Contents

ENVS363/563

Geographic Data Science

Welcome to Geographic Data Science, a course taught by Dr. Dani Arribas-Bel in the Autumn of 2020 at the University of Liverpool.

> **Note**
>
> A PDF version of this course is available for download here

Contact

> Dani Arribas-Bel - `D.Arribas-Bel [at] liverpool.ac.uk`
> Senior Lecturer in Geographic Data Science
> Office 508, Roxby Building,
> University of Liverpool - 74 Bedford St S,
> Liverpool, L69 7ZT,
> United Kingdom.

Citation

JOSE 10.21105/jose.00042


```
@article{darribas_gds_course,
  author = {Dani Arribas-Bel},
  title = {A course on Geographic Data Science},
  year = 2019,
  journal = {The Journal of Open Source Education},
  volume = 2,
  number = 14,
  doi = {https://doi.org/10.21105/jose.00042}
}
```

Overview

Aims

The module provides students with little or no prior knowledge core competences in Geographic Data Science (GDS). This includes the following:

- Advancing their statistical and numerical literacy.
- Introducing basic principles of programming and state-of-the-art computational tools for GDS.
- Presenting a comprehensive overview of the main methodologies available to the Geographic Data Scientist, as well as their intuition as to how and when they can be applied.
- Focusing on real world applications of these techniques in a geographical and applied context.

Learning outcomes

By the end of the course, students will be able to:

- Demonstrate advanced GIS/GDS concepts and be able to use the tools programmatically to import, manipulate and analyse spatial data in different formats.
- Understand the motivation and inner workings of the main methodological approcahes of GDS, both analytical and visual.
- Critically evaluate the suitability of a specific technique, what it can offer and how it can help answer questions of interest.
- Apply a number of spatial analysis techniques and explain how to interpret the results, in a process of turning data into information.
- When faced with a new data-set, work independently using GIS/GDS tools programmatically to extract valuable insight.

Feedback strategy

The student will receive feedback through the following channels:

- Formal assessment of three summative assignments: two tests and a computational essay. This will be on the form of reasoning of the mark assigned as well as comments specifying how the mark could be improved. This will be provided no later than three working weeks after the deadline of the assignment submission.
- Direct interaction with Module Leader and demonstrators in the computer labs. This will take place in each of the scheduled lab sessions of the course.
- Online forum maintained by the Module Leader where students can contribute by asking and answering questions related to the module.

Key texts and learning resources

Access to materials, including lecture slides and lab notebooks, is centralized through the use of a course website available in the following url:

```
https://darribas.org/gds_course
```

Specific videos, (computational) notebooks, and other resources, as well as academic references are provided for each learning block.

In addition, the currently-in-progress book *"Geographic Data Science with PySAL and the PyData stack"* provides and additional resource for more in-depth coverage of similar content.

Syllabus

Week 1: Introduction

- Lecture: Geographic Data Science.
- Tutorial: Tools + Manipulating data in Python - Tidy Data.

Week 2: Modern Computational Environments

- Lecture: Modern Computational Environments.
- Tutorial: Manipulating data in Python - Advanced Tricks.

Week 3: Spatial Data

- Lecture: Spatial Data.
- Tutorial: Manipulating geospatial data in Python.

Week 4: (Geo)Visualization + Choropleths

- Lecture: (Geo)Visualization + Choropleths.
- Tutorial: Mapping deprivation.

Week 5: Spatial Weights

- Lecture: Spatial Weights.
- Tutorial:
  - **TEST 1** (1h): Thursday Oct. 24th
  - Spatial Weights with PySAL.

Week 6: ESDA

- Lecture: Exploratory Spatial Data Analysis (ESDA).
- Tutorial: ESDA in Python.

Week 7: Clustering

- Lecture: Clustering.
- Tutorial: Geodemographic analysis.

Week 8: Point Data

- Lecture: Point Data.
- Tutorial: Exploring Twitter patterns.

Week 9

- Lecture: Assignment preparation.
- Tutorial:
  - **TEST 2** (1h): Thursday Nov. 21st
  - Assignment Clinic

Week 10: (Spatial) causal inference

- Lecture: Spatial causal inference.

- Tutorial: Assignment Clinic.

Week 11: Geographic Data Science in Action

- Lecture: Geographic Data Science *in the wild*.
- Tutorial: Assignment Clinic.

ASSIGNMENT due on Thursday, December 5th-2019.

Bibliography

[AR14]
Luc Anselin and Sergio J Rey. Modern spatial econometrics in practice: a guide to geoda, geodaspace and pysal. 2014.

[AB14]
Daniel Arribas-Bel. Accidental, open and everywhere: emerging data sources for the understanding of cities. *Applied Geography*, 49():45 – 53, 2014. The New Urban World. URL: http://www.sciencedirect.com/science/article/pii/S0143622813002178, doi:http://dx.doi.org/10.1016/j.apgeog.2013.09.012.

[ABGLopezVM19]
Daniel Arribas-Bel, M-À Garcia-López, and Elisabet Viladecans-Marsal. Building (s and) cities: delineating urban areas with a machine learning algorithm. *Journal of Urban Economics*, pages 103217, 2019.

[Bre15]
Cynthia Brewer. *Designing better Maps: A Guide for GIS users*. ESRI press, 2015.

[Don17]
David Donoho. 50 years of data science. *Journal of Computational and Graphical Statistics*, 26(4):745–766, 2017.

[DRSurinach07]
Juan Carlos Duque, Raúl Ramos, and Jordi Suriñach. Supervised regionalization methods: a survey. *International Regional Science Review*, 30(3):195–220, 2007.

[EKS+96]
Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, and others. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96, 226–231. 1996.

[Hai14]
Robert Haining. Spatial data and statistical methods: a chronological overview. In *Handbook of Regional Science*, pages 1277–1294. Springer, 2014.

[LR17]
David Lazer and Jason Radford. Data ex machina: introduction to big data. *Annual Review of Sociology*, 2017.

[McK12]
Wes McKinney. *Python for data analysis: Data wrangling with Pandas, NumPy, and IPython*. O'Reilly Media, Inc., 2012.

[MMSV20]
Ana I Moreno-Monroy, Marcello Schiavina, and Paolo Veneri. Metropolitan areas in the world. delineation and population trends. *Journal of Urban Economics*, pages 103242, 2020.

[RABWng]
Sergio J. Rey, Daniel Arribas-Bel, and Levi J. Wolf. *Geographic Data Science with PySAL and the PyData stack*. CRC press, forthcoming.

[SONeil13]
Rachel Schutt and Cathy O'Neil. *Doing data science: Straight talk from the frontline*. " O'Reilly Media, Inc.", 2013.

[SAB19]
Alex Singleton and Daniel Arribas-Bel. Geographic data science. *Geographical Analysis*, 2019.

[Som18]
James Somers. The scientific paper is obsolete. *The Atlantic*, 2018.

[Sym14]
Jürgen Symanzik. Exploratory spatial data analysis. In *Handbook of Regional Science*, pages 1295–1310. Springer, 2014.

[WB18]
Richard Webber and Roger Burrows. *The Predictive Postcode: The Geodemographic Classification of British Society*. SAGE, 2018.

[Wic14]
Hadley Wickham. Tidy data. *Journal of Statistical Software*, 59(10):??–??, 9 2014. URL: http://www.jstatsoft.org/v59/i10.

Concepts

The concepts in this block are delivered through:

- Two video clips
- Accompanying slides
- `[Optional]` further readings for the interested and curious mind

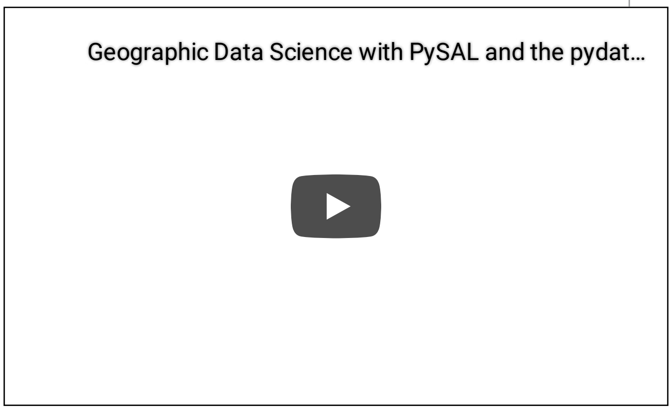
This course

Let us start from the beginning, here is a snapshot of what this course is about! In the following clip, you will find out about the philosophy behind the course, how the content is structured, and why this is all designed like this. And, also, a little bit about the assessment…

Slides

The slides used in the clip are available at:

- [HTML]
- [PDF]



What is *Geographic Data Science*?

Once it is clearer how this course is going to run, let's dive right into why this course is necessary. The following clip is taken from a keynote response by Dani Arribas-Bel at the first Spatial Data Science Conference, organised by CARTO and held in Brooklyn in 2017. The talk provides a bit of background and context, which will hopefully help you understand a bit better what Geographic Data Science is.

Slides

The slides used in the clip are available at:

- [HTML]
- [PDF]

Further readings

To get a better picture, the following readings complement the overview provided above very well:

1. The introductory chapter to "Doing Data Science" [SONeil13], by Cathy O'Neil and Rachel Schutt is general overview of why we needed Data Science and where if came from.

Bonus

The chapter is available free online HTML | PDF

2. A slightly more technical historical perspective on where Data Science came from and where it might go can be found in David Donoho's recent overview [Don17].
3. A geographic take on Data Science, proposing more interaction between Geography and Data Science [SAB19].

Hands-on

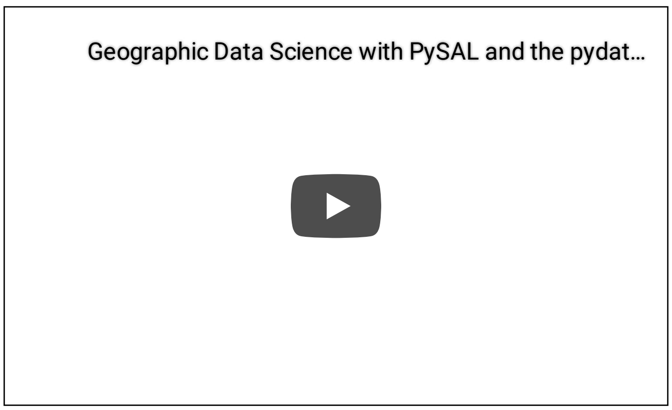
In this first

Following this course interactively

Maybe use this for start up a notebook:

> http://darribas.org/gds19/content/labs/begin.html

Video with walk through Jupyter Lab



Re-write this:

> http://darribas.org/gds19/content/labs/lab_00.html

Software infrastructure for the course

- Point to available guides + install options

Downloading data for this course

Do-It-Yourself

To do:

- Make sure you have the setup installed and/or access to a campus computer to complete the course
- Launch JupyterLab, and explore

Concepts

The ideas behind this block are better communicated through narrative than video or lectures. Hence, the concepts section are delivered through a few references you are expected to read. These will total up about one and a half hours of your focused time.

Open Science

The first part of this block is about setting the philosophical background. Why do we care about the processes and tools we use when we do computational work? Where do the current paradigm come from? Are we on the verge of a new model? For all of this, we we have two reads to set the tone. Make sure to get those in first thing before moving on to the next bits.

- First half of Chapter 1 in "Geographic Data Science with PySAL and the PyData stack" [RABWng].

Read the chapter here. Estimated time: 15min.

- The 2018 Atlantic piece *"The scientific paper is obsolete"* on computational notebooks, by James Somers [Som18].

Read the piece here. Estimated time: 35min.

Modern Scientific Tools

Once we know a bit more about why we should care about the tools we use, let's dig into those that will underpin much of this course. This part is interesting in itself, but will also valuable to better understand the practical aspects of the course. Again, we have two reads here to set the tone and complement the practical introduction we saw in the Hands-on and DIY parts of the previous block. We are closing the circle here:

- Second half of Chapter 1 in "Geographic Data Science with PySAL and the PyData stack" [RABWng].
- The chapter in the GIS&T Book of Knowledge on computational notebooks, by Geoff Boeing and Dani Arribas-Bel.

Read the chapter here. Estimated time: 15min.

Hands-on

Once we know a bit about what computational notebooks are and why we should care about them, let's jump to using them! This section introduces you to using Python for manipulating tabular data. Please read through it carefully and pay attention to how ideas about manipulating data are translated into Python code that "does stuff". For this part, you can read directly from the course website, although it is recommended you follow the section interactively by running code on your own.

Once you have read through and have a bit of a sense of how things work, jump on the Do-It-Yourself section, which will provide you with a challenge to complete it on your own, and will allow you to put what you have already learnt to good use. Happy hacking!

Data munging

Real world datasets are messy. There is no way around it: datasets have "holes" (missing data), the amount of formats in which data can be stored is endless, and the best structure to share data is not always the optimum to analyze them, hence the need to munge them. As has been correctly pointed out in many outlets (e.g.), much of the time spent in what is called (Geo-)Data Science is related not only to sophisticated modeling and insight, but has to do with much more basic and less exotic tasks such as obtaining data, processing, turning them into a shape that makes analysis possible, and exploring it to get to know their basic properties.

For how labor intensive and relevant this aspect is, there is surprisingly very little published on patterns, techniques, and best practices for quick and efficient data cleaning, manipulation, and transformation. In this session, you will use a few real world datasets and learn how to process them into Python so they can be transformed and manipulated, if necessary, and analyzed. For this, we will introduce some of the bread and butter of data analysis and scientific computing in Python. These are fundamental tools that are constantly used in almost any task relating to data analysis.

This notebook covers the basic and the content that is expected to be learnt by every student. We use a prepared dataset that saves us much of the more intricate processing that goes beyond the introductory level the session is aimed at. As a companion to this introduction, there is an additional notebook (see link on the website page for Lab 01) that covers how the dataset used here was prepared from raw data downloaded from the internet, and includes some additional exercises you can do if you want dig deeper into the content of this lab.

In this notebook, we discuss several patterns to clean and structure data properly, including tidying, subsetting, and aggregating; and we finish with some basic visualization. An additional extension presents more advanced tricks to manipulate tabular data.

Before we get our hands data-dirty, let us import all the additional libraries we will need, so we can get that out of the way and focus on the task at hand:

```
# This ensures visualizations are plotted inside the notebook
%matplotlib inline

import os              # This provides several system utilities
import pandas as pd    # This is the workhorse of data munging in Python
import seaborn as sns  # This allows us to efficiently and beautifully plot
```

Dataset

We will be exploring some demographic characteristics in Liverpool. To do that, we will use a dataset that contains population counts, split by ethnic origin. These counts are aggregated at the Lower Layer Super Output Area (LSOA from now on). LSOAs are an official Census geography defined by the Office of National Statistics. You can think of them, more or less, as neighbourhoods. Many data products (Census, deprivation indices, etc.) use LSOAs as one of their main geographies.

To make things easier, we will read data from a file posted online so, for now, you do not need to download any dataset:

```python
# Read table
db = pd.read_csv("https://darribas.org/gds_course/content/data/liv_pop.csv",
                 index_col='GeographyCode')
```

> **Important**
>
> Make sure you are connected to the internet when you run this cell

Let us stop for a minute to learn how we have read the file. Here are the main aspects to keep in mind:

- We are using the method `read_csv` from the `pandas` library, which we have imported with the alias `pd`.
- In this form, all that is required is to pass the path to the file we want to read, which in this case is a web address.
- The argument `index_col` is not strictly necessary but allows us to choose one of the columns as the index of the table. More on indices below.
- We are using `read_csv` because the file we want to read is in the `csv` format. However, `pandas` allows for many more formats to be read and write. A full list of formats supported may be found here.
- To ensure we can access the data we have read, we store it in an *object* that we call `db`. We will see more on what we can do with it below but, for now, just keep in mind that allows us to save the result of `read_csv`.

> **Alternative**
>
> Instead of reading the file directly off the web, it is possible to download it manually, store it on your computer, and read it locally. To do that, you can follow these steps:
>
> 1. Download the file by right-clicking on this link and saving the file
> 2. Place the file on the *same folder as the notebook* where you intend to read it
> 3. Replace the code in the cell above by:
>
> ```python
> db = pd.read_csv("liv_pop.csv", index_col="GeographyCode")
> ```

Data, sliced and diced

Now we are ready to start playing and interrogating the dataset! What we have at our fingertips is a table that summarizes, for each of the LSOAs in Liverpool, how many people live in each, by the region of the world where they were born. Now, let us learn a few cool tricks built into `pandas` that work out-of-the box with a table like ours.

Inspect

Inspecting what it looks like. We can check the top (bottom) X lines of the table by passing X to the method `head` (`tail`). For example, for the top/bottom five lines:

```python
db.head()
```

	Europe	Africa	Middle East and Asia	The Americas and the Caribbean	Antarctica and Oceania
GeographyCode					
E01006512	910	106	840	24	0
E01006513	2225	61	595	53	7
E01006514	1786	63	193	61	5
E01006515	974	29	185	18	2
E01006518	1531	69	73	19	4

```
db.tail()
```

	Europe	Africa	Middle East and Asia	The Americas and the Caribbean	Antarctica and Oceania
GeographyCode					
E01033764	2106	32	49	15	0
E01033765	1277	21	33	17	3
E01033766	1028	12	20	8	7
E01033767	1003	29	29	5	1
E01033768	1016	69	111	21	6

Or getting an overview of the table:

```
db.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 298 entries, E01006512 to E01033768
Data columns (total 5 columns):
 #   Column                          Non-Null Count  Dtype
---  ------                          --------------  -----
 0   Europe                          298 non-null    int64
 1   Africa                          298 non-null    int64
 2   Middle East and Asia            298 non-null    int64
 3   The Americas and the Caribbean  298 non-null    int64
 4   Antarctica and Oceania          298 non-null    int64
dtypes: int64(5)
memory usage: 14.0+ KB
```

Summarise

Or of the *values* of the table:

```
db.describe()
```

	Europe	Africa	Middle East and Asia	The Americas and the Caribbean	Antarctica and Oceania
count	298.00000	298.000000	298.000000	298.000000	298.000000
mean	1462.38255	29.818792	62.909396	8.087248	1.949664
std	248.67329	51.606065	102.519614	9.397638	2.168216
min	731.00000	0.000000	1.000000	0.000000	0.000000
25%	1331.25000	7.000000	16.000000	2.000000	0.000000
50%	1446.00000	14.000000	33.500000	5.000000	1.000000
75%	1579.75000	30.000000	62.750000	10.000000	3.000000
max	2551.00000	484.000000	840.000000	61.000000	11.000000

Note how the output is also a `DataFrame` object, so you can do with it the same things you would with the original table (e.g. writing it to a file).

In this case, the summary might be better presented if the table is “transposed”:

```
db.describe().T
```

	count	mean	std	min	25%	50%	
Europe	298.0	1462.382550	248.673290	731.0	1331.25	1446.0	1
Africa	298.0	29.818792	51.606065	0.0	7.00	14.0	
Middle East and Asia	298.0	62.909396	102.519614	1.0	16.00	33.5	
The Americas and the Caribbean	298.0	8.087248	9.397638	0.0	2.00	5.0	
Antarctica and Oceania	298.0	1.949664	2.168216	0.0	0.00	1.0	

Equally, common descriptive statistics are also available:

```
# Obtain minimum values for each table
db.min()
```

```
Europe                            731
Africa                              0
Middle East and Asia                1
The Americas and the Caribbean      0
Antarctica and Oceania              0
dtype: int64
```

```
# Obtain minimum value for the column `Europe`
db['Europe'].min()
```

```
731
```

Note here how we have restricted the calculation of the maximum value to one column only.

Similarly, we can restrict the calculations to a single row:

```
# Obtain standard deviation for the row `E01006512`,
# which represents a particular LSOA
db.loc['E01006512', :].std()
```

```
457.8842648530303
```

Create new columns

We can generate new variables by applying operations on existing ones. For example, we can calculate the total population by area. Here is a couple of ways to do it:

```
# Longer, hardcoded
total = db['Europe'] + db['Africa'] + db['Middle East and Asia'] + \
        db['The Americas and the Caribbean'] + db['Antarctica and Oceania']
# Print the top of the variable
total.head()
```

```
GeographyCode
E01006512    1880
E01006513    2941
E01006514    2108
E01006515    1208
E01006518    1696
dtype: int64
```

```
# One shot
total = db.sum(axis=1)
# Print the top of the variable
total.head()
```

```
GeographyCode
E01006512    1880
E01006513    2941
E01006514    2108
E01006515    1208
E01006518    1696
dtype: int64
```

Note how we are using the command `sum`, just like we did with `max` or `min` before but, in this case, we are not applying it over columns (e.g. the max of each column), but over rows, so we get the total sum of populations by areas.

Once we have created the variable, we can make it part of the table:

```
db['Total'] = total
db.head()
```

	Europe	Africa	Middle East and Asia	The Americas and the Caribbean	Antarctica and Oceania	Tota
GeographyCode						
E01006512	910	106	840	24	0	188
E01006513	2225	61	595	53	7	294
E01006514	1786	63	193	61	5	210
E01006515	974	29	185	18	2	120
E01006518	1531	69	73	19	4	169

A different spin on this is assigning new values: we can generate new variables with scalars, and modify those:

```
# New variable with all ones
db['ones'] = 1
db.head()
```

	Europe	Africa	Middle East and Asia	The Americas and the Caribbean	Antarctica and Oceania	Tota
GeographyCode						
E01006512	910	106	840	24	0	188
E01006513	2225	61	595	53	7	294
E01006514	1786	63	193	61	5	210
E01006515	974	29	185	18	2	120
E01006518	1531	69	73	19	4	169

And we can modify specific values too:

```
db.loc['E01006512', 'ones'] = 3
db.head()
```

	Europe	Africa	Middle East and Asia	The Americas and the Caribbean	Antarctica and Oceania	Tota
GeographyCode						
E01006512	910	106	840	24	0	188
E01006513	2225	61	595	53	7	294
E01006514	1786	63	193	61	5	210
E01006515	974	29	185	18	2	120
E01006518	1531	69	73	19	4	169

Delete columns

Permanently deleting variables is also within reach of one command:

```
del db['ones']
db.head()
```

	Europe	Africa	Middle East and Asia	The Americas and the Caribbean	Antarctica and Oceania	Tota
GeographyCode						
E01006512	910	106	840	24	0	188
E01006513	2225	61	595	53	7	294
E01006514	1786	63	193	61	5	210
E01006515	974	29	185	18	2	120
E01006518	1531	69	73	19	4	169

Index-based queries

We have already seen how to subset parts of a `DataFrame` if we know exactly which bits we want. For example, if we want to extract the total and European population of the first four areas in the table, we use `loc` with lists:

```
eu_tot_first4 = db.loc[['E01006512', 'E01006513', 'E01006514', 'E01006515'], \
                      ['Total', 'Europe']]
eu_tot_first4
```

	Total	Europe
GeographyCode		
E01006512	1880	910
E01006513	2941	2225
E01006514	2108	1786
E01006515	1208	974

Condition-based queries

However, sometimes, we do not know exactly which observations we want, but we do know what conditions they need to satisfy (e.g. areas with more than 2,000 inhabitants). For these cases, `DataFrames` support selection based on conditions. Let us see a few examples. Suppose we want to select…

… *areas with more than 2,500 people in Total*:

```
m5k = db.loc[db['Total'] > 2500, :]
m5k
```

	Europe	Africa	Middle East and Asia	The Americas and the Caribbean	Antarctica and Oceania	Tota
GeographyCode						
E01006513	2225	61	595	53	7	294
E01006747	2551	163	812	24	2	355
E01006751	1843	139	568	21	1	257

… *areas where there are no more than 750 Europeans*:

```
nm5ke = db.loc[db['Europe'] < 750, :]
nm5ke
```

	Europe	Africa	Middle East and Asia	The Americas and the Caribbean	Antarctica and Oceania	Tota
GeographyCode						
E01033757	731	39	223	29	3	102

… *areas with exactly ten person from Antarctica and Oceania*:

```
oneOA = db.loc[db['Antarctica and Oceania'] == 10, :]
oneOA
```

	Europe	Africa	Middle East and Asia	The Americas and the Caribbean	Antarctica and Oceania	Tota
GeographyCode						
E01006679	1353	484	354	31	10	223

Pro-tip: these queries can grow in sophistication with almost no limits. For example, here is a case where we want to find out the areas where European population is less than half the population:

```
eu_lth = db.loc[(db['Europe'] * 100. / db['Total']) < 50, :]
eu_lth
```

	Europe	Africa	Middle East and Asia	The Americas and the Caribbean	Antarctica and Oceania	Tota
GeographyCode						
E01006512	910	106	840	24	0	188

If you are interested, more detail about `query` is available on the [official pandas documentation](#).

All the condition-based queries above are expressed using the `loc` operator. This is a powerful way and, since it shares syntax with index-based queries, it is also easier to remember. However, sometimes querying using `loc` involves a lot of quotation marks, parenthesis, etc. A more streamlined approach for condition-based queries of rows is provided by the `query` engine. Using this approach, we express everything in our query on a single string, or piece of text, and that is evaluated in the table at once. For example, we can run the same operation as in the first query above with the following syntax:

```
m5k_query = db.query("Total > 2500")
```

If we want to combine operations, this is also possible:

```
m5k_query2 = db.query("(Total > 2500) & (Total < 10000)")
```

Note that, in these cases, using `query` results in code that is much more streamlined and easier to read. However, `query` is not perfect and, particularly for more sophisticated queries, it does not afford the same degree of flexibility. For example, the last query we had using `loc` would not be possible using `query`.

Combining queries

Now all of these queries can be combined with each other, for further flexibility. For example, imagine we want areas with more than 25 people from the Americas and Caribbean, but less than 1,500 in total:

```
ac25_l500 = db.loc[(db['The Americas and the Caribbean'] > 25) & \
                   (db['Total'] < 1500), :]
ac25_l500
```

	Europe	Africa	Middle East and Asia	The Americas and the Caribbean	Antarctica and Oceania	Tota
GeographyCode						
E01033750	1235	53	129	26	5	144
E01033752	1024	19	114	33	6	119
E01033754	1262	37	112	32	9	145
E01033756	886	31	221	42	5	118
E01033757	731	39	223	29	3	102
E01033761	1138	52	138	33	11	137

Sorting

Among the many operations `DataFrame` objects support, one of the most useful ones is to sort a table based on a given column. For example, imagine we want to sort the table by total population:

```
db_pop_sorted = db.sort_values('Total', ascending=False)
db_pop_sorted.head()
```

	Europe	Africa	Middle East and Asia	The Americas and the Caribbean	Antarctica and Oceania	Tota
GeographyCode						
E01006747	2551	163	812	24	2	355
E01006513	2225	61	595	53	7	294
E01006751	1843	139	568	21	1	257
E01006524	2235	36	125	24	11	243
E01006787	2187	53	75	13	2	233

If you inspect the help of `db.sort_values`, you will find that you can pass more than one column to sort the table by. This allows you to do so-called hiearchical sorting: sort first based on one column, if equal then based on another column, etc.

Visual exploration

The next step to continue exploring a dataset is to get a feel for what it looks like, visually. We have already learnt how to unconver and inspect specific parts of the data, to check for particular cases we might be intersted in. Now we will see how to plot the data to get a sense of the overall distribution of values. For that, we will be using the Python library [`seaborn`](seaborn).

- Histograms.

One of the most common graphical devices to display the distribution of values in a variable is a histogram. Values are assigned into groups of equal intervals, and the groups are plotted as bars rising as high as the number of values into the group.

A histogram is easily created with the following command. In this case, let us have a look at the shape of the overall population:

```
_ = sns.distplot(db['Total'], kde=False)
```

../../_images/lab_B_52_0.png

Note we are using `sns` instead of `pd`, as the function belongs to `seaborn` instead of `pandas`.

We can quickly see most of the areas contain somewhere between 1,200 and 1,700 people, approx. However, there are a few areas that have many more, even up to 3,500 people.

An additional feature to visualize the density of values is called `rug`, and adds a little tick for each value on the horizontal axis:

```
_ = sns.distplot(db['Total'], kde=False, rug=True)
```

../../_images/lab_B_54_0.png

- Kernel Density Plots

Histograms are useful, but they are artificial in the sense that a continuous variable is made discrete by turning the values into discrete groups. An alternative is kernel density estimation (KDE), which produces an empirical density function:

```
_ = sns.kdeplot(db['Total'], shade=True)
```

../../_images/lab_B_56_0.png

- Line and bar plots

Another very common way of visually displaying a variable is with a line or a bar chart. For example, if we want to generate a line plot of the (sorted) total population by area:

```
_ = db['Total'].sort_values(ascending=False).plot()
```

```
/opt/conda/lib/python3.7/site-packages/pandas/plotting/_matplotlib/core.py:1235:
UserWarning: FixedFormatter should only be used together with FixedLocator
  ax.set_xticklabels(xticklabels)
```

../../_images/lab_B_58_1.png

For a bar plot all we need to do is to change from `plot` to `plot.bar`. Since there are many neighbourhoods, let us plot only the ten largest ones (which we can retrieve with `head`):

```
_ = db['Total'].sort_values(ascending=False)\
               .head(10)\
               .plot.bar()
```

../../_images/lab_B_60_0.png

We can turn the plot around by displaying the bars horizontally (see how it’s just changing `bar` for `barh`). Let’s display now the top 50 areas and, to make it more readable, let us expand the plot’s height:

```
_ = db['Total'].sort_values()\
               .head(50)\
               .plot.barh(figsize=(6, 20))
```

../../_images/lab_B_62_0.png

Un/tidy data

Warning

This section is a bit more advanced and hence considered optional. Fell free to skip it, move to the next, and return later when you feel more confident.

> *Happy families are all alike; every unhappy family is unhappy in its own way.*
> Leo Tolstoy.

Once you can read your data in, explore specific cases, and have a first visual approach to the entire set, the next step can be preparing it for more sophisticated analysis. Maybe you are thinking of modeling it through regression, or on creating subgroups in the dataset with particular characteristics, or maybe you simply need to present summary measures that relate to a slightly different arrangement of the data than you have been presented with.

For all these cases, you first need what statistician, and general R wizard, Hadley Wickham calls *"tidy data"*. The general idea to "tidy" your data is to convert them from whatever structure they were handed in to you into one that allows convenient and standardized manipulation, and that supports directly inputting the data into what he calls "*tidy*" analysis tools. But, at a more practical level, what is exactly *"tidy data"*? In Wickham's own words:

> *Tidy data is a standard way of mapping the meaning of a dataset to its structure. A dataset is messy or tidy depending on how rows, columns and tables are matched up with observations, variables and types.*

He then goes on to list the three fundamental characteristics of *"tidy data"*:

1. Each variable forms a column.
2. Each observation forms a row.
3. Each type of observational unit forms a table.

If you are further interested in the concept of *"tidy data"*, I recommend you check out the original paper (open access) and the public repository associated with it.

Let us bring in the concept of "*tidy data*" to our own Liverpool dataset. First, remember its structure:

```
db.head()
```

	Europe	**Africa**	**Middle East and Asia**	**The Americas and the Caribbean**	**Antarctica and Oceania**	**Total**
GeographyCode						
E01006512	910	106	840	24	0	1880
E01006513	2225	61	595	53	7	2941
E01006514	1786	63	193	61	5	2108
E01006515	974	29	185	18	2	1208
E01006518	1531	69	73	19	4	1696

Thinking through *tidy* lenses, this is not a tidy dataset. It is not so for each of the three conditions:

- Starting by the last one (*each type of observational unit forms a table*), this dataset actually contains not one but two observational units: the different areas of Liverpool, captured by `GeographyCode`; *and* subgroups of an area. To *tidy* up this aspect, we can create two different tables:

```
# Assign column `Total` into its own as a single-column table
db_totals = db[['Total']]
db_totals.head()
```

GeographyCode	Total
E01006512	1880
E01006513	2941
E01006514	2108
E01006515	1208
E01006518	1696

```
# Create a table `db_subgroups` that contains every column in `db` without `Total`
db_subgroups = db.drop('Total', axis=1)
db_subgroups.head()
```

GeographyCode	Europe	Africa	Middle East and Asia	The Americas and the Caribbean	Antarctica and Oceania
E01006512	910	106	840	24	0
E01006513	2225	61	595	53	7
E01006514	1786	63	193	61	5
E01006515	974	29	185	18	2
E01006518	1531	69	73	19	4

Note we use `drop` to exclude “Total”, but we could also use a list with the names of all the columns to keep. Additionally, notice how, in this case, the use of `drop` (which leaves `db` untouched) is preferred to that of `del` (which permanently removes the column from `db`).

At this point, the table `db_totals` is tidy: every row is an observation, every table is a variable, and there is only one observational unit in the table.

The other table (`db_subgroups`), however, is not entirely tidied up yet: there is only one observational unit in the table, true; but every row is not an observation, and there are variable values as the names of columns (in other words, every column is not a variable).

To obtain a fully tidy version of the table, we need to re-arrange it in a way that every row is a population subgroup in an area, and there are three variables: `GeographyCode`, population subgroup, and population count (or frequency).

Because this is actually a fairly common pattern, there is a direct way to solve it in `pandas`:

```
tidy_subgroups = db_subgroups.stack()
tidy_subgroups.head()
```

```
GeographyCode
E01006512      Europe                              910
               Africa                              106
               Middle East and Asia                840
               The Americas and the Caribbean       24
               Antarctica and Oceania                0
dtype: int64
```

The method `stack`, well, "stacks" the different columns into rows. This fixes our "tidiness" problems but the type of object that is returning is not a `DataFrame`:

```
type(tidy_subgroups)
```

```
pandas.core.series.Series
```

It is a `Series`, which really is like a `DataFrame`, but with only one column. The additional information (`GeographyCode` and population group) are stored in what is called an multi-index. We will skip these for now, so we would really just want to get a `DataFrame` as we know it out of the `Series`. This is also one line of code away:

```
# Unfold the multi-index into different, new columns
tidy_subgroupsDF = tidy_subgroups.reset_index()
tidy_subgroupsDF.head()
```

	GeographyCode	level_1	0
0	E01006512	Europe	910
1	E01006512	Africa	106
2	E01006512	Middle East and Asia	840
3	E01006512	The Americas and the Caribbean	24
4	E01006512	Antarctica and Oceania	0

To which we can apply to renaming to make it look better:

```
tidy_subgroupsDF = tidy_subgroupsDF.rename(columns={'level_1': 'Subgroup', 0: 'Freq'})
tidy_subgroupsDF.head()
```

	GeographyCode	Subgroup	Freq
0	E01006512	Europe	910
1	E01006512	Africa	106
2	E01006512	Middle East and Asia	840
3	E01006512	The Americas and the Caribbean	24
4	E01006512	Antarctica and Oceania	0

Now our table is fully tidied up!

Grouping, transforming, aggregating

One of the advantage of tidy datasets is they allow to perform advanced transformations in a more direct way. One of the most common ones is what is called "group-by" operations. Originated in the world of databases, these operations allow you to group observations in a table by one of its labels, index, or category, and apply operations on the data group by group.

For example, given our tidy table with population subgroups, we might want to compute the total sum of population by each group. This task can be split into two different ones:

- Group the table in each of the different subgroups.
- Compute the sum of `Freq` for each of them.

To do this in `pandas`, meet one of its workhorses, and also one of the reasons why the library has become so popular: the `groupby` operator.

```
pop_grouped = tidy_subgroupsDF.groupby('Subgroup')
pop_grouped
```

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x7f840a5cf5d0>
```

The object `pop_grouped` still hasn't computed anything, it is only a convenient way of specifying the grouping. But this allows us then to perform a multitude of operations on it. For our example, the sum is calculated as follows:

```
pop_grouped.sum()
```

	Freq
Subgroup	
Africa	8886
Antarctica and Oceania	581
Europe	435790
Middle East and Asia	18747
The Americas and the Caribbean	2410

Similarly, you can also obtain a summary of each group:

```
pop_grouped.describe()
```

	Freq						
	count	mean	std	min	25%	50%	7
Subgroup							
Africa	298.0	29.818792	51.606065	0.0	7.00	14.0	
Antarctica and Oceania	298.0	1.949664	2.168216	0.0	0.00	1.0	
Europe	298.0	1462.382550	248.673290	731.0	1331.25	1446.0	1
Middle East and Asia	298.0	62.909396	102.519614	1.0	16.00	33.5	
The Americas and the Caribbean	298.0	8.087248	9.397638	0.0	2.00	5.0	

We will not get into it today as it goes beyond the basics we want to conver, but keep in mind that `groupby` allows you to not only call generic functions (like `sum` or `describe`), but also your own functions. This opens the door for virtually any kind of transformation and aggregation possible.

Additional lab materials

The following provide a good "next step" from some of the concepts and tools covered in the lab and DIY sections of this block:

- This NY Times article does a good job at conveying the relevance of data "cleaning" and munging.
- A good introduction to data manipulation in Python is Wes McKinney's "Python for Data Analysis" [McK12].
- To explore further some of the visualization capabilities in at your fingertips, the Python library `seaborn` is an excellent choice. Its online tutorial is a fantastic place to start.
- A good extension is Hadley Wickham' "Tidy data" paper [Wic14], which presents a very popular way of organising tabular data for efficient manipulation.

Do-It-Yourself

```
import pandas
```

This section is all about you taking charge of the steering wheel and choosing your own adventure. For this block, we are going to use what we've learnt before to take a look at a dataset of casualties in the war in Afghanistan. The data was originally released by

Wikileaks, and the version we will use is published by The Guardian.

You can read a bit more about the data at The Guardian's data blog

Data preparation

Before you can set off on your data journey, the dataset needs to be read, and there's a couple of details we will get out of the way so it is then easier for you to start working.

The data are published on a Google Sheet you can check out at:

> https://docs.google.com/spreadsheets/d/1EAx8_ksSCmoWW_SlhFyq2QrRn0FNNhcg1TtDFJzZRgc/edit?hl=en#gid=1

As you will see, each row includes casualties recorded month by month, split by Taliban, Civilians, Afghan forces, and NATO.

To read it into a Python session, we need to slightly modify the URL to access it into:

```
url = ("https://docs.google.com/spreadsheets/d/"\
       "1EAx8_ksSCmoWW_SlhFyq2QrRn0FNNhcg1TtDFJzZRgc/"\
       "export?format=csv&gid=1")
url
```

```
'https://docs.google.com/spreadsheets/d/1EAx8_ksSCmoWW_SlhFyq2QrRn0FNNhcg1TtDFJzZRgc
/export?format=csv&gid=1'
```

Note how we split the url into three lines so it is more readable in narrow screens. The result however, stored in `url`, is the same as one long string.

This allows us to read the data straight into a DataFrame, as we have done in the previous session:

```
db = pandas.read_csv(url, skiprows=[0, -1])
```

Note also we use the `skiprows=[0, -1]` to avoid reading the top (`0`) and bottom (`-1`) rows which, if you check on the Google Sheet, involves the title of the table.

Now we are good to go!

```
db.head()
```

	Year	Month	Taliban	Civilians	Afghan forces	Nato (detailed in spreadsheet)	Nato - official figures
0	2004.0	January	15	51	23	NaN	11.0
1	2004.0	February	NaN	7	4	5	2.0
2	2004.0	March	19	2	NaN	2	3.0
3	2004.0	April	5	3	19	NaN	3.0
4	2004.0	May	18	29	56	6	9.0

Tasks

Now, the challenge is to put to work what we have learnt in this block. For that, the suggestion is that you carry out an analysis of the Afghan Logs in a similar way as how we looked at population composition in Liverpool. These are of course very different datasets reflecting immensely different realities. Their structure, however, is relatively parallel: both capture counts aggregated by a spatial (neighbourhood) or temporal unit (month), and each count is split by a few categories.

Try to answer the following questions:

- Obtain the minimum number of civilian casualties (in what month was that?)
- How many NATO casualties were registered in August 2008?

- What is the month with the most total number of casualties?
- Can you make a plot of the distribution of casualties over time?

Tip

You will need to first create a column with total counts

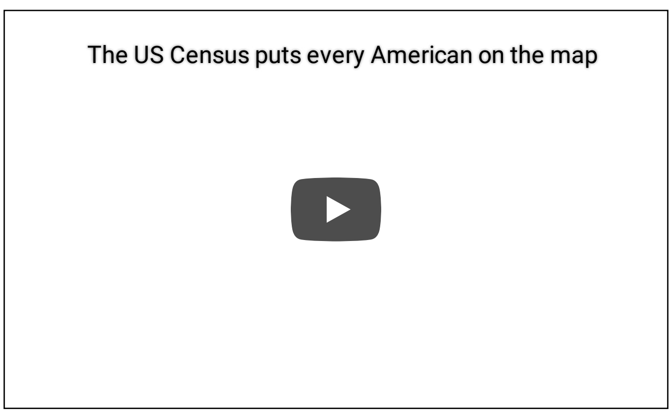
Concepts

This blocks explore spatial data, old and new. We start with an overview of traditional datasets, discussing their benefits and challenges for social scientists; then we move on to new forms of data, and how they pose different challenges, but also exciting opportunities. These two areas are covered with clips and slides that can be complemented with readings. Once conceptual areas are covered, we jump into [working with spatial data in Python](), which will prepare you for your [own adventure]() in exploring spatial data.

"Good old" (geo) data

To understand what is new in new forms of data, it is useful to begin by considering traditional data. In this section we look at the main characteristics of traditional data available to Social Scientists. Warm up before the main part coming up next!

Before you jump on the clip, please watch the following video by the US Census Burearu, which will be discussed:

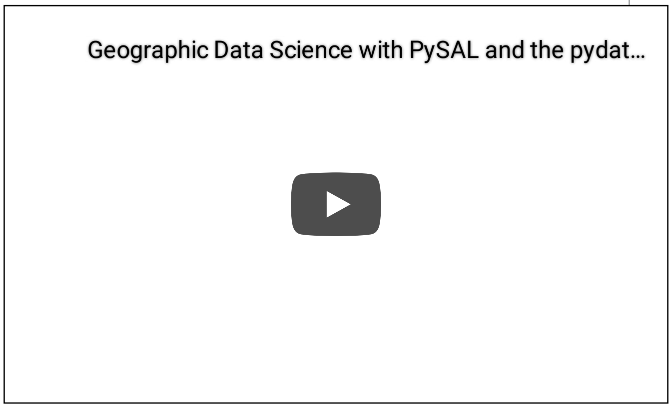


Then go on to the following clip, which will help you put the Census Bureau's view in perspective:

Slides

The slides used in the clip are available at:

- [HTML]
- [PDF]

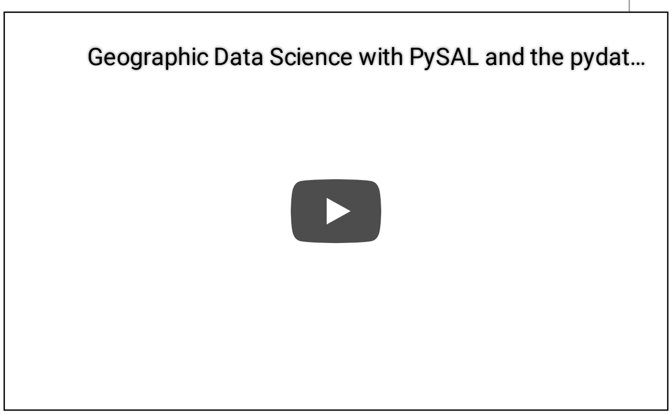


New forms of (geo) data

Slides

The slides used in the clip are available at:

- [HTML]
- [PDF]



This section discusses two references in particular:

- "Data Ex-Machina", by Lazer & Radford [LR17]
- And the accidental data paper by Dani Arribas-Bel [AB14]

Although both papers are discussed in the clip, if you are interested in the ideas mentioned, do go to the original sources as they provide much more detail and nuance.

Hands-on

Mapping in Python

```
%matplotlib inline

import geopandas
import osmnx
import contextily as cx
from keplergl import KeplerGl
import matplotlib.pyplot as plt
```

In this lab, we will learn how to load, manipulate and visualize spatial data. In some senses, spatial data are usually included simply as "one more column" in a table. However, *spatial is special* sometimes and there are few aspects in which geographic data differ from standard numerical tables. In this session, we will extend the skills developed in the previous one about non-spatial data, and combine them. In the process, we will discover that, although with some particularities, dealing with spatial data in Python largely resembles dealing with non-spatial data.

Datasets

To learn these concepts, we will be playing with three main datasets. Same as in the previous block, these datasets can be loaded dynamically from the web, or you can download them manually, keep a copy on your computer, and load them from there.

> **Important**
>
> Make sure you are connected to the internet when you run these cells as they need to access data hosted online

Cities

First we will use a polygon geography. We will use an open dataset that contains the boundaries of Spanish cities. We can read it into an object named `cities` by:

```
cities = geopandas.read_file("https://ndownloader.figshare.com/files/20232174")
```

> **Note**
>
> This dataset is derived from [ABGLopezVM19], which proposes a machine learning algorithm to delineate city boundaries from building footprints.

Streets

In addition to polygons, we will play with a line layer. For that, we are going to use a subset of street network from the Spanish city of Zaragoza.

The data is available on the following web address:

```
url = ("https://github.com/geochicasosm/lascallesdelasmujeres"\
       "/raw/master/data/zaragoza/final_tile.geojson")
url
```

```
'https://github.com/geochicasosm/lascallesdelasmujeres/raw/master/data/zaragoza/fi
nal_tile.geojson'
```

> **Note**
>
> This dataset froms of a project called "Las calles de las mujeres", a community-driven initiative exploring the extent to which streets are named after women.
>
> Check out more about the project, including an interactive map at:
>
> https://geochicasosm.github.i

And you can read it into an object called `streets` with:

```
streets = geopandas.read_file(url)
```

Bars

The final dataset we will rely on is a set of points demarcating the location of bars in Zaragoza. To obtain it, we will use `osmnx`, a Python library that allows us to query OpenStreetMap. Note that we use the method `pois_from_place`, which queries for points of interest (POIs, or `pois`) in a particular place (Zaragoza in this case). In addition, we can specify a set of tags to delimit the query. We use this to ask *only* for amenities of the type “bar”:

```
pois = osmnx.pois.pois_from_place("Zaragoza, Spain",
                                  tags={"amenity": "bar"}
                                 )
```

You do not need to know at this point what happens behind the scenes when we run `pois_from_place` but, if you are curious, we are making a query to OpenStreetMap (almost as if you typed “bars in Zaragoza, Spain” within Google Maps) and getting the response as a table of data, instead of as a website with an interactive map. Pretty cool, huh?

Inspecting spatial data

The most direct way to get from a file to a quick visualization of the data is by loading it as a `GeoDataFrame` and calling the `plot` command. The main library employed for all of this is `geopandas` which is a geospatial extension of the `pandas` library, already introduced before. `geopandas` supports the same functionality that `pandas` does, plus a wide range of spatial extensions that make manipulation and general “munging” of spatial data similar to non-spatial tables.

In two lines of code, we will obtain a graphical representation of the spatial data contained in a file that can be in many formats; actually, since it uses the same drivers under the hood, you can load pretty much the same kind of vector files that Desktop GIS packages like QGIS permit. Let us start by plotting single layers in a crude but quick form, and we will build style and sophistication into our plots later on.

Polygons

Now `lsoas` is a `GeoDataFrame`. Very similar to a traditional, non-spatial `DataFrame`, but with an additional column called `geometry`:

```
cities.head()
```

	city_id	n_buildings	geometry
0	ci000	2348	POLYGON ((385390.071 4202949.446, 384488.697 4...
1	ci001	2741	POLYGON ((214893.033 4579137.558, 215258.185 4...
2	ci002	5472	POLYGON ((690674.281 4182188.538, 691047.526 4...
3	ci003	14608	POLYGON ((513378.282 4072327.639, 513408.853 4...
4	ci004	2324	POLYGON ((206989.081 4129478.031, 207275.702 4...

This allows us to quickly produce a plot by executing the following line:

```
cities.plot()
```

```
<AxesSubplot:>
```

../../_images/lab_C_21_1.png

This might not be the most aesthetically pleasant visual representation of the LSOAs geography, but it is hard to argue it is not quick to produce. We will work on styling and customizing spatial plots later on.

Pro-tip: if you call a single row of the `geometry` column, it'll return a small plot ith the shape:

```
cities.loc[0, 'geometry']
```

../../_images/lab_C_24_0.svg

Lines

Similarly to the polygon case, if we pick the `"geometry"` column of a table with lines, a single row will display the geometry as well:

```
streets.loc[0, 'geometry']
```

../../_images/lab_C_27_0.svg

A quick plot is similarly generated by:

```
streets.plot()
```

```
<AxesSubplot:>
```

../../_images/lab_C_29_1.png

Again, this is not the prettiest way to display the streets maybe, and you might want to change a few parameters such as colors, etc. All of this is possible, as we will see below, but this gives us a quick check of what lines look like.

Points

Points take a similar approach for quick plotting:

```
pois.plot()
```

```
<AxesSubplot:>
```

../../_images/lab_C_33_1.png

Styling plots

It is possible to tweak several aspects of a plot to customize if to particular needs. In this section, we will explore some of the basic elements that will allow us to obtain more compelling maps.

NOTE: some of these variations are very straightforward while others are more intricate and require tinkering with the internal parts of a plot. They are not necessarily organized by increasing level of complexity.

Changing transparency

The intensity of color of a polygon can be easily changed through the `alpha` attribute in plot. This is specified as a value betwee zero and one, where the former is entirely transparent while the latter is the fully opaque (maximum intensity):

```
pois.plot(alpha=0.1)
```

```
<AxesSubplot:>
```

../../_images/lab_C_38_1.png

Removing axes

Although in some cases, the axes can be useful to obtain context, most of the times maps look and feel better without them. Removing the axes involves wrapping the plot into a figure, which takes a few more lines of aparently useless code but that, in time, it will allow you to tweak the map further and to create much more flexible designs:

```
# Setup figure and axis
f, ax = plt.subplots(1)
# Plot layer of polygons on the axis
cities.plot(ax=ax)
# Remove axis frames
ax.set_axis_off()
# Display
plt.show()
```

../../_images/lab_C_41_0.png

Let us stop for a second a study each of the previous lines:

1. We have first created a figure named `f` with one axis named `ax` by using the command `plt.subplots` (part of the library `matplotlib`, which we have imported at the top of the notebook). Note how the method is returning two elements and we can assign each of them to objects with different name (`f` and `ax`) by simply listing them at the front of the line, separated by commas.
2. Second, we plot the geographies as before, but this time we tell the function that we want it to draw the polygons on the axis we are passing, `ax`. This method returns the axis with the geographies in them, so we make sure to store it on an object with the same name, `ax`.
3. On the third line, we effectively remove the box with coordinates.
4. Finally, we draw the entire plot by calling `plt.show()`.

Adding a title

Adding a title is an extra line, if we are creating the plot within a figure, as we just did. To include text on top of the figure:

```python
# Setup figure and axis
f, ax = plt.subplots(1)
# Add layer of polygons on the axis
streets.plot(ax=ax)
# Add figure title
f.suptitle("Streets in Zaragoza")
# Display
plt.show()
```

../../_images/lab_C_45_0.png

Changing the size of the map

The size of the plot is changed equally easily in this context. The only difference is that it is specified when we create the figure with the argument `figsize`. The first number represents the width, the X axis, and the second corresponds with the height, the Y axis.

```python
# Setup figure and axis with different size
f, ax = plt.subplots(1, figsize=(12, 12))
# Add layer of polygons on the axis
cities.plot(ax=ax)
# Display
plt.show()
```

../../_images/lab_C_48_0.png

Modifying borders

Border lines sometimes can distort or impede proper interpretation of a map. In those cases, it is useful to know how they can be modified. Although not too complicated, the way to access borders in `geopandas` is not as straightforward as it is the case for other aspects of the map, such as size or frame. Let us first see the code to make the *lines thicker* and *black*, and then we will work our way through the different steps:

```python
# Setup figure and axis
f, ax = plt.subplots(1, figsize=(12, 12))
# Add layer of polygons on the axis, set fill color (`facecolor`) and boundary
# color (`edgecolor`)
cities.plot(linewidth=1,
            facecolor='red',
            edgecolor='black',
            ax=ax
           )
```

```
<AxesSubplot:>
```

../../_images/lab_C_51_1.png

Note how the lines are thicker. In addition, all the polygons are colored in the same (default) color, light red. However, because the lines are thicker, we can only see the polygon filling for those cities with an area large enough.

Let us examine line by line what we are doing in the code snippet:

- We begin by creating the figure (`f`) object and one axis inside it (`ax`) where we will plot the map.

- Then, we call `plot` as usual, but pass in two new arguments: `linewidth` for the width of the line; `facecolor`, to control the color each polygon is filled with; and `edgecolor`, to control the color of the boundary.

This approach works very similarly with other geometries, such as lines. For example, if we wanted to plot the streets in red, we would simply:

```
# Setup figure and axis
f, ax = plt.subplots(1)
# Add layer with lines, set them red and with different line width
# and append it to the axis `ax`
streets.plot(linewidth=2, color='red', ax=ax)
```

```
<AxesSubplot:>
```

../../_images/lab_C_53_1.png

Important, note that in the case of lines the parameter to control the color is simply `color`. This is because lines do not have an area, so there is no need to distinguish between the main area (`facecolor`) and the border lines (`edgecolor`).

Transforming CRS

The coordindate reference system (CRS) is the way geographers and cartographers have to represent a three-dimentional object, such as the round earth, on a two-dimensional plane, such as a piece of paper or a computer screen. If the source data contain information on the CRS of the data, we can modify this in a `GeoDataFrame`. First let us check if we have the information stored properly:

```
cities.crs
```

```
<Projected CRS: EPSG:25830>
Name: ETRS89 / UTM zone 30N
Axis Info [cartesian]:
- E[east]: Easting (metre)
- N[north]: Northing (metre)
Area of Use:
- name: Europe - 6°W to 0°W and ETRS89 by country
- bounds: (-6.0, 35.26, 0.0, 80.53)
Coordinate Operation:
- name: UTM zone 30N
- method: Transverse Mercator
Datum: European Terrestrial Reference System 1989
- Ellipsoid: GRS 1980
- Prime Meridian: Greenwich
```

As we can see, there is information stored about the reference system: it is using the standard Spanish projection, which is expressed in meters. There are also other less decipherable parameters but we do not need to worry about them right now.

If we want to modify this and "reproject" the polygons into a different CRS, the quickest way is to find the EPSG code online (epsg.io is a good one, although there are others too). For example, if we wanted to transform the dataset into lat/lon coordinates, we would use its EPSG code, 4326:

```
# Reproject (`to_crs`) and plot (`plot`) polygons
cities.to_crs(epsg=4326).plot()
# Set equal axis
lims = plt.axis('equal')
```

../../_images/lab_C_59_0.png

The shape of the polygons is slightly different. Furthermore, note how the *scale* in which they are plotted differs.

Composing multi-layer maps

So far we have considered many aspects of plotting *a single* layer of data. However, in many cases, an effective map will require more than one: for example we might want to display streets on top of the polygons of neighborhoods, and add a few points for specific locations we want to highlight. At the very heart of GIS is the possibility to combine spatial information from different sources by overlaying it on top of each other, and this is fully supported in Python.

For this section, let's select only Zaragoza from the `streets` table and convert it to lat/lon so it's aligned with the streets and POIs layers:

```
zgz = cities.loc[[112], :].to_crs(epsg=4326)
zgz
```

	city_id	n_buildings	geometry
112	ci122	23604	POLYGON ((-0.93057 41.60615, -0.93092 41.60622...

Combining different layers on a single map boils down to adding each of them to the same axis in a sequential way, as if we were literally overlaying one on top of the previous one. For example, let's plot the boundary of Zaragoza and its bars:

```
# Setup figure and axis
f, ax = plt.subplots(1)
# Add a layer with polygon on to axis `ax`
zgz.plot(ax=ax, color="yellow")
# Add a layer with lines on top in axis `ax`
pois.plot(ax=ax, color="green")
```

```
<AxesSubplot:>
```

../../_images/lab_C_65_1.png

Saving maps to figures

Once we have produced a map we are content with, we might want to save it to a file so we can include it into a report, article, website, etc. Exporting maps in Python involves replacing `plt.show` by `plt.savefig` at the end of the code block to specify where and how to save it. For example to save the previous map into a `png` file in the same folder where the notebook is hosted:

```
# Setup figure and axis
f, ax = plt.subplots(1)
# Add a layer with polygon on to axis `ax`
zgz.plot(ax=ax, color="yellow")
# Add a layer with lines on top in axis `ax`
pois.plot(ax=ax, color="green")
# Save figure to a PNG file
plt.savefig('zaragoza_bars.png')
```

![../../_images/lab_C_68_0.png]

If you now check on the folder, you'll find a `png` (image) file with the map.

The command `plt.savefig` contains a large number of options and additional parameters to tweak. Given the size of the figure created is not very large, we can increase this with the argument `dpi`, which stands for "dots per inch" and it's a standard measure of resolution in images. For example, for a high quality image, we could use 500:

```
# Setup figure and axis
f, ax = plt.subplots(1)
# Add a layer with polygon on to axis `ax`
zgz.plot(ax=ax, color="yellow")
# Add a layer with lines on top in axis `ax`
pois.plot(ax=ax, color="green")
# Save figure to a PNG file
plt.savefig('zaragoza_bars.png', dpi=500)
```

![../../_images/lab_C_70_0.png]

Manipulating spatial tables (`GeoDataFrames`)

Once we have an understanding of how to visually display spatial information contained, let us see how it can be combined with the operations learnt in the previous session about manipulating non-spatial tabular data. Essentially, the key is to realize that a `GeoDataFrame` contains most of its spatial information in a single column named `geometry`, but the rest of it looks and behaves exactly like a non-spatial `DataFrame` (in fact, it is). This concedes them all the flexibility and convenience that we saw in manipulating, slicing, and transforming tabular data, with the bonus that spatial data is carried away in all those steps. In addition, `GeoDataFrames` also incorporate a set of explicitly spatial operations to combine and transform data. In this section, we will consider both.

`GeoDataFrame`s come with a whole range of traditional GIS operations built-in. Here we will run through a small subset of them that contains some of the most commonly used ones.

Area calculation

One of the spatial aspects we often need from polygons is their area. "How big is it?" is a question that always haunts us when we think of countries, regions, or cities. To obtain area measurements, first make sure you `GeoDataFrame` is projected. If that is the case, you can calculate areas as follows:

```
city_areas = cities.area
city_areas.head()
```

```
0    8.449666e+06
1    9.121270e+06
2    1.322653e+07
3    6.808121e+07
4    1.072284e+07
dtype: float64
```

This indicates that the area of the first city in our table takes up 8,450,000 squared metres. If we wanted to convert into squared kilometres, we can divide by 1,000,000:

```
areas_in_sqkm = city_areas / 1000000
areas_in_sqkm.head()
```

```
0     8.449666
1     9.121270
2    13.226528
3    68.081212
4    10.722843
dtype: float64
```

Length

Similarly, an equally common question with lines is their length. Also similarly, their computation is relatively straightforward in Python, provided that our data are projected. Here we will perform the projection (`to_crs`) and the calculation of the length at the same time:

```
street_length = streets.to_crs(epsg=25830).length
street_length.head()
```

```
0     37.338828
1    104.510732
2    365.969719
3     97.101436
4     94.002218
dtype: float64
```

Since the CRS we use (`EPSG:25830`) is expressed in metres, we can tell the first street segment is about 37m.

Centroid calculation

Sometimes it is useful to summarize a polygon into a single point and, for that, a good candidate is its centroid (almost like a spatial analogue of the average). The following command will return a `GeoSeries` (a single column with spatial data) with the centroids of a polygon `GeoDataFrame`:

```
cents = cities.centroid
cents.head()
```

```
0    POINT (386147.759 4204605.994)
1    POINT (216296.159 4579397.331)
2    POINT (688901.588 4180201.774)
3    POINT (518262.028 4069898.674)
4    POINT (206940.936 4127361.966)
dtype: geometry
```

Note how `cents` is not an entire table but a single column, or a `GeoSeries` object. This means you can plot it directly, just like a table:

```
cents.plot()
```

```
<AxesSubplot:>
```

../../_images/lab_C_84_1.png

But you don’t need to call a `geometry` column to inspect the spatial objects. In fact, if you do it will return an error because there is not any `geometry` column, the object `cents` itself is the geometry.

Point in polygon (PiP)

Knowing whether a point is inside a polygon is conceptually a straightforward exercise but computationally a tricky task to perform. The way to perform this operation in `GeoPandas` is through the `contains` method, available for each polygon object.

```
poly = cities.loc[112, "geometry"]
pt1 = cents[0]
pt2 = cents[112]
```

And we can perform the checks as follows:

```
poly.contains(pt1)
```

```
False
```

```
poly.contains(pt2)
```

```
True
```

Performing point-in-polygon in this way is instructive and useful for pedagogical reasons, but for cases with many points and polygons, it is not particularly efficient. In these situations, it is much more advisable to perform then as a “spatial join”. If you are interested in these, see the link provided below to learn more about them.

Buffers

Buffers are one of the classical GIS operations in which an area is drawn around a particular geometry, given a specific radious. These are very useful, for instance, in combination with point-in-polygon operations to calculate accessibility, catchment areas, etc.

For this example, we will use the bars table, but will project it to the same CRS as `cities`, so it is expressed in metres:

```
pois_projected = pois.to_crs(cities.crs)
pois_projected.crs
```

```
<Projected CRS: EPSG:25830>
Name: ETRS89 / UTM zone 30N
Axis Info [cartesian]:
- E[east]: Easting (metre)
- N[north]: Northing (metre)
Area of Use:
- name: Europe - 6°W to 0°W and ETRS89 by country
- bounds: (-6.0, 35.26, 0.0, 80.53)
Coordinate Operation:
- name: UTM zone 30N
- method: Transverse Mercator
Datum: European Terrestrial Reference System 1989
- Ellipsoid: GRS 1980
- Prime Meridian: Greenwich
```

To create a buffer using `geopandas`, simply call the `buffer` method, passing in the radious. For example, to draw a 500m. buffer around every bar in Zaragoza:

```
buf = pois_projected.buffer(500)
buf.head()
```

```
603129991    POLYGON ((676071.164 4612191.116, 676068.756 4...
673807647    POLYGON ((675740.197 4612333.309, 675737.789 4...
759470843    POLYGON ((676083.551 4614558.818, 676081.143 4...
765953540    POLYGON ((675653.012 4611997.279, 675650.604 4...
772973291    POLYGON ((675616.277 4614955.710, 675613.870 4...
dtype: geometry
```

And plotting it is equally straighforward:

```
f, ax = plt.subplots(1)
# Plot buffer
buf.plot(ax=ax, linewidth=0)
# Plot named places on top for reference
# [NOTE how we modify the dot size (`markersize`)
# and the color (`color`)]
pois_projected.plot(ax=ax, markersize=1, color='yellow')
```

```
<AxesSubplot:>
```

../../_images/lab_C_97_1.png

Adding base layers from web sources

Many single datasets lack context when displayed on their own. A common approach to alleviate this is to use web tiles, which are a way of quickly obtaining geographical context to present spatial data. In Python, we can use [`contextily`] to pull down tiles and display them along with our own geographic data.

We can begin by creating a map in the same way we would do normally, and then use the `add_basemap` command to, er, add a basemap:

```
ax = cities.plot(color="black")
cx.add_basemap(ax, crs=cities.crs);
```

../../_images/lab_C_99_0.png

Note that we need to be explicit when adding the basemap to state the coordinate reference system (`crs`) our data is expressed in, `contextily` will not be able to pick it up otherwise. Conversely, we could change our data's CRS into Pseudo-Mercator, the native reference system for most web tiles:

```
cities_wm = cities.to_crs(epsg=3857)
ax = cities_wm.plot(color="black")
cx.add_basemap(ax);
```

../../_images/lab_C_101_0.png

Note how the coordinates are different but, if we set it right, either approach aligns tiles and data nicely.

Web tiles can be integrated with other features of maps in a similar way as we have seen above. So, for example, we can change the size of the map, and remove the axis. Let's use Zaragoza for this example:

```
f, ax = plt.subplots(1, figsize=(9, 9))
zgz.plot(alpha=0.25, ax=ax)
cx.add_basemap(ax, crs=zgz.crs)
ax.set_axis_off()
```

../../_images/lab_C_104_0.png

Now, `contextily` offers a lot of options in terms of the sources and providers you can use to create your basemaps. For example, we can use satellite imagery instead:

```
f, ax = plt.subplots(1, figsize=(9, 9))
zgz.plot(alpha=0.25, ax=ax)
cx.add_basemap(ax,
               crs=zgz.crs,
               source=cx.providers.Esri.WorldImagery
              )
ax.set_axis_off()
```

../../_images/lab_C_106_0.png

Have a look at this Twitter thread to get some further ideas on providers:

Dani Arribas-Bel @darribas · Aug 2, 2019
Replying to @darribas
Get world imagery and make a map of a place in two lines:

Dani Arribas-Bel
@darribas

Terrain maps

And consider checking out the documentation website for the package:

https://contextily.readthedocs.io/en/latest/

Interactive maps

Everything we have seen so far relates to static maps. These are useful for publication, to include in reports or to print. However, modern web technologies afford much more flexibility to explore spatial data interactively.

One of the most prominents examples of interactive maps integrated with notebooks is Kepler.GL. Using its Jupyter integration, we can easily convert data (expressed in lon/lat) into interactive maps, and then play further to tweak its appearance. In this context, we will only show how you can take a `GeoDataFrame` into an interactive map in two lines of code:

```
# NOTE: this may not render on the website

# Create a new map
m = KeplerGl()
# Append your data in lon/lat
m.add_data(streets, name="Streets")
# Display
m
```

```
User Guide: https://docs.kepler.gl/docs/keplergl-jupyter
```

```
/opt/conda/lib/python3.7/site-packages/geopandas/geodataframe.py:852: UserWarning:
Geometry column does not contain geometry.
  warnings.warn("Geometry column does not contain geometry.")
```

Further resources

More advanced GIS operations are possible in `geopandas` and, in most cases, they are extensions of the same logic we have used in this document. If you are thinking about taking the next step from here, the following two operations (and the documentation provided) will give you the biggest "bang for the buck":

- Spatial joins

> https://github.com/geopandas/geopandas/blob/master/examples/spatial_joins.ipynb

- Spatial overlays

> https://github.com/geopandas/geopandas/blob/master/examples/overlays.ipynb

Do-It-Yourself

In this session, we will practice your skills in mapping with Python. Fire up a notebook you can edit interactively, and let's do this!

```
import geopandas, osmnx
```

Data preparation

Polygons

For this section, you will have to push yourself out of the comfort zone when it comes to sourcing the data. As nice as it is to be able to pull a dataset directly from the web at the stroke of a url address, most real-world cases are not that straight forward. Instead, you usually have to download a dataset manually and store it locally on your computer before you can get to work.

We are going to use data from the Consumer Data Research Centre (CDRC) about Liverpool, in particular an extract from the Census. You can download a copy of the data at:

> **Important**
>
> You will need a username and password to download the data. Create it for free at:
>
> > https://data.cdrc.ac.uk/user/register

> Liverpool Census'11 Residential data pack download

Once you have the `.zip` file on your computer, right-click and "Extract all". The resulting folder will contain all you need. For the sake of the example, let's assume you place the resulting folder in the same location as the notebook you are using. If that is the case, you can load up a `GeoDataFrame` of Liverpool neighborhoods with:

```
import geopandas
liv =
geopandas.read_file("Census_Residential_Data_Pack_2011_E08000012/data/Census_Residenti
al_Data_Pack_2011/Local_Authority_Districts/E08000012/shapefiles/E08000012.shp")
```

Lines

For a line layer, we are going to use a different bit of `osmnx` functionality that will allow us to extract all the highways

```
bikepaths = osmnx.geocode_to_gdf("Liverpool, UK", )
```

```
bikepaths = osmnx.graph_from_place("Liverpool, UK", network_type="bike")
```

```
len(bikepaths)
```

```
23460
```

Points

For points, we will use an analogue of the POI layer we have used in the Lab: pubs in Liverpool, as recorded by OpenStreetMap.

Let's import `osmnx`:

```
import osmnx
```

And make a similar query to retrieve the table:

```
pubs = osmnx.pois.pois_from_place("Liverpool, UK",
                                   tags={"amenity": "pub"}
                                  )
```

Tasks

Task I: *Tweak your map*

With those three layers, try to complete the following tasks:

- Make a map of the Liverpool neighborhoods that includes the following characteristics:
    - Features a title
    - Does not include axes frame
    - It has a figure size of 10 by 11
    - Polygons are all in color `"#525252"` and 50% transparent
    - Boundary lines ("edges") have a width of 0.3 and are of color `"#B9EBE3"`
    - Includes a basemap with the [Stamen watercolor theme]

> **Note**
>
> Not all of the requirements above are not equally hard to achieve. If you can get some but not all of them, that's also great! The point is you learn something every time you try.

Task II: *Non-spatial manipulations*

For this one we will combine some of the ideas we learnt in the [previous block] with this one.

Focus on the LSOA `liv` layer and use it to do the following:

1. Calculate the area of each neighbourhood
2. Find the five smallest areas in the table. Create a new object (e.g. `smallest` with them only)
3. Create a multi-layer map of Liverpool where the five smallest areas are coloured in red, and the rest appear in black.

Task III: *The gender gap on the streets*

This one is a bit more advanced, so don't despair if you can't get it on your first try. It also relies on the [streets dataset from the "Hands-on" section], so you will need to load it up on your own. Here're the questions for you to answer:

> *Which group accounts for longer total street length in Zaragoza: men or women? By how much*?

The suggestion is that you get to work right away. However, if this task seems too daunting, you can expand the tip below for a bit of help.

Tip

Concepts

This block is all about Geovisualisation and displaying statistical information on maps. We start with an introduction on *what* geovisualisation is; then we follow with the modifiable areal unit problem, a key concept to keep in mind when displaying statistical information spatially; and we wrap up with tips to make awesome choropleths, thematic maps. Each section contains a short clip and a set of slides, plus a few (optional) readings.

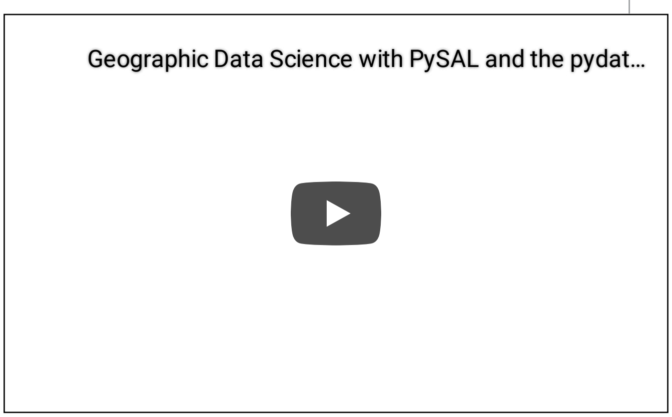
Geovisualisation

Geovisualisation is an area that underpins much what we will discuss in this course. Often, we will be presenting the results of more sophisticated analyses as maps. So getting the principles behind mapping right is critical. In this clip, we cover *what* is (geo)visualisation and why it is important.

Slides

The slides used in the clip are available at:

- [`[HTML]`]
- [`[PDF]`]



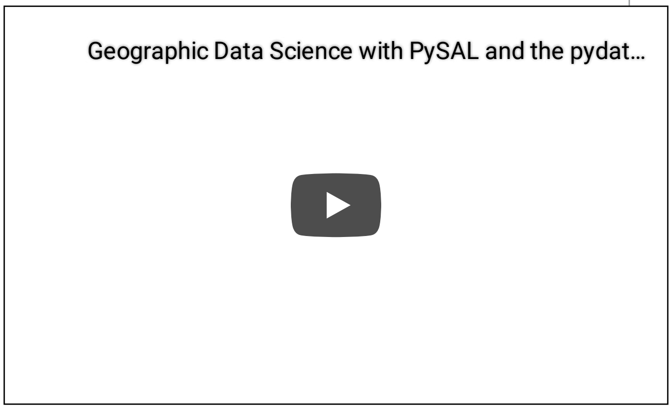
Geographical containers for data

This section tries to get you to think about the geographical containers we use to represent data in maps. By that, we mean the areas, delineations and aggregations we, implicitly or explicitly, incur in when mapping data. This is an important aspect, but Geographers have been aware of them for a long time, so we are standing on the shoulders of giants.

Slides

The slides used in the clip are available at:

- [HTML]
- [PDF]



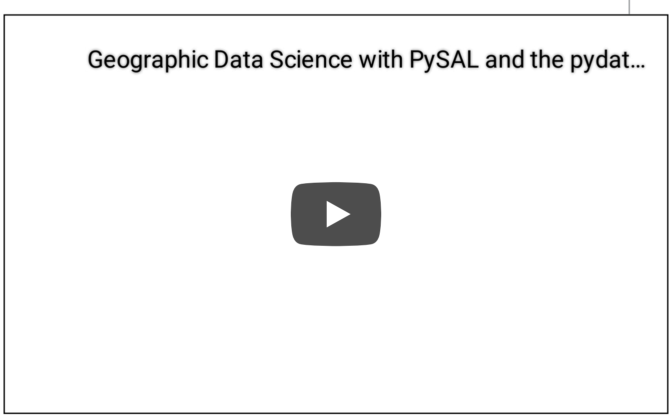
Choropleths

Choropleths are thematic maps and, these days, are everywhere. From elections, to economic inequality, to the distribution of population density, there's a choropleth for everyone. Although technically, it is easy to create choropleths, it is even easier to make *bad* choropleths. Fortunately, there are a few principles that we can follow to create effective choropleths. Get them all delivered right to the conform of your happy place in the following clip and slides!

Slides

The slides used in the clip are available at:

- [HTML]
- [PDF]



Further readings

The clip above contains a compressed version of the key principles behind successful choropleths. For a more comprehensive coverage, please refer to:

- Choropleths chapter on the GDS book (in progress) [RABWng].

The chapter is available for free here

- Cynthia Brewer's "Designing Better Maps" [Bre15] covers several core aspects of building effective geovisualisations.

Hands-on

Choropleths in Python

> **Important**
>
> This is an adapted version, with a bit less content and detail, of the chapter on choropleth mapping by Rey, Arribas-Bel and Wolf (*in progress*) [RABWng]. Check out the full chapter, available for free at:
>
> > https://geographicdata.science/book/notebooks/05_choropleth.html

In this session, we will build on all we have learnt so far about loading and manipulating (spatial) data and apply it to one of the most commonly used forms of spatial analysis: choropleths. Remember these are maps that display the spatial distribution of a variable encoded in a color scheme, also called *palette*. Although there are many ways in which you can convert the values of a variable into a specific color, we will focus in this context only on a handful of them, in particular:

- Unique values
- Equal interval
- Quantiles
- Fisher-Jenks

Before all this mapping fun, let us get the importing of libraries and data loading out of the way:

```
%matplotlib inline

import geopandas
from pysal.lib import examples
import seaborn as sns
import pandas as pd
from pysal.viz import mapclassify
import numpy as np
import matplotlib.pyplot as plt
```

Data

To mirror the original chapter this section is based on, we will use the same dataset: the Mexico GDP per capita dataset, which we can access as a PySAL example dataset.

> **Note**
>
> You can read more about PySAL example datasets here

We can get a short explanation of the dataset through the `explain` method:

```python
examples.explain("mexico")
```

```
mexico
======

Decennial per capita incomes of Mexican states 1940-2000
--------------------------------------------------------

* mexico.csv: attribute data. (n=32, k=13)
* mexico.gal: spatial weights in GAL format.
* mexicojoin.shp: Polygon shapefile. (n=32)

Data used in Rey, S.J. and M.L. Sastre Gutierrez. (2010) "Interregional inequality
dynamics in Mexico." Spatial Economic Analysis, 5: 277-298.
```

Now, to download it from its remote location, we can use `load_example`:

```python
mx = examples.load_example("mexico")
```

This will download the data and place it on your home directory. We can inspect the directory where it is stored:

```python
mx.get_file_list()
```

```
['/opt/conda/lib/python3.7/site-packages/libpysal/examples/mexico/mexicojoin.shp',
 '/opt/conda/lib/python3.7/site-packages/libpysal/examples/mexico/mexico.csv',
 '/opt/conda/lib/python3.7/site-packages/libpysal/examples/mexico/README.md',
 '/opt/conda/lib/python3.7/site-packages/libpysal/examples/mexico/mexicojoin.dbf',
 '/opt/conda/lib/python3.7/site-packages/libpysal/examples/mexico/mexicojoin.prj',
 '/opt/conda/lib/python3.7/site-packages/libpysal/examples/mexico/mexico.gal',
 '/opt/conda/lib/python3.7/site-packages/libpysal/examples/mexico/mexicojoin.shx']
```

For this section, we will read the ESRI shapefile, which we can do by pointing `geopandas.read_file` to the `.shp` file. The utility function `get_path` makes it a bit easier for us:

```python
db = geopandas.read_file(examples.get_path("mexicojoin.shp"))
```

And, from now on, `db` is a table as we are used to so far in this course:

```python
db.info()
```

```
<class 'geopandas.geodataframe.GeoDataFrame'>
RangeIndex: 32 entries, 0 to 31
Data columns (total 35 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   POLY_ID     32 non-null     int64
 1   AREA        32 non-null     float64
 2   CODE        32 non-null     object
 3   NAME        32 non-null     object
 4   PERIMETER   32 non-null     float64
 5   ACRES       32 non-null     float64
 6   HECTARES    32 non-null     float64
 7   PCGDP1940   32 non-null     float64
 8   PCGDP1950   32 non-null     float64
 9   PCGDP1960   32 non-null     float64
 10  PCGDP1970   32 non-null     float64
 11  PCGDP1980   32 non-null     float64
 12  PCGDP1990   32 non-null     float64
 13  PCGDP2000   32 non-null     float64
 14  HANSON03    32 non-null     float64
 15  HANSON98    32 non-null     float64
 16  ESQUIVEL99  32 non-null     float64
 17  INEGI       32 non-null     float64
 18  INEGI2      32 non-null     float64
 19  MAXP        32 non-null     float64
 20  GR4000      32 non-null     float64
 21  GR5000      32 non-null     float64
 22  GR6000      32 non-null     float64
 23  GR7000      32 non-null     float64
 24  GR8000      32 non-null     float64
 25  GR9000      32 non-null     float64
 26  LPCGDP40    32 non-null     float64
 27  LPCGDP50    32 non-null     float64
 28  LPCGDP60    32 non-null     float64
 29  LPCGDP70    32 non-null     float64
 30  LPCGDP80    32 non-null     float64
 31  LPCGDP90    32 non-null     float64
 32  LPCGDP00    32 non-null     float64
 33  TEST        32 non-null     float64
 34  geometry    32 non-null     geometry
dtypes: float64(31), geometry(1), int64(1), object(2)
memory usage: 8.9+ KB
```

The data however does not include a CRS:

```
db.crs
```

To be able to add baselayers, we need to specify one. Looking at the details and the original reference, we find the data are expressed in longitude and latitude, so the CRS we can use is `EPSG:4326`. Let's add it to `db`:

```
db.crs = "EPSG:4326"
db.crs
```

```
<Geographic 2D CRS: EPSG:4326>
Name: WGS 84
Axis Info [ellipsoidal]:
- Lat[north]: Geodetic latitude (degree)
- Lon[east]: Geodetic longitude (degree)
Area of Use:
- name: World
- bounds: (-180.0, -90.0, 180.0, 90.0)
Datum: World Geodetic System 1984
- Ellipsoid: WGS 84
- Prime Meridian: Greenwich
```

Now we are fully ready to map!

Choropleths

Unique values

A choropleth for categorical variables simply assigns a different color to every potential value in the series. The main requirement in this case is then for the color scheme to reflect the fact that different values are not ordered or follow a particular scale.

In Python, creating categorical choropleths is possible with one line of code. To demonstrate this, we can plot the Mexican states and the region each belongs to based on the Mexican Statistics Institute (coded in our table as the `INEGI` variable):

```
db.plot(column="INEGI",
        categorical=True,
        legend=True
       )
```

```
<AxesSubplot:>
```

../../_images/lab_D_21_1.png

Let us stop for a second in a few crucial aspects:

- Note how we are using the same approach as for basic maps, the command `plot`, but we now need to add the argument `column` to specify which column in particular is to be represented.
- Since the variable is categorical we need to make that explicit by setting the argument `categorical` to `True`.
- As an optional argument, we can set `legend` to `True` and the resulting figure will include a legend with the names of all the values in the map.
- Unless we specify a different colormap, the selected one respects the categorical nature of the data by not implying a gradient or scale but a qualitative structure.

Equal interval

If, instead of categorical variables, we want to display the geographical distribution of a continuous phenomenon, we need to select a way to encode each value into a color. One potential solution is applying what is usually called “equal intervals”. The intuition of this method is to split the *range* of the distribution, the difference between the minimum and maximum value, into equally large segments and to assign a different color to each of them according to a palette that reflects the fact that values are ordered.

Creating the choropleth is relatively straightforward in Python. For example, to create an equal interval on the GDP per capita in 2000 (`PCGDP2000`), we can run a similar command as above:

```
db.plot(column="PCGDP2000",
        scheme="equal_interval",
        k=7,
        cmap="YlGn",
        legend=True
       )
```

```
<AxesSubplot:>
```

../../_images/lab_D_25_1.png

Pay attention to the key differences:

- Instead of specifyig `categorical` as `True`, we replace it by the argument `scheme`, which we will use for all choropleths that require a continuous classification scheme. In this case, we set it to `equal_interval`.
- As above, we set the number of colors to 7. Note that we need not pass the bins we calculated above, the plotting method does it itself under the hood for us.
- As optional arguments, we can change the colormap to a yellow to green gradient, which is one of the recommended ones by [ColorBrewer] for a sequential palette.

Now, let's dig a bit deeper into the classification, and how exactly we are encoding values into colors. Each segment, also called bins or buckets, can also be calculated using the library `mapclassify` from the `PySAL` family:

```
classi = mapclassify.EqualInterval(db["PCGDP2000"], k=7)
classi
```

```
EqualInterval

      Interval         Count
----------------------------
[ 8684.00, 15207.57] |    10
(15207.57, 21731.14] |    10
(21731.14, 28254.71] |     5
(28254.71, 34778.29] |     4
(34778.29, 41301.86] |     2
(41301.86, 47825.43] |     0
(47825.43, 54349.00] |     1
```

The only additional argument to pass to `Equal_Interval`, other than the actual variable we would like to classify is the number of segments we want to create, `k`, which we are arbitrarily setting to seven in this case. This will be the number of colors that will be plotted on the map so, although having several can give more detail, at some point the marginal value of an additional one is fairly limited, given the ability of the brain to tell any differences.

Once we have classified the variable, we can check the actual break points where values stop being in one class and become part of the next one:

```
classi.bins
```

```
array([15207.57142857, 21731.14285714, 28254.71428571, 34778.28571429,
       41301.85714286, 47825.42857143, 54349.        ])
```

The array of breaking points above implies that any value in the variable below 15,207.57 will get the first color in the gradient when mapped, values between 15,207.57 and 21,731.14 the next one, and so on.

The key characteristic in equal interval maps is that the bins are allocated based on the magnitude on the values, irrespective of how many obervations fall into each bin as a result of it. In highly skewed distributions, this can result in bins with a large number of observations, while others only have a handful of outliers. This can be seen in the summary table printed out above, where ten states are in the first group, but only one of them belong to the one with highest values. This can also be represented visually with a kernel density plot where the break points are included as well:

../../_images/lab_D_31_0.png

Technically speaking, the figure is created by overlaying a KDE plot with vertical bars for each of the break points. This makes much more explicit the issue highlighed by which the first bin contains a large amount of observations while the one with top values only encompasses a handful of them.

Quantiles

One solution to obtain a more balanced classification scheme is using quantiles. This, by definition, assigns the same amount of values to each bin: the entire series is laid out in order and break points are assigned in a way that leaves exactly the same amount of observations between each of them. This “observation-based” approach contrasts with the “value-based” method of equal intervals and, although it can obscure the magnitude of extreme values, it can be more informative in cases with skewed distributions.

The code required to create the choropleth mirrors that needed above for equal intervals:

```
db.plot(column="PCGDP2000",
        scheme="quantiles",
        k=7,
        cmap="YlGn",
        legend=True
       )
```

```
<AxesSubplot:>
```

../../_images/lab_D_35_1.png

Note how, in this case, the amount of polygons in each color is by definition much more balanced (almost equal in fact, except for rounding differences). This obscures outlier values, which get blurred by significantly smaller values in the same group, but allows to get more detail in the “most populated” part of the distribution, where instead of only white polygons, we can now discern more variability.

To get further insight into the quantile classification, let’s calculate it with `mapclassify`:

```
classi = mapclassify.Quantiles(db["PCGDP2000"], k=7)
classi
```

```
Quantiles

      Interval         Count
----------------------------
[ 8684.00, 11752.00] |     5
(11752.00, 13215.43] |     4
(13215.43, 15996.29] |     5
(15996.29, 20447.14] |     4
(20447.14, 26109.57] |     5
(26109.57, 30357.86] |     4
(30357.86, 54349.00] |     5
```

And, similarly, the bins can also be inspected:

```
classi.bins
```

```
array([11752.        , 13215.42857143, 15996.28571429, 20447.14285714,
       26109.57142857, 30357.85714286, 54349.        ])
```

The visualization of the distribution can be generated in a similar way as well:

../../_images/lab_D_42_0.png

Fisher-Jenks

Equal interval and quantiles are only two examples of very many classification schemes to encode values into colors. Although not all of them are integrated into `geopandas`, `PySAL` includes several other classification schemes (for a detailed list, have a look at this link). As an example of a more sophisticated one, let us create a Fisher-Jenks choropleth:

```
db.plot(column="PCGDP2000",
        scheme="fisher_jenks",
        k=7,
        cmap="YlGn",
        legend=True
       )
```

```
<AxesSubplot:>
```

../../_images/lab_D_45_1.png

The same classification can be obtained with a similar approach as before:

```
classi = mapclassify.FisherJenks(db["PCGDP2000"], k=7)
classi
```

```
FisherJenks

      Interval         Count
----------------------------
[ 8684.00, 13360.00] |    10
(13360.00, 18170.00] |     8
(18170.00, 24068.00] |     4
(24068.00, 28460.00] |     4
(28460.00, 33442.00] |     3
(33442.00, 38672.00] |     2
(38672.00, 54349.00] |     1
```

This methodology aims at minimizing the variance *within* each bin while maximizing that *between* different classes.

```
classi.bins
```

```
array([13360., 18170., 24068., 28460., 33442., 38672., 54349.])
```

Graphically, we can see how the break points are not equally spaced but are adapting to obtain an optimal grouping of observations:

../../_images/lab_D_51_0.png

For example, the bin with highest values covers a much wider span that the one with lowest, because there are fewer states in that value range.

Zooming into the map

Zoom into full map

A general map of an entire region, or urban area, can sometimes obscure local patterns because they happen at a much smaller scale that cannot be perceived in the global view. One way to solve this is by providing a focus of a smaller part of the map in a separate figure. Although there are many ways to do this in Python, the most straightforward one is to reset the limits of the axes to center them in the area of interest.

As an example, let us consider the quantile map produced above:

```
db.plot(column="PCGDP2000",
        scheme="quantiles",
        k=7,
        cmap="YlGn",
        legend=False
       )
```

```
<AxesSubplot:>
```

../../_images/lab_D_56_1.png

If we want to focus around the capital, Mexico DF, the first step involves realising that such area of the map (the little dark green polygon in the SE centre of the map), falls within the coordinates of -102W/-97W, and 18N/21N, roughly speaking. To display a zoom map into that area, we can do as follows:

```
# Setup the figure
f, ax = plt.subplots(1)
# Draw the choropleth
db.plot(column="PCGDP2000",
        scheme="quantiles",
        k=7,
        cmap="YlGn",
        legend=False,
        ax=