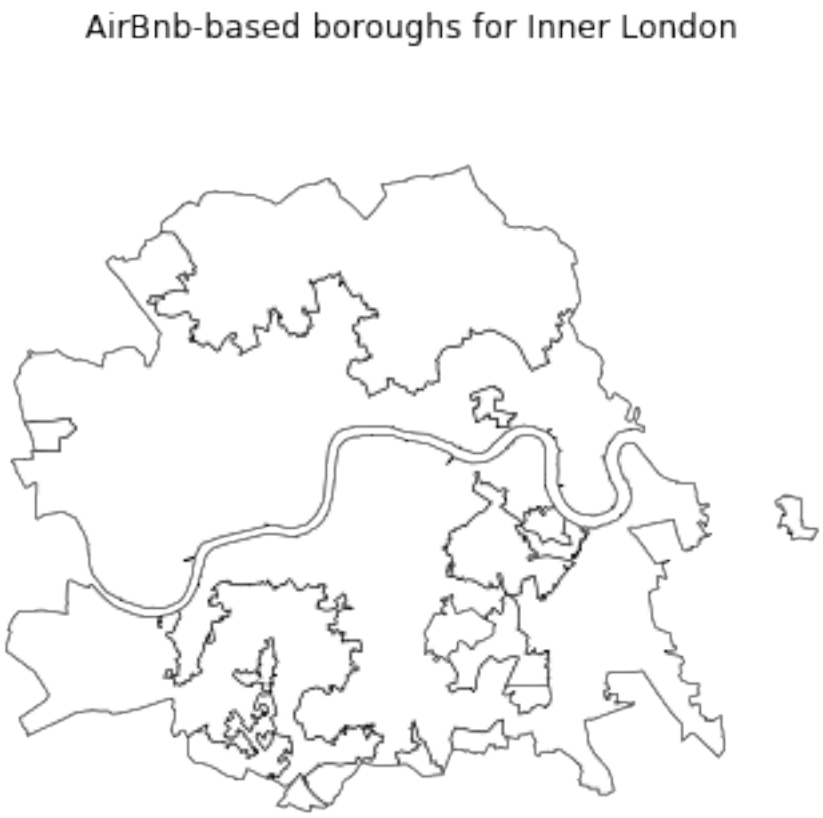
```
In [27]: def dissolve(gs):
             '''
             Take a series of polygons and dissolve them into a single one

             Arguments
             ---------
             gs        : GeoSeries
                         Sequence of polygons to be dissolved
             Returns
             -------
             dissolved : Polygon
                         Single polygon containing all the polygons in `gs`
             '''
             return gs.unary_union
```

The boundaries for the AirBnb boroughs can then be obtained as follows:

```
In [28]: # Dissolve the polygons based on `sagg13cls`
         abb_boroughs = gpd.GeoSeries(abb.groupby(abb['sagg13cls'])\
                                         .apply(dissolve),
                                      crs=abb.crs)
```

Which we can plot:

```
In [29]: # Setup figure and ax
         f, ax = plt.subplots(1, figsize=(6, 6))
         # Plot boundary lines
         abb_boroughs.plot(ax=ax, linewidth=0.5,\
                           facecolor='white', edgecolor='k')
         # Remove axis
         ax.set_axis_off()
         # Keep axes proportionate
         plt.axis('equal')
         # Add title
         plt.title('AirBnb-based boroughs for Inner London')
         # Display the map
         plt.show()
```



1.4.4 Comparison with the administrative London boroughs

The delineation we have just created can be compared with the administrative boundaries of the Inner London boroughs. These can be loaded up from the `geojson` file in the same way as we did with the original dataset:

```
In [30]: # Read GeoJSON file
         admin_boroughs = gpd.read_file(path+'london_boroughs.geojson')
         # Manually set CRS (it might work without depending on
         # machine, but just in case)
         admin_boroughs.crs = {'init': u'epsg:4326'}
         admin_boroughs.head()

Out[30]:                    name color  \
         0  Barking and Dagenham  blue
```

```
1                Barnet  blue
2                Bexley  blue
3                 Brent  blue
4               Bromley  blue

                                                 geometry
0  POLYGON ((0.1885882 51.5538749, 0.188310749438...
1  POLYGON ((-0.285151 51.6371097, -0.2851648 51...
2  POLYGON ((0.1341785938683765 51.51493117279971...
3  POLYGON ((-0.2654891397269057 51.5988648405349...
4  POLYGON ((-0.0033685 51.3468601, -0.0030238 51...
```

The table above contains *all* the boroughs in London. Since our data is focused on Inner London, we need the list of boroughs considered part of Inner London to subset it. Let us manually input it for use later:

```
In [31]: inner_bor_names = ['Camden', 'Greenwich', 'Hackney', 'Hammersmith and Fulham', \
                            'Islington', 'Kensington and Chelsea', 'Lambeth', 'Lewisham', \
                            'Southwark', 'Tower Hamlets', 'Wandsworth', 'Westminster', \
                            'City of London']
```

Subsetting the table is more easily done if we *index* the table on the names. Remember, by *indexing* we mean assigning one of the columns as the "name of the rows":

```
In [32]: # Index on the name of the boroughs
         admin_inner_boroughs = admin_boroughs.set_index('name')
         admin_inner_boroughs.head()
```

```
Out[32]:                       color                                           geometry
         name
         Barking and Dagenham  blue  POLYGON ((0.1885882 51.5538749, 0.188310749438...
         Barnet                blue  POLYGON ((-0.285151 51.6371097, -0.2851648 51...
         Bexley                blue  POLYGON ((0.1341785938683765 51.51493117279971...
         Brent                 blue  POLYGON ((-0.2654891397269057 51.5988648405349...
         Bromley               blue  POLYGON ((-0.0033685 51.3468601, -0.0030238 51...
```

Onced indexed on names, we *reindex* it to the list of inner boroughs. Reindexing means replacing the original rows by those with the index that we are passing. In this case, it really means we are subsetting the table to keep only those in Inner London, but reindexing can do many more things:

```
In [33]: admin_inner_boroughs = admin_inner_boroughs.reindex(inner_bor_names)
```

Finally, one more piece of housekeeping. Since the original file is expressed in raw latitude and longitude, it is convenient to project it to the same CRS as we have been using:

```
In [34]: # Projecting the dataset using the same CRS as the original abb table
         admin_inner_boroughs = admin_inner_boroughs.to_crs(abb.crs)
```

And displayed in a similar way as with the newly created ones:

```
In [35]: # Setup figure and ax
         f, ax = plt.subplots(1, figsize=(6, 6))
         # Plot boundary lines
         admin_inner_boroughs.plot(ax=ax, linewidth=0.5, \
                                   edgecolor='k', facecolor='white')
         # Remove axis
         ax.set_axis_off()
         # Keep axes proportionate
         plt.axis('equal')
         # Add title
         plt.title('Administrative boroughs for Inner London')
         # Display the map
         plt.show()
```

[Optional extension]

In order to more easily compare the administrative and the "regionalized" boundary lines, we can plot them side by side. To do this, we need to create a figure that contains two subplots. The rest of the logic is fairly similar to the usual plotting approach:

In [36]:
```
# Setup figure and an axis grid of one row and two columns
f, axs = plt.subplots(nrows=1, ncols=2, figsize=(12, 6))
# Split the axs object (a list) into two
ax1 = axs[0]
ax2 = axs[1]
                    # First axis
# Plot boundary lines
admin_inner_boroughs.plot(ax=ax1, linewidth=0.5, \
                          edgecolor='k', facecolor='white')
# Remove axis
ax1.set_axis_off()
# Keep axes proportionate
ax1.axis('equal')
# Add title
ax1.set_title('Administrative boroughs for Inner London')
                    # Second axis
# Plot boundary lines
abb_boroughs.plot(ax=ax2, linewidth=0.5,\
                  facecolor='white', edgecolor='k')
# Remove axis
ax2.set_axis_off()
# Keep axes proportionate
ax2.axis('equal')
# Add title
ax2.set_title('AirBnb defined boroughs for Inner London')
# Display the map
plt.show()
```

The code to produce the figure above differs from the usual "single maps" in the following key aspects:

- When we set up the figure with `plt.subplots`, we create two subplots and, to align them horizontally, we specify the number of rows and columns separately.
- The resulting axes object, `axs` is not a single one but a sequence of them. Because of this, we can't plot directly on `axs`, but we need to access the axis objects explicitly, hence the two following lines (4 and 5).
- The plotting on each of the axes then proceeds in the same way as if it was a single one, as this is about what goes into each of the two axes. All you have to make sure is the plotting is being assigned into the right axis object (`ax1` or `ax2`).

Looking at the figure, there are several differences between the two maps. The clearest one is that, while the administrative boundaries have a very balanced size (with the exception of the city of London), the regions created with the spatial agglomerative algorithm are very different in terms of size between each other. This is a consequence of both the nature of the underlying data and the algorithm itself. Substantively, this shows how, based on AirBnb, we can observe large areas that are similar and hence are grouped into the same region, while there also exist pockets with characteristics different enough to be assigned into a different region.

1.5 Optional exercise (if time permits)

Reproduce with the components of the IMD for Liverpool and compare the output with the distribution of the IMD scores. This involves the following steps:

- Read in the data.
- Create a geodemographic classification for Liverpool using K-means and all the components of the IMD (income, employment, education, health, crime, and housing). This is composed of:
  - Run K-means.
  - Extract the labels.
  - Plot the classes on a map.
  - Compare the map obtained from the geodemographic classification with the distribution of the IMD scores.
- Create a regionalization with the same attribute variables as for the geodemographic classification and producing 60 resulting observations. This will involve:
  - Create the weights for the LSOA geography.
  - Run the regionalisation algorithm (e.g. spatially constrained hierarchical clustering).
  - Extract the labels and plot them on a map.
  - Compare the resulting map with that of the MSOAs geography, which is available on the Census data pack under `Liverpool/shapefiles/Liverpool_msoa11.shp`.

This notebook, as well as the entire set of materials, code, and data included in this course are available as an open Github repository available at: https://github.com/darribas/gds18