> Geographic Data Science with PySAL and the pydata stack

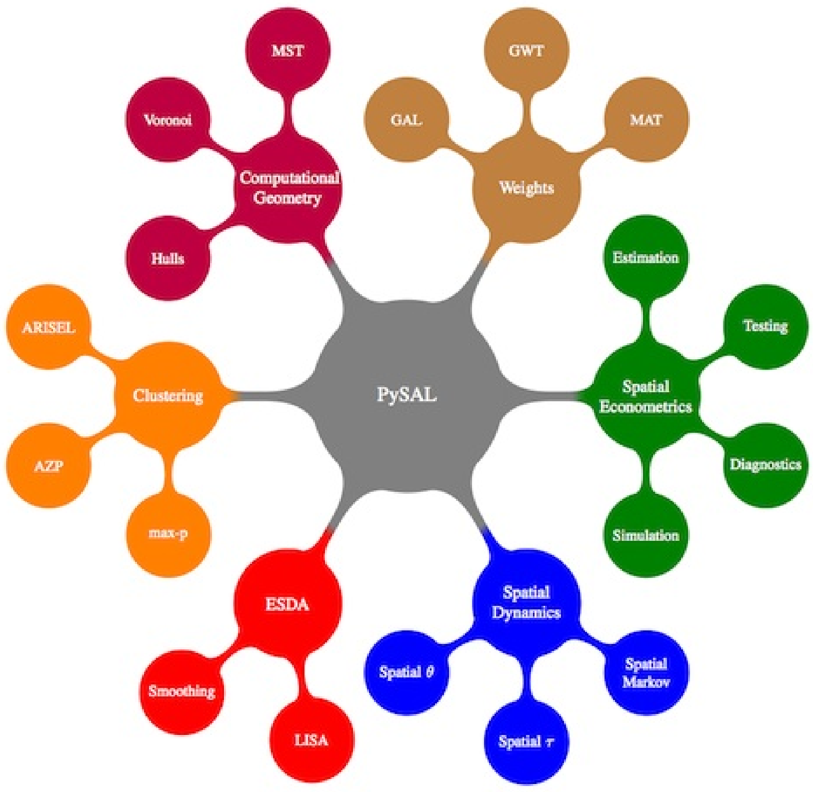
Sergio J. Rey

Dani Arribas-Bel

Table of Contents

Geographic Data Science with PySAL and the pydata stack

This two-part tutorial will first provide participants with a gentle introduction to Python for geospatial analysis, and an introduction to version `PySAL` 1.11 and the related eco-system of libraries to facilitate common tasks for Geographic Data Scientists. The first part will cover munging geo-data and exploring relations over space. This includes importing data in different formats (e.g. shapefile, GeoJSON), visualizing, combining and tidying them up for analysis, and will use libraries such as `pandas` , `geopandas` , `PySAL` , or `rasterio` . The second part will provide a gentle overview to demonstrate several techniques that allow to extract geospatial insight from the data. This includes spatial clustering and regression and point pattern analysis, and will use libraries such as `PySAL` , `scikit-learn` , or `clusterpy` . A particular emphasis will be set on presenting concepts through visualization, for which libraries such as `matplotlib` , `seaborn` , and `folium` will be used.



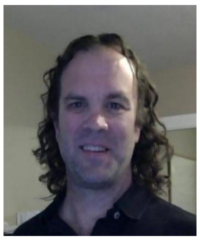
Geographic Data Science Lab

Distribution

> `[URL]` `[PDF]` `[EPUB]` `[MOBI]` `[IPYNB]`

License

About the authors


Sergio Rey is professor of geographical sciences and core faculty member of the GeoDa Center for Geospatial Analysis and Computation at the Arizona State University. His research interests include open science, spatial and spatio-temporal data analysis, spatial econometrics, visualization, high performance geocomputation, spatial inequality dynamics, integrated multiregional modeling, and regional science. He co-founded the Python Spatial Analysis Library (PySAL) in 2007 and continues to direct the PySAL project. Rey is a fellow of the spatial econometrics association and editor of the journal Geographical Analysis.

Dani Arribas-Bel is Lecturer in Geographic Data Science and member of the Geographic Data Science Lab at the University of Liverpool (UK). Dani is interested in undestanding cities as well as in the quantitative and computational methods required to leverage the power of the large amount of urban data increasingly becoming available. He is also part of the team of core developers of PySAL, the open-source library written in Python for spatial analysis. Dani regularly teaches Geographic Data Science and Python courses at the University of Liverpool and has designed and developed several workshops at different levels on spatial analysis and econometrics, Python and open source scientific computing.

Install howto

Install description here to be exposed on the website/ebook/pdf.

Outline

Part I

1. Software and Tools Installation (10 min)
2. Spatial data processing with PySAL (45 min)

   a. Input-output

   b. Visualization and Mapping

   c. Spatial weights
3. Exercise (10 min.)
4. ESDA with PySAL (45 min)

   a. Global Autocorrelation

   b. Local Autocorrelation

   c. Space-Time exploratory analysis
5. Exercise (10 min)

Part II

1. Point Patterns (30 min)

   a. Point visualization

   b. Kernel Density Estimation
2. Exercise (10 min)
3. Spatial clustering a (30 min)

   a. Geodemographic analysis

   b. Regionalization
4. Exercise (30 min)
5. Spatial Regression (30 min)

   a. Baseline (nonspatial) regression

   b. Exogenous and endogenous spatially lagged regressors

   c. Prediction performance of spatial models
6. Exercise (10 min)

Data

This tutorial makes use of a variety of data sources. Below is a brief description of each dataset as well as the links to the original source where the data was downloaded from. For convenience, we have repackaged the data and included them in the compressed file with the notebooks. You can download it here.

AirBnb listing for Austin (TX)

This dataset contains information for AirBnb properties for the area of Austin (TX). It is originally provided by Inside AirBnb. Same as the source, the dataset is released under a CC0 1.0 Universal License. You can see a summary of the dataset here.

Source: Inside AirBnb's extract of AirBnb locations in Austin (TX).

Path: `data/listings.csv.gz`

Austin Zipcodes

Boundaries for Zipcodes in Austin. The original source is provided by the City of Austin GIS Division.

Source: open data from the city of Austin [url]

Path: `data/Zipcodes.geojson`

Part I

Software and Tools Installation

Dependencies

Participants should have installed the following dependencies:

- Anaconda or MiniConda Python distributions for Python 2.7. See installation instructions on the links.
- `git`
- A `conda` environment loaded with all the dependencies can be installed by running the `pydata.sh` script available as part of the `envs` repository (Github link). To install it, follow these instructions:
    - Clone the repository on your machine:

      ```
      > git clone https://github.com/darribas/envs.git
      ```

    - Navigate into the folder:

      ```
      > cd envs
      ```

    - Run the script:

      ```
      > bash pydata.sh
      ```

Once installed, you need to activate the environment to run the notebooks. In Windows, open up PowerShell and type:

```
> activate pydata
```

And if you are on GNU/Linux or OSX:

```
> source activate pydata
```

Get started

Instructions to fire up a notebook here.

Spatial Data Processing

Notebook here.

```
%matplotlib inline

import pysal as ps
```

ESDA with `PySAL`

```
%matplotlib inline

import pysal as ps
```

```
/home/dani/anaconda/envs/pydata/lib/python2.7/site-packages/matplotlib/font_manager.py:273: UserWarning: Matplotlib i
s building the font cache using fc-list. This may take a moment.
  warnings.warn('Matplotlib is building the font cache using fc-list. This may take a moment.')
```

Part II

Point Patterns

> IPYNB
>
> **NOTE**: some of this material has been ported and adapted from "Lab 9" in Arribas-Bel (2016).

This notebook covers a brief introduction on how to visualize and analyze point patterns. To demonstrate this, we will use a dataset of all the AirBnb listings in the city of Austin (check the Data section for more information about the dataset).

Before anything, let us load up the libraries we will use:

```
%matplotlib inline

import numpy as np
import pandas as pd
import geopandas as gpd
import seaborn as sns
import matplotlib.pyplot as plt
import mplleaflet as mpll
```

```
/home/dani/anaconda/envs/pydata/lib/python2.7/site-packages/matplotlib/font_manager.py:273: UserWarning: Matplotlib i
s building the font cache using fc-list. This may take a moment.
  warnings.warn('Matplotlib is building the font cache using fc-list. This may take a moment.')
```

Data preparation

Let us first set the paths to the datasets we will be using:

```
# Adjust this to point to the right file in your computer
listings_link = '../data/listings.csv.gz'
```

The core dataset we will use is `listings.csv`, which contains a lot of information about each individual location listed at AirBnb within Austin:

```
lst = pd.read_csv(listings_link)
lst.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5835 entries, 0 to 5834
Data columns (total 92 columns):
id                                  5835 non-null int64
listing_url                         5835 non-null object
scrape_id                           5835 non-null int64
last_scraped                        5835 non-null object
name                                5835 non-null object
summary                             5373 non-null object
space                               4475 non-null object
description                         5832 non-null object
experiences_offered                 5835 non-null object
neighborhood_overview               3572 non-null object
notes                               2413 non-null object
transit                             3492 non-null object
thumbnail_url                       5542 non-null object
medium_url                          5542 non-null object
picture_url                         5835 non-null object
xl_picture_url                      5542 non-null object
host_id                             5835 non-null int64
host_url                            5835 non-null object
```

```
host_name                           5820 non-null object
host_since                          5820 non-null object
host_location                       5810 non-null object
host_about                          3975 non-null object
host_response_time                  4177 non-null object
host_response_rate                  4177 non-null object
host_acceptance_rate                3850 non-null object
host_is_superhost                   5820 non-null object
host_thumbnail_url                  5820 non-null object
host_picture_url                    5820 non-null object
host_neighbourhood                  4977 non-null object
host_listings_count                 5820 non-null float64
host_total_listings_count           5820 non-null float64
host_verifications                  5835 non-null object
host_has_profile_pic                5820 non-null object
host_identity_verified              5820 non-null object
street                              5835 non-null object
neighbourhood                       4800 non-null object
neighbourhood_cleansed              5835 non-null int64
neighbourhood_group_cleansed        0 non-null float64
city                                5835 non-null object
state                               5835 non-null object
zipcode                             5810 non-null float64
market                              5835 non-null object
smart_location                      5835 non-null object
country_code                        5835 non-null object
country                             5835 non-null object
latitude                            5835 non-null float64
longitude                           5835 non-null float64
is_location_exact                   5835 non-null object
property_type                       5835 non-null object
room_type                           5835 non-null object
accommodates                        5835 non-null int64
bathrooms                           5789 non-null float64
bedrooms                            5829 non-null float64
beds                                5812 non-null float64
bed_type                            5835 non-null object
amenities                           5835 non-null object
square_feet                         302 non-null float64
price                               5835 non-null object
weekly_price                        2227 non-null object
monthly_price                       1717 non-null object
security_deposit                    2770 non-null object
cleaning_fee                        3587 non-null object
guests_included                     5835 non-null int64
extra_people                        5835 non-null object
minimum_nights                      5835 non-null int64
maximum_nights                      5835 non-null int64
calendar_updated                    5835 non-null object
has_availability                    5835 non-null object
availability_30                     5835 non-null int64
availability_60                     5835 non-null int64
availability_90                     5835 non-null int64
availability_365                    5835 non-null int64
calendar_last_scraped               5835 non-null object
number_of_reviews                   5835 non-null int64
first_review                        3827 non-null object
last_review                         3829 non-null object
review_scores_rating                3789 non-null float64
review_scores_accuracy              3776 non-null float64
review_scores_cleanliness           3778 non-null float64
review_scores_checkin               3778 non-null float64
review_scores_communication         3778 non-null float64
review_scores_location              3779 non-null float64
review_scores_value                 3778 non-null float64
requires_license                    5835 non-null object
license                             1 non-null float64
jurisdiction_names                  0 non-null float64
instant_bookable                    5835 non-null object
cancellation_policy                 5835 non-null object
require_guest_profile_picture       5835 non-null object
require_guest_phone_verification    5835 non-null object
```

```
calculated_host_listings_count      5835 non-null int64
reviews_per_month                   3827 non-null float64
dtypes: float64(20), int64(14), object(58)
memory usage: 4.1+ MB
```

It turns out that one record displays a very odd location and, for the sake of the illustration, we will remove it:

```
odd = lst.loc[lst.longitude>-80, ['longitude', 'latitude']]
odd
```

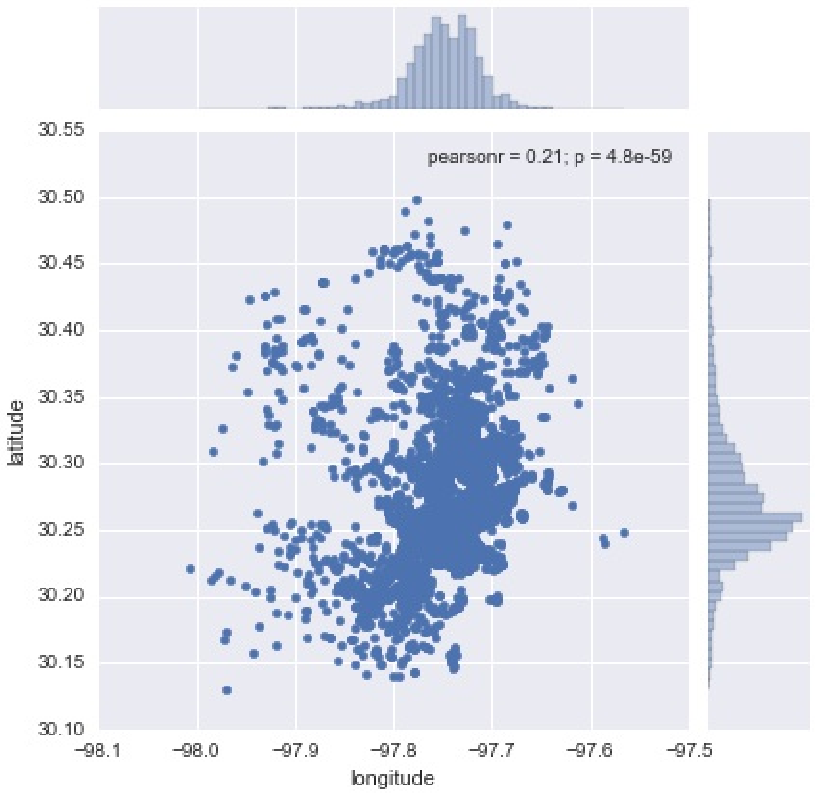
	longitude	latitude
5832	-5.093682	43.214991

```
lst = lst.drop(odd.index)
```

Point Visualization

The most straighforward way to get a first glimpse of the distribution of the data is to plot their latitude and longitude:

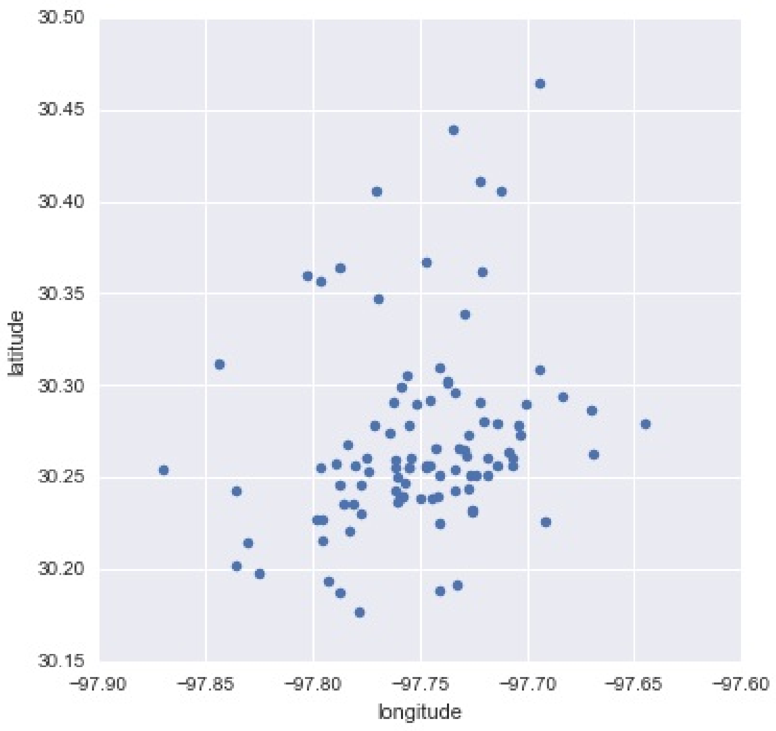
```
sns.jointplot(x="longitude", y="latitude", data=lst);
```



Now this does not neccesarily tell us much about the dataset or the distribution of locations within Austin. There are two main challenges in interpreting the plot: one, there is lack of context, which means the points are not identifiable over space (unless you are so familiar with lon/lat pairs that they have a clear meaning to you); and two, in the center of the plot, there are so many points that it is hard to tell any patter other than a big blurb of blue.

Let us first focus on the first problem, geographical context. The quickest and easiest way to provide context to this set of points is to overlay a general map. If we had an image with the map or a set of several data sources that we could aggregate to create a map, we could build it from scratch. But in the XXI Century, the easiest is to overlay our point dataset on top of a web map. In this case, we will use Leaflet, and we will convert our underlying `matplotlib` points with `mplleaflet`. The full dataset (+5k observations) is a bit too much for leaflet to plot it directly on screen, so we will obtain a random sample of 100 points:

```
# NOTE: `mpll.display` turned off to be able to compile the website,
#       comment out the last line of this cell for rendering Leaflet map.
rids = np.arange(lst.shape[0])
np.random.shuffle(rids)
f, ax = plt.subplots(1, figsize=(6, 6))
lst.iloc[rids[:100], :].plot(kind='scatter', x='longitude', y='latitude', \
                             s=30, linewidth=0, ax=ax);
#mpll.display(fig=f,)
```



This map allows us to get a much better sense of where the points are and what type of location they might be in. For example, now we can see that the bigh blue blurb has to do with the urbanized core of Austin.

`bokeh` alternative

Leaflet is not the only technology to display data on maps, although it is probably the default option in many cases. When the data is larger than "acceptable", we need to resort to more technically sophisticated alternatives. One option is provided by `bokeh` and its `datashaded` submodule (see here for a very nice introduction to the library, from where this example has been adapted).

Before we delve into `bokeh`, let us reproject our original data (lon/lat coordinates) into Web Mercator, as `bokeh` will expect them. To do that, we turn the coordinates into a `GeoSeries`:

```
from shapely.geometry import Point
xys_wb = gpd.GeoSeries(lst[['longitude', 'latitude']].apply(Point, axis=1), \
                      crs="+init=epsg:4326")
xys_wb = xys_wb.to_crs(epsg=3857)
x_wb = xys_wb.apply(lambda i: i.x)
y_wb = xys_wb.apply(lambda i: i.y)
```

Now we are ready to setup the plot in `bokeh` :

```
from bokeh.plotting import figure, output_notebook, show
from bokeh.tile_providers import STAMEN_TERRAIN
output_notebook()

minx, miny, maxx, maxy = xys_wb.total_bounds
y_range = miny, maxy
x_range = minx, maxx

def base_plot(tools='pan,wheel_zoom,reset',plot_width=600, plot_height=400, **plot_args):
    p = figure(tools=tools, plot_width=plot_width, plot_height=plot_height,
        x_range=x_range, y_range=y_range, outline_line_color=None,
        min_border=0, min_border_left=0, min_border_right=0,
        min_border_top=0, min_border_bottom=0, **plot_args)

    p.axis.visible = False
    p.xgrid.grid_line_color = None
    p.ygrid.grid_line_color = None
    return p

options = dict(line_color=None, fill_color='#800080', size=4)
```

```
<div class="bk-banner">
    <a href="http://bokeh.pydata.org" target="_blank" class="bk-logo bk-logo-small bk-logo-notebook"></a>
    <span id="5c0094a0-0ee5-4dba-8405-86719c3d1fa2">Loading BokehJS ...</span>
</div>
```

And good to go for mapping!

```
# NOTE: `show` turned off to be able to compile the website,
#       comment out the last line of this cell for rendering.
p = base_plot()
p.add_tile(STAMEN_TERRAIN)
p.circle(x=x_wb, y=y_wb, **options)
#show(p)
```

```
<bokeh.models.renderers.GlyphRenderer at 0x7fac4f7abc90>
```

As you can quickly see, `bokeh` is substantially faster at rendering larger amounts of data.

The second problem we have spotted with the first scatter is that, when the number of points grows, at some point it becomes impossible to discern anything other than a big blur of color. To some extent, interactivity gets at that problem by allowing the user to zoom in until every point is an entity on its own. However, there exist techniques that allow to summarize the data to be able to capture the overall pattern at once. Traditionally, kernel density estimation (KDE) has been one of the most common solutions by approximating a continuous surface of point intensity. In this context, however, we will explore a more recent alternative suggested by the `datashader` library (see the paper if interested in more details).

Arguably, our dataset is not large enough to justify the use of a reduction technique like datashader, but we will create the plot for the sake of the illustration. Keep in mind, the usefulness of this approach increases the more points you need to be plotting.

```
# NOTE: `show` turned off to be able to compile the website,
#       comment out the last line of this cell for rendering.

import datashader as ds
from datashader.callbacks import InteractiveImage
from datashader.colors import viridis
from datashader import transfer_functions as tf
from bokeh.tile_providers import STAMEN_TONER

p = base_plot()
p.add_tile(STAMEN_TONER)

pts = pd.DataFrame({'x': x_wb, 'y': y_wb})
pts['count'] = 1
def create_image90(x_range, y_range, w, h):
    cvs = ds.Canvas(plot_width=w, plot_height=h, x_range=x_range, y_range=y_range)
    agg = cvs.points(pts, 'x', 'y',  ds.count('count'))
    img = tf.interpolate(agg.where(agg > np.percentile(agg,90)), \
                         cmap=viridis, how='eq_hist')
    return tf.dynspread(img, threshold=0.1, max_px=4)

#InteractiveImage(p, create_image90)
```

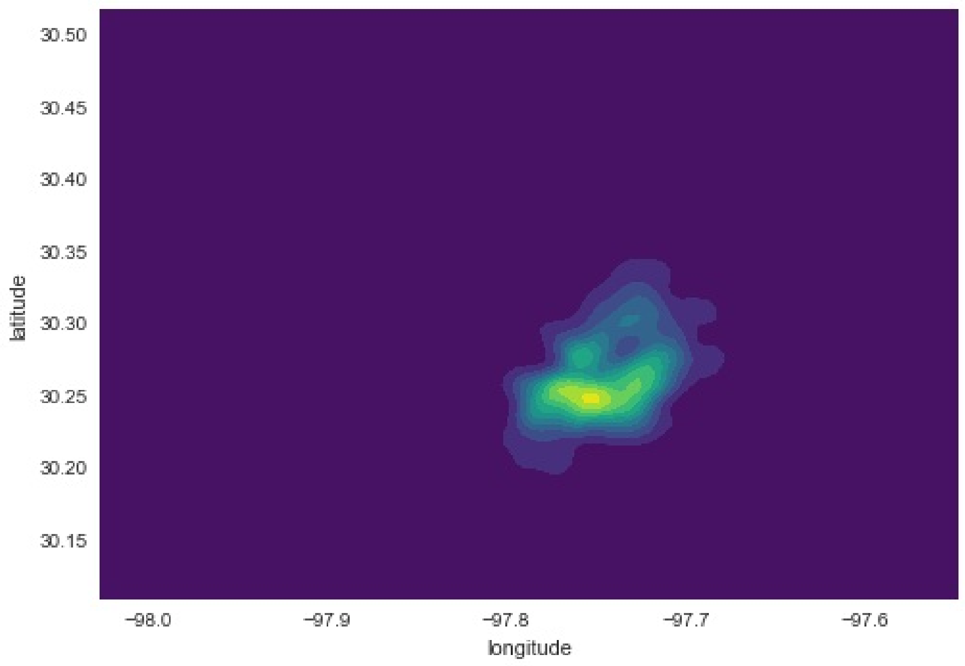
The key advandage of `datashader` is that is decouples the point processing from the plotting. That is the bit that allows it to be scalable to truly large datasets (e.g. millions of points). Essentially, the approach is based on generating a very fine grid, counting points within pixels, and encoding the count into a color scheme. In our map, this is not particularly effective because we do not have too many points (the previous plot is probably a more effective one) and esssentially there is a pixel per location of every point. However, hopefully this example shows how to create this kind of scalable maps.

Kernel Density Estimation

A common alternative when the number of points grows is to replace plotting every single point by estimating the continuous observed probability distribution. In this case, we will not be visualizing the points themselves, but an abstracted surface that models the probability of point density over space. The most commonly used method to do this is the so called kernel density estimate (KDE). The idea behind KDEs is to count the number of points in a continious way. Instead of using discrete counting, where you include a point in the count if it is inside a certain boundary and ignore it otherwise, KDEs use functions (kernels) that include points but give different weights to each one depending of how far of the location where we are counting the point is.

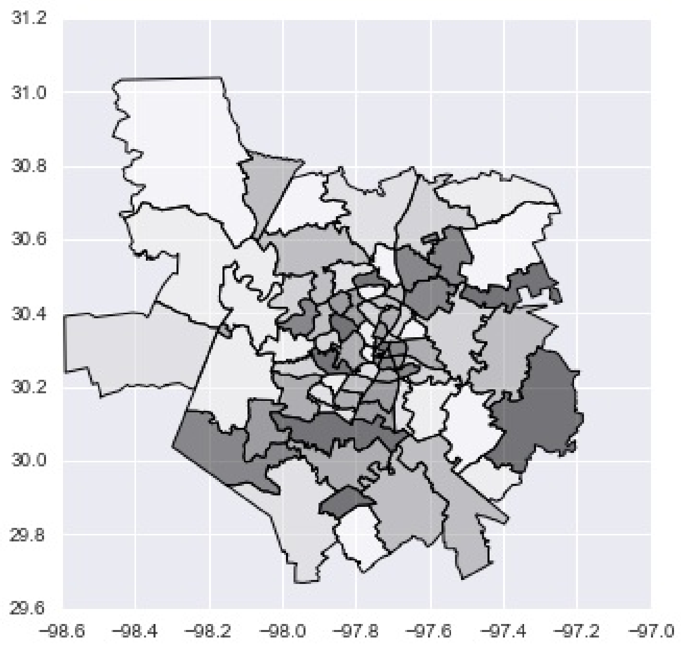
Creating a KDE is very straightfoward in Python. In its simplest form, we can run the following single line of code:

```
sns.kdeplot(lst['longitude'], lst['latitude'], shade=True, cmap='viridis');
```



Now, if we want to include additional layers of data to provide context, we can do so in the same way we would layer up different elements in `matplotlib`. Let us load first the Zip codes in Austin, for example:

```
zc = gpd.read_file('../data/Zipcodes.geojson')
zc.plot();
```



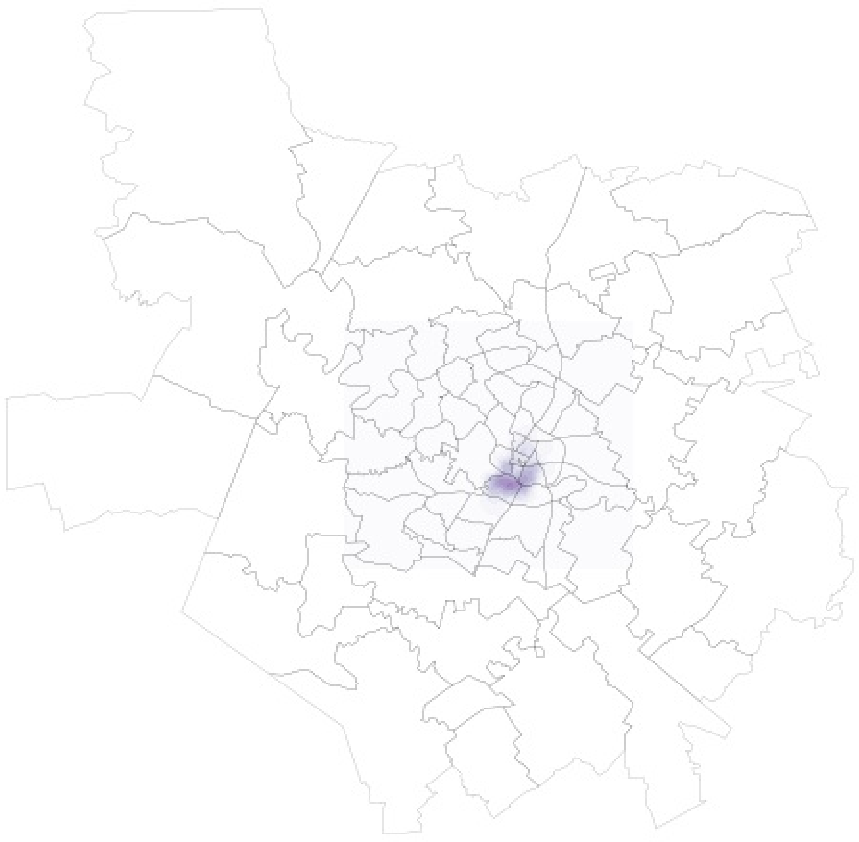
And, to overlay both layers:

```
f, ax = plt.subplots(1, figsize=(9, 9))

zc.plot(color='white', linewidth=0.1, ax=ax)

sns.kdeplot(lst['longitude'], lst['latitude'], \
            shade=True, cmap='Purples', \
            ax=ax);

ax.set_axis_off()
plt.axis('equal')
plt.show()
```



Exercise

> *Split the dataset by type of property and create a map for the five most common types.*

Consider the following sorting of property types:

```
lst.property_type.groupby(lst.property_type)\
                 .count()\
                 .sort_values(ascending=False)
```

```
property_type
House              3549
Apartment          1855
Condominium         106
Loft                 83
Townhouse            57
Other                47
Bed & Breakfast      37
Camper/RV            34
Bungalow             18
Cabin                17
Tent                 11
Villa                 7
Treehouse             7
Earth House           2
Chalet                1
Hut                   1
Boat                  1
Tipi                  1
Name: property_type, dtype: int64
```

Spatial Clustering

> IPYNB
>
> **NOTE**: much of this material has been ported and adapted from "Lab 8" in Arribas-Bel (2016).

This notebook covers a brief introduction to spatial regression. To demonstrate this, we will use a dataset of all the AirBnb listings in the city of Austin (check the Data section for more information about the dataset).

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 core 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 multi-dimensional 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.

The basic premise of the exercises we will be doing in this notebook is that, through the characteristics of the houses listed in AirBnb, we can learn about the geography of Austin. In particular, we will try to classify the city's zipcodes into a small number of groups that will allow us to extract some patterns about the main kinds of houses and areas in the city.

Data

Before anything, let us load up the libraries we will use:

```
%matplotlib inline

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

sns.set(style="whitegrid")
```

Let us also set the paths to all the files we will need throughout the tutorial:

```
# Adjust this to point to the right file in your computer
abb_link = '../data/listings.csv.gz'
zc_link = '../data/Zipcodes.geojson'
```

Before anything, let us load the main dataset:

```
lst = pd.read_csv(link)
```

Originally, this is provided at the individual level. Since we will be working in terms of neighborhoods and areas, we will need to aggregate them to that level. For this illustration, we will be using the following subset of variables:

```
varis = ['bedrooms', 'bathrooms', 'beds']
```

This will allow us to capture the main elements that describe the "look and feel" of a property and, by aggregation, of an area or neighborhood. All of the variables above are numerical values, so a sensible way to aggregate them is by obtaining the average (of bedrooms, etc.) per zipcode.

```
aves = lst.groupby('zipcode')[varis].mean()
aves.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Float64Index: 47 entries, 33558.0 to 78759.0
Data columns (total 3 columns):
bedrooms     47 non-null float64
bathrooms    47 non-null float64
beds         47 non-null float64
dtypes: float64(3)
memory usage: 1.5 KB
```

In addition to these variables, it would be good to include also a sense of what proportions of different types of houses each zipcode has. For example, one can imagine that neighborhoods with a higher proportion of condos than single-family homes will probably look and feel more urban. To do this, we need to do some data munging:

```
types = pd.get_dummies(lst['property_type'])
prop_types = types.join(lst['zipcode'])\
                  .groupby('zipcode')\
                  .sum()
prop_types_pct = (prop_types * 100.).div(prop_types.sum(axis=1), axis=0)
prop_types_pct.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Float64Index: 47 entries, 33558.0 to 78759.0
Data columns (total 18 columns):
Apartment          47 non-null float64
Bed & Breakfast    47 non-null float64
Boat               47 non-null float64
Bungalow           47 non-null float64
Cabin              47 non-null float64
Camper/RV          47 non-null float64
Chalet             47 non-null float64
Condominium        47 non-null float64
Earth House        47 non-null float64
House              47 non-null float64
Hut                47 non-null float64
Loft               47 non-null float64
Other              47 non-null float64
Tent               47 non-null float64
Tipi               47 non-null float64
Townhouse          47 non-null float64
Treehouse          47 non-null float64
Villa              47 non-null float64
dtypes: float64(18)
memory usage: 7.0 KB
```

Now we bring both sets of variables together:

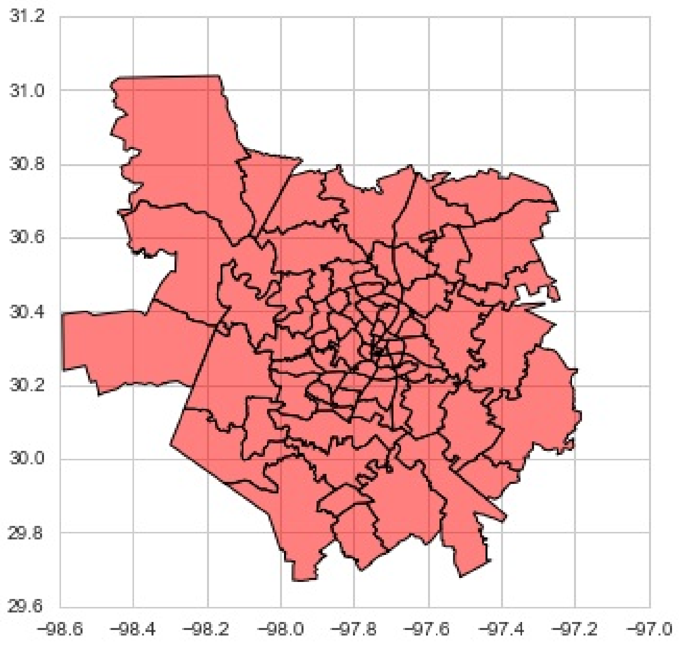
```
aves_props = aves.join(prop_types_pct)
```

And since we will be feeding this into the clustering algorithm, we will first standardize the columns:

```
db = pd.DataFrame(\
                 scale(aves_props), \
                 index=aves_props.index, \
                 columns=aves_props.columns)\
       .rename(lambda x: str(int(x)))
```

Now let us bring geography in:

```
zc = gpd.read_file(zc_link)
zc.plot(color='red');
```



And combine the two:

```
zdb = zc[['geometry', 'zipcode', 'name']].join(db, on='zipcode')\
                                          .dropna()
```

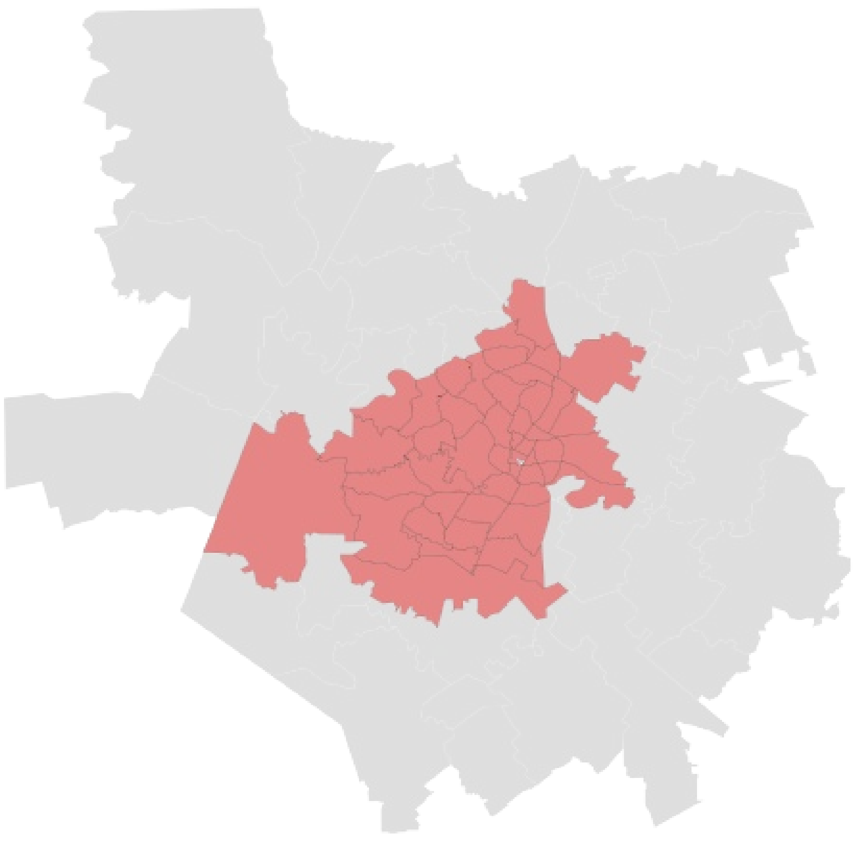
To get a sense of which areas we have lost:

```
f, ax = plt.subplots(1, figsize=(9, 9))

zc.plot(color='grey', linewidth=0, ax=ax)
zdb.plot(color='red', linewidth=0.1, ax=ax)

ax.set_axis_off()

plt.show()
```



Geodemographic analysis

The main intuition behind geodemographic analysis is to group disparate areas of a city or region into a small set of classes that capture several characteristics shared by those in the same group. By doing this, we can get a new perspective not only on the types of areas in a city, but on how they are distributed over space. In the context of our AirBnb data analysis, the idea is that we can group different zipcodes of Austin based on the type of houses listed on the website. This will give us a hint into the geography of AirBnb in the Texan tech capital.

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.

Technically speaking, we describe the method and the parameters on the following line of code, where we specifically ask for five groups:

```
km5 = cluster.KMeans(n_clusters=5)
```

Following the `sklearn` pipeline approach, all the heavy-lifting of the clustering happens when we `fit` the model to the data:

```
km5cls = km5.fit(zdb.drop(['geometry', 'name'], axis=1).values)
```

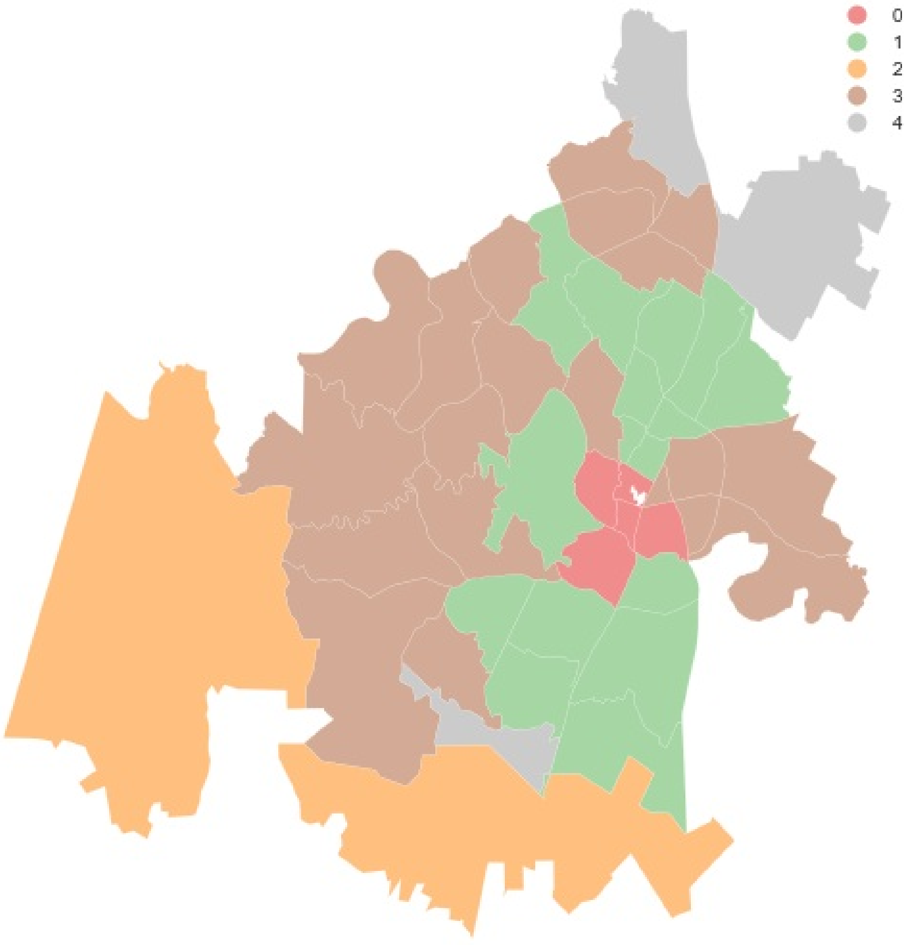
Now we can extract the classes and put them on a map:

```
f, ax = plt.subplots(1, figsize=(9, 9))

zdb.assign(cl=km5cls.labels_)\
   .plot(column='cl', categorical=True, legend=True, \
         linewidth=0.1, edgecolor='white', ax=ax)

ax.set_axis_off()

plt.show()
```

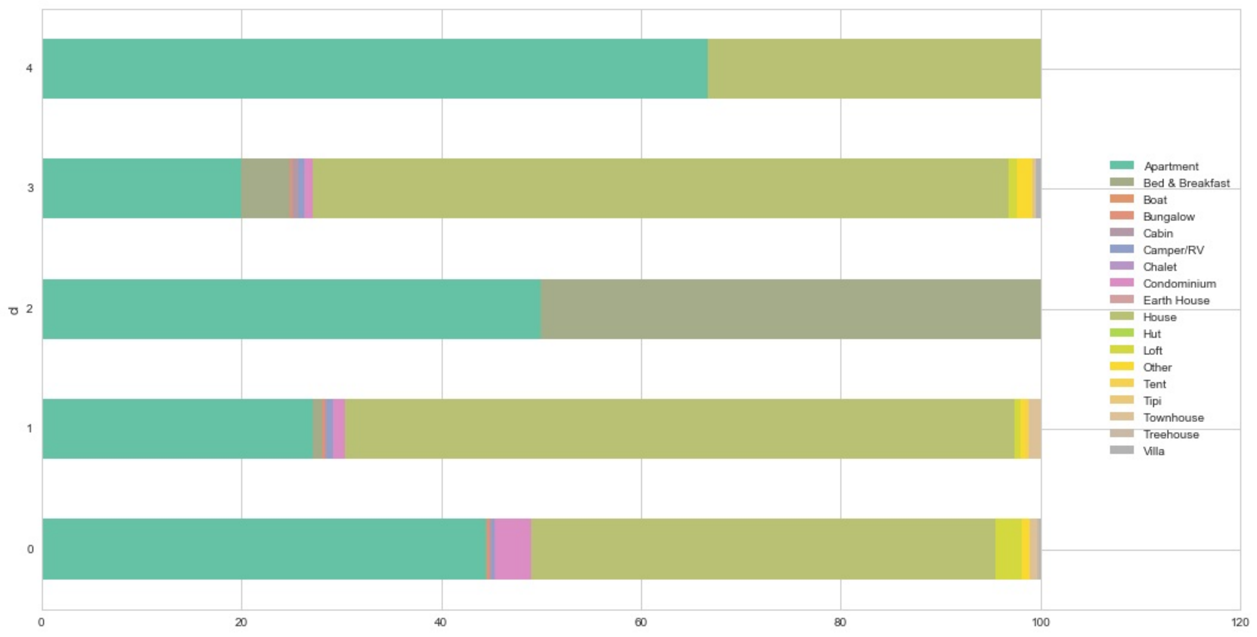


The map above shows a clear pattern: there is a class at the core of the city (number 0, in red), then two other ones in a sort of "urban ring" (number 1 and 3, in green and brown, respectively), and two peripheral sets of areas (number 2 and 4, yellow and green).

This gives us a good insight into the geographical structure, but does not tell us much about what are the defining elements of these groups. To do that, we can have a peak into the characteristics of the classes. For example, let us look at how the proportion of different types of properties are distributed across clusters:

```
cl_pcts = prop_types_pct.rename(lambda x: str(int(x)))\
                          .reindex(zdb['zipcode'])\
                          .assign(cl=km5cls.labels_)\
                          .groupby('cl')\
                          .mean()
```

```
f, ax = plt.subplots(1, figsize=(18, 9))
cl_pcts.plot(kind='barh', stacked=True, ax=ax, \
             cmap='Set2', linewidth=0)
ax.legend(ncol=1, loc="right");
```



A few interesting, albeit maybe not completely unsurprising, characteristics stand out. First, most of the locations we have in the dataset are either apartments or houses. However, how they are distributed is interesting. The urban core -cluster 0- distinctively has the highest proportion of condos and lofts. The suburban ring -clusters 1 and 3- is very consistent, with a large share of houses and less apartments, particularly so in the case of cluster 3. Class 4 has only two types of properties, houses and apartments, suggesting there are not that many places listed at AirBnb. Finally, class 3 arises as a more rural and leisure one: beyond apartmentes, it has a large share of bed & breakfasts.

Mini Exercise

> *What are the average number of beds, bedrooms and bathrooms for every class?*

Regionalization analysis: building (meaningful) regions

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 class or cluster -hence regionalization, the construction of regions from smaller areas.

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.

In this example, we are going to create aggregations of zipcodes into groups that have areas where the AirBnb listed location have similar ratings. In other words, we will create delineations for the "quality" or "satisfaction" of AirBnb users. In other words, we will explore what are the boundaries that separate areas where AirBnb users tend to be satisfied about their experience versus those where the ratings are not as high. To do this, we will focus on the `review_scores_X` set of variables in the original dataset:

```
ratings = [i for i in lst if 'review_scores_' in i]
ratings
```

```
['review_scores_rating',
 'review_scores_accuracy',
 'review_scores_cleanliness',
 'review_scores_checkin',
 'review_scores_communication',
 'review_scores_location',
 'review_scores_value']
```

Similarly to the case above, we now bring this at the zipcode level. Note that, since they are all scores that range from 0 to 100, we can use averages and we do not need to standardize.

```
rt_av = lst.groupby('zipcode')[ratings]\
           .mean()\
           .rename(lambda x: str(int(x)))
```

And we link these to the geometries of zipcodes:

```
zrt = zc[['geometry', 'zipcode']].join(rt_av, on='zipcode')\
                                 .dropna()
zrt.info()
```

```
<class 'geopandas.geodataframe.GeoDataFrame'>
Int64Index: 43 entries, 0 to 78
Data columns (total 9 columns):
geometry                       43 non-null object
zipcode                        43 non-null object
review_scores_rating           43 non-null float64
review_scores_accuracy         43 non-null float64
review_scores_cleanliness      43 non-null float64
review_scores_checkin          43 non-null float64
review_scores_communication    43 non-null float64
review_scores_location         43 non-null float64
review_scores_value            43 non-null float64
dtypes: float64(7), object(2)
memory usage: 3.4+ KB
```

In contrast to the standard clustering techniques, regionalization requires a formal representation of topology. This is so the algorithm can impose spatial constraints during the process of clustering the observations. We will use exactly the same approach as in the previous sections of this tutorial for this and build spatial weights objects `W` with `PySAL`. For the sake of this illustration, we will consider queen contiguity, but any other rule should work fine as long as there is a rational behind it. Weights constructors currently only work from shapefiles on disk, so we will write our `GeoDataFrame` first, then create the `W` object, and remove the files.

```
zrt.to_file('tmp')
w = ps.queen_from_shapefile('tmp/tmp.shp', idVariable='zipcode')
# NOTE: this might not work on Windows
! rm -r tmp
w
```

```
<pysal.weights.weights.W at 0x7f0cc9b433d0>
```

Now we are ready to run the regionalization algorithm. In this case we will use the `max-p` (Duque, Anselin & Rey, 2012), which does not require a predefined number of output regions but instead it takes a target variable that you want to make sure a minimum threshold is met. In our case, since it is based on ratings, we will impose that every resulting region has at least 10% of the total number of reviews. Let us work through what that would mean:

```
n_rev = lst.groupby('zipcode')\
           .sum()\
           ['number_of_reviews']\
           .rename(lambda x: str(int(x)))\
           .reindex(zrt['zipcode'])
thr = np.round(0.1 * n_rev.sum())
thr
```

```
6271.0
```

This means we want every resulting region to be based on at least 6,271 reviews. Now we have all the pieces, let us glue them together through the algorithm:

```
# Set the seed for reproducibility
np.random.seed(1234)

z = zrt.drop(['geometry', 'zipcode'], axis=1).values
maxp = ps.region.Maxp(w, z, thr, n_rev.values[:, None], initial=1000)
```

We can check whether the solution is better (lower within sum of squares) than we would have gotten from a purely random regionalization process using the `cinference` method:

```
%%time
np.random.seed(1234)
maxp.cinference(nperm=999)
```

```
CPU times: user 27.4 s, sys: 0 ns, total: 27.4 s
Wall time: 27.4 s
```

Which allows us to obtain an empirical p-value:

```
maxp.cpvalue
```

```
0.001
```

Which gives us reasonably good confidence that the solution we obtain is more meaningful than pure chance.

With that out of the way, let us see what the result looks like on a map! First we extract the labels:

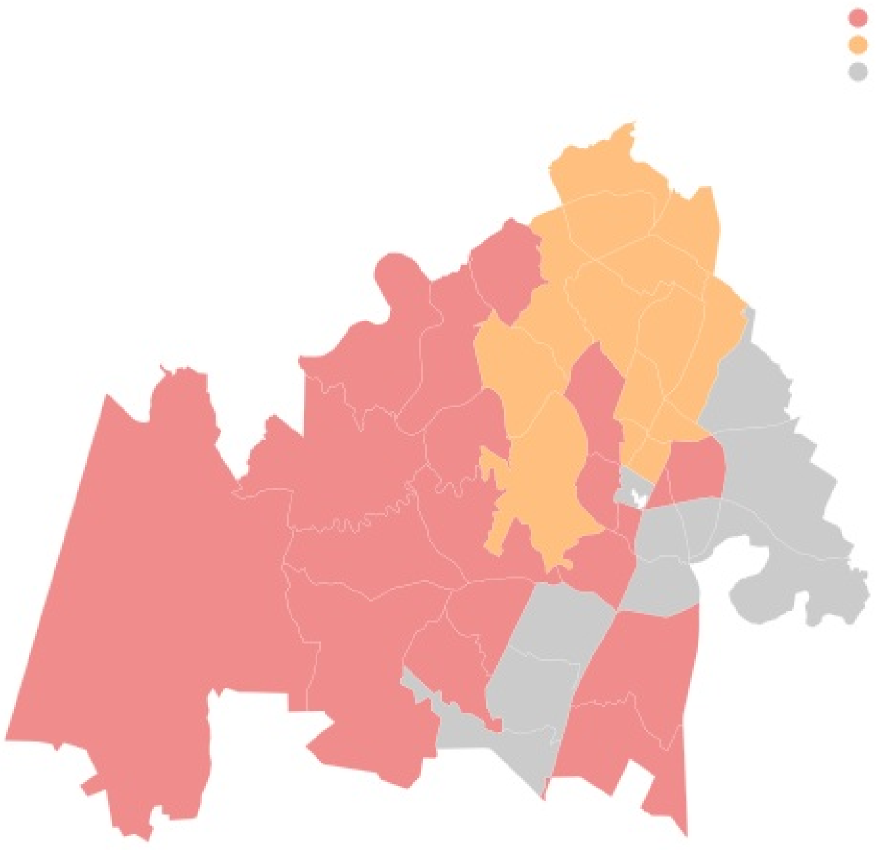
```
lbls = pd.Series(maxp.area2region).reindex(zrt['zipcode'])
```

```
f, ax = plt.subplots(1, figsize=(9, 9))

zrt.assign(cl=lbls.values)\
   .plot(column='cl', categorical=True, legend=True, \
         linewidth=0.1, edgecolor='white', ax=ax)

ax.set_axis_off()

plt.show()
```



The map shows a clear geographical pattern with a western area, another in the North and a smaller one in the East. Let us unpack what each of them is made of:

```
zrt[ratings].groupby(lbls.values).mean().T
```

	0	1	2
review_scores_rating	96.425334	95.264603	92.111148
review_scores_accuracy	9.703800	9.561277	9.548701
review_scores_cleanliness	9.597942	9.610098	8.965437
review_scores_checkin	9.876563	9.821887	9.764887
review_scores_communication	9.909209	9.821964	9.775652
review_scores_location	9.609283	9.483582	8.893100
review_scores_value	9.618152	9.543733	9.441086

Although very similar, there are some patterns to be extracted. For example, the East area seems to have lower overall scores.

Exercise

> *Obtain a geodemographic classification with eight classes instead of five and replicate the analysis above*
>
> *Re-run the regionalization exercise imposing a minimum of 5% reviews per area*

Spatial Regression

> IPYNB
>
> **NOTE**: some of this material has been ported and adapted from the Spatial Econometrics note in Arribas-Bel (2016b).

This notebook covers a brief and gentle introduction to spatial econometrics in Python. To do that, we will use a set of Austin properties listed in AirBnb.

The core idea of spatial econometrics is to introduce a formal representation of space into the statistical framework for regression. This can be done in many ways: by including predictors based on space (e.g. distance to relevant features), by splitting the datasets into subsets that map into different geographical regions (e.g. spatial regimes), by exploiting close distance to other observations to borrow information in the estimation (e.g. kriging), or by introducing variables that put in relation their value at a given location with those in nearby locations, to give a few examples. Some of these approaches can be implemented with standard non-spatial techniques, while others require bespoke models that can deal with the issues introduced. In this short tutorial, we will focus on the latter group. In particular, we will introduce some of the most commonly used methods in the field of spatial econometrics.

The example we will use to demonstrate this draws on hedonic house price modelling. This a well-established methodology that was developed by Rosen (1974) that is capable of recovering the marginal willingness to pay for goods or services that are not traded in the market. In other words, this allows us to put an implicit price on things such as living close to a park or in a neighborhood with good quality of air. In addition, since hedonic models are based on linear regression, the technique can also be used to obtain predictions of house prices.

Data

Before anything, let us load up the libraries we will use:

```
%matplotlib inline

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

sns.set(style="whitegrid")
```

Let us also set the paths to all the files we will need throughout the tutorial, which is only the original table of listings:

```
# Adjust this to point to the right file in your computer
abb_link = '../data/listings.csv.gz'
```

And go ahead and load it up too:

```
lst = pd.read_csv(abb_link)
```

Baseline (nonspatial) regression

Before introducing explicitly spatial methods, we will run a simple linear regression model. This will allow us, on the one hand, set the main principles of hedonic modeling and how to interpret the coefficients, which is good because the spatial models will build on this; and, on the other hand, it will provide a baseline model that we can use to evaluate how meaningful the spatial extensions are.

Essentially, the core of a linear regression is to explain a given variable -the price of a listing i on AirBnb (P_i)- as a linear function of a set of other characteristics we will collectively call X_i:

$$\ln(P_i) = \alpha + \beta X_i + \epsilon_i$$

For several reasons, it is common practice to introduce the price in logarithms, so we will do so here. Additionally, since this is a probabilistic model, we add an error term ϵ_i that is assumed to be well-behaved (i.i.d. as a normal).

For our example, we will consider the following set of explanatory features of each listed property:

```
x = ['host_listings_count', 'bathrooms', 'bedrooms', 'beds', 'guests_included']
```

Additionally, we are going to derive a new feature of a listing from the `amenities` variable. Let us construct a variable that takes 1 if the listed property has a pool and 0 otherwise:

```
def has_pool(a):
    if 'Pool' in a:
        return 1
    else:
        return 0

lst['pool'] = lst['amenities'].apply(has_pool)
```

For convenience, we will re-package the variables:

```
yxs = lst.loc[:, x + ['pool', 'price']].dropna()
y = np.log(\
           yxs['price'].apply(lambda x: float(x.strip('$').replace(',', '')))\
           + 0.000001
          )
```

To run the model, we can use the `spreg` module in `PySAL`, which implements a standard OLS routine, but is particularly well suited for regressions on spatial data. Also, although for the initial model we do not need it, let us build a spatial weights matrix that connects every observation to its 8 nearest neighbors. This will allow us to get extra diagnostics from the baseline model.

```
w = ps.knnW_from_array(lst.loc[\
                               yxs.index, \
                              ['longitude', 'latitude']\
                              ].values)
w.transform = 'R'
w
```

```
<pysal.weights.weights.W at 0x7f218e902890>
```

At this point, we are ready to fit the regression:

```
m1 = ps.spreg.OLS(y.values[:, None], yxs.drop('price', axis=1).values, \
                  w=w, spat_diag=True, \
                  name_x=yxs.drop('price', axis=1).columns.tolist(), name_y='ln(price)')
```

To get a quick glimpse of the results, we can print its summary:

```
print(m1.summary)
```

```
REGRESSION
----------
SUMMARY OF OUTPUT: ORDINARY LEAST SQUARES
-----------------------------------------
Data set            :     unknown
Weights matrix      :     unknown
Dependent Variable  :   ln(price)                Number of Observations:        5767
Mean dependent var  :      5.1952                Number of Variables   :           7
S.D. dependent var  :      0.9455                Degrees of Freedom    :        5760
R-squared           :      0.4042
Adjusted R-squared  :      0.4036
Sum squared residual:    3071.189                F-statistic           :    651.3958
Sigma-square        :       0.533                Prob(F-statistic)     :           0
S.E. of regression  :       0.730                Log likelihood        :   -6366.162
Sigma-square ML     :       0.533                Akaike info criterion :   12746.325
S.E of regression ML:      0.7298                Schwarz criterion     :   12792.944

------------------------------------------------------------------------------------
            Variable     Coefficient       Std.Error     t-Statistic     Probability
------------------------------------------------------------------------------------
            CONSTANT       4.0976886       0.0223530     183.3171506       0.0000000
 host_listings_count      -0.0000130       0.0001790      -0.0726772       0.9420655
           bathrooms       0.2947079       0.0194817      15.1273879       0.0000000
            bedrooms       0.3274226       0.0159666      20.5067654       0.0000000
                beds       0.0245741       0.0097379       2.5235601       0.0116440
     guests_included       0.0075119       0.0060551       1.2406028       0.2148030
                pool       0.0888039       0.0221903       4.0019209       0.0000636
------------------------------------------------------------------------------------

REGRESSION DIAGNOSTICS
MULTICOLLINEARITY CONDITION NUMBER            9.260

TEST ON NORMALITY OF ERRORS
TEST                             DF        VALUE           PROB
Jarque-Bera                       2     1358479.047           0.0000

DIAGNOSTICS FOR HETEROSKEDASTICITY
RANDOM COEFFICIENTS
TEST                             DF        VALUE           PROB
Breusch-Pagan test                6        1414.297           0.0000
Koenker-Bassett test              6          36.756           0.0000

DIAGNOSTICS FOR SPATIAL DEPENDENCE
TEST                           MI/DF       VALUE           PROB
Lagrange Multiplier (lag)         1         255.796           0.0000
Robust LM (lag)                   1          13.039           0.0003
Lagrange Multiplier (error)       1         278.752           0.0000
Robust LM (error)                 1          35.995           0.0000
Lagrange Multiplier (SARMA)       2         291.791           0.0000

================================ END OF REPORT =====================================
```

Results are largely unsurprising, but nonetheless reassuring. Both an extra bedroom and an extra bathroom increase the final price around 30%. Accounting for those, an extra bed pushes the price about 2%. Neither the number of guests included nor the number of listings the host has in total have a significant effect on the final price.

Including a spatial weights object in the regression buys you an extra bit: the summary provides results on the diagnostics for spatial dependence. These are a series of statistics that test whether the residuals of the regression are spatially correlated, against the null of a random distribution over space. If the latter is rejected a key assumption of OLS, independently distributed error terms, is violated. Depending on the structure of the spatial pattern, different strategies have been defined within the spatial econometrics literature to deal with them. If you are interested in this, a very recent and good resource to check out is

Anselin & Rey (2015). The main summary from the diagnostics for spatial dependence is that there is clear evidence to reject the null of spatial randomness in the residuals, hence an explicitly spatial approach is warranted.

Spatially lagged exogenous regressors (`WX`)

The first and most straightforward way to introduce space is by "spatially lagging" one of the explanatory variables. Mathematically, this can be expressed as follows:

$$\ln(P_i) = \alpha + \beta X_i + \delta \sum_j w_{ij} X_i' + \epsilon_i$$

where $X'*i$ is a subset of X_i, although it could encompass all of the explanatory variables, and $w*{ij}$ is the ij-th cell of a spatial weights matrix W. Because W assigns non-zero values only to spatial neighbors, if W is row-standardized (customary in this context), then $\sum*j w*{ij} X'_i$ captures the average value of X'_i in the surroundings of location i. This is what we call the *spatial lag* of X_i. Also, since it is a spatial transformation of an explanatory variable, the standard estimation approach -OLS- is sufficient: spatially lagging the variables does not violate any of the assumptions on which OLS relies.

Usually, we will want to spatially lag variables that we think may affect the price of a house in a given location. For example, one could think that pools represent a visual amenity. If that is the case, then listed properties surrounded by other properties with pools might, everything else equal, be more expensive. To calculate the number of pools surrounding each property, we can build an alternative weights matrix that we do not row-standardize:

```
w_pool = ps.knnW_from_array(lst.loc[\
                               yxs.index, \
                              ['longitude', 'latitude']\
                              ].values)
yxs_w = yxs.assign(w_pool=ps.lag_spatial(w_pool, yxs['pool'].values))
```

And now we can run the model, which has the same setup as `m1`, with the exception that it includes the number of AirBnb properties with pools surrounding each house:

```
m2 = ps.spreg.OLS(y.values[:, None], \
                  yxs_w.drop('price', axis=1).values, \
                  w=w, spat_diag=True, \
                  name_x=yxs_w.drop('price', axis=1).columns.tolist(), name_y='ln(price)')
```

```
print(m2.summary)
```

```
REGRESSION
----------
SUMMARY OF OUTPUT: ORDINARY LEAST SQUARES
-----------------------------------------
Data set            :     unknown
Weights matrix      :     unknown
Dependent Variable  :   ln(price)                Number of Observations:        5767
Mean dependent var  :      5.1952                Number of Variables   :           8
S.D. dependent var  :      0.9455                Degrees of Freedom    :        5759
R-squared           :      0.4044
Adjusted R-squared  :      0.4037
Sum squared residual:    3070.363                F-statistic           :    558.6139
Sigma-square        :       0.533                Prob(F-statistic)     :           0
S.E. of regression  :       0.730                Log likelihood        :   -6365.387
Sigma-square ML     :       0.532                Akaike info criterion :   12746.773
S.E of regression ML:      0.7297                Schwarz criterion     :   12800.053

------------------------------------------------------------------------------------
            Variable     Coefficient       Std.Error     t-Statistic     Probability
------------------------------------------------------------------------------------
            CONSTANT       4.0906444       0.0230571     177.4134022       0.0000000
 host_listings_count      -0.0000108       0.0001790      -0.0603617       0.9518697
           bathrooms       0.2948787       0.0194813      15.1365024       0.0000000
            bedrooms       0.3277450       0.0159679      20.5252404       0.0000000
                beds       0.0246650       0.0097377       2.5329419       0.0113373
     guests_included       0.0076894       0.0060564       1.2696250       0.2042695
                pool       0.0725756       0.0257356       2.8200486       0.0048181
              w_pool       0.0188875       0.0151729       1.2448141       0.2132508
------------------------------------------------------------------------------------

REGRESSION DIAGNOSTICS
MULTICOLLINEARITY CONDITION NUMBER            9.605

TEST ON NORMALITY OF ERRORS
TEST                             DF        VALUE           PROB
Jarque-Bera                       2     1368880.320           0.0000

DIAGNOSTICS FOR HETEROSKEDASTICITY
RANDOM COEFFICIENTS
TEST                             DF        VALUE           PROB
Breusch-Pagan test                7        1565.566           0.0000
Koenker-Bassett test              7          40.537           0.0000

DIAGNOSTICS FOR SPATIAL DEPENDENCE
TEST                           MI/DF       VALUE           PROB
Lagrange Multiplier (lag)         1         255.124           0.0000
Robust LM (lag)                   1          13.448           0.0002
Lagrange Multiplier (error)       1         276.862           0.0000
Robust LM (error)                 1          35.187           0.0000
Lagrange Multiplier (SARMA)       2         290.310           0.0000

================================ END OF REPORT =====================================
```

Results are largely consistent with the original model. Also, incidentally, the number of pools surrounding a property does not appear to have any significant effect on the price of a given property. This could be for a host of reasons: maybe AirBnb customers do not value the number of pools surrounding a property where they are looking to stay; but maybe they do but our dataset only allows us to capture the number of pools in *other* AirBnb properties, which is not necessarily a good proxy of the number of pools in the immediate surroundings of a given property.

Spatially lagged endogenous regressors (`WY`)

In a similar way to how we have included the spatial lag, one could think the prices of houses surrounding a given property also enter its own price function. In math terms, this implies the following:

$$\ln(P_i) = \alpha + \lambda \sum_j w_{ij} \ln(P_i) + \beta X_i + \epsilon_i$$

This is essentially what we call a *spatial lag* model in spatial econometrics. Two calls for caution:

1. Unlike before, this specification *does* violate some of the assumptions on which OLS relies. In particular, it is including an endogenous variable on the right-hand side. This means we need a new estimation method to obtain reliable coefficients. The technical details of this go well beyond the scope of this workshop (although, if you are interested, go check Anselin & Rey, 2015). But we can offload those to `PySAL` and use the `GM_Lag` class, which implements the state-of-the-art approach to estimate this model.
2. A more conceptual *gotcha*: you might be tempted to read the equation above as the effect of the price in neighboring locations j on that of location i. This is not exactly the exact interpretation. Instead, we need to realize this is all assumed to be a "joint decission": rather than some houses setting their price first and that having a subsequent effect on others, what the equation models is an interdependent process by which each owner sets her own price *taking into account* the price that will be set in neighboring locations. This might read a bit like a technical subtlety and, to some extent, it is; but it is important to keep it in mind when you are interpreting the results.

Let us see how you would run this using `PySAL`:

```
m3 = ps.spreg.GM_Lag(y.values[:, None], yxs.drop('price', axis=1).values, \
                  w=w, spat_diag=True, \
                  name_x=yxs.drop('price', axis=1).columns.tolist(), name_y='ln(price)')
```

```
print(m3.summary)
```

```
REGRESSION
----------
SUMMARY OF OUTPUT: SPATIAL TWO STAGE LEAST SQUARES
--------------------------------------------------
Data set            :     unknown
Weights matrix      :     unknown
Dependent Variable  :   ln(price)                Number of Observations:        5767
Mean dependent var  :      5.1952                Number of Variables   :           8
S.D. dependent var  :      0.9455                Degrees of Freedom    :        5759
Pseudo R-squared    :      0.4224
Spatial Pseudo R-squared:  0.4056

------------------------------------------------------------------------------------
            Variable     Coefficient       Std.Error     z-Statistic     Probability
------------------------------------------------------------------------------------
            CONSTANT       3.7085715       0.1075621      34.4784213       0.0000000
 host_listings_count      -0.0000587       0.0001765      -0.3324585       0.7395430
           bathrooms       0.2857932       0.0193237      14.7897969       0.0000000
            bedrooms       0.3272598       0.0157132      20.8270544       0.0000000
                beds       0.0239548       0.0095848       2.4992528       0.0124455
     guests_included       0.0065147       0.0059651       1.0921407       0.2747713
                pool       0.0891100       0.0218383       4.0804521       0.0000449
         W_ln(price)       0.0785059       0.0212424       3.6957202       0.0002193
------------------------------------------------------------------------------------
Instrumented: W_ln(price)
Instruments: W_bathrooms, W_bedrooms, W_beds, W_guests_included,
             W_host_listings_count, W_pool

DIAGNOSTICS FOR SPATIAL DEPENDENCE
TEST                           MI/DF       VALUE           PROB
Anselin-Kelejian Test             1          31.545          0.0000
================================ END OF REPORT =====================================
```

As we can see, results are again very similar in all the other variable. It is also very clear that the estimate of the spatial lag of price is statistically significant. This points to evidence that there are processes of spatial interaction between property owners when they set their price.

Prediction performance of spatial models

Even if we are not interested in the interpretation of the model to learn more about how alternative factors determine the price of an AirBnb property, spatial econometrics can be useful. In a purely predictive setting, the use of explicitly spatial models is likely to improve accuracy in cases where space plays a key role in the data generating process. To have a quick look at this issue, we can use the mean squared error (MSE), a standard metric of accuracy in the machine learning literature, to evaluate whether explicitly spatial models are better than traditional, non-spatial ones:

```
from sklearn.metrics import mean_squared_error as mse

mses = pd.Series({'OLS': mse(y, m1.predy.flatten()), \
                  'OLS+W': mse(y, m2.predy.flatten()), \
                  'Lag': mse(y, m3.predy_e)
                 })
mses.sort_values()
```

```
Lag      0.531327
OLS+W    0.532402
OLS      0.532545
dtype: float64
```

We can see that the inclusion of the number of surrounding pools (which was insignificant) only marginally reduces the MSE. The inclusion of the spatial lag of price, however, does a better job at improving the accuracy of the model.

Exercise

> *Run a regression including both the spatial lag of pools and of the price. How does its predictive performance compare?*

Development workflow

Dependencies

In addition to the packages required to run the tutorial (see the install guide for more detail), you will need the following libraries:

- `npm` and `node.js`
- `gitbook`
- `make`
- `cp`, `rm`, and `zip` Unix utilities.

Workflow

The overall structure of the workflow is as follows:

1. Develop material on Jupyter notebooks and place them under the `content/` folder.
2. When you want to build the website with the new content run on, the root folder:

   `> make notebooks`

3. When you want to obtain a new version of the pdf or ebook formats, run on the root folder:

   `> make book`

4. When you want to push a new version to the website to Github Pages, make sure to commit all your changes first on the `master` branch (assuming your remote is named as `origin`):

   ```
   > git add .
   > git commit -m "commit message"
   > git push origin master
   ```

   Then you can run:

   `> make website`

   This will compile a new version of the website, pdf, eupb and mobi files, check them in, switch to the `gh-pages` branch, check the new version of the website and push it to Github.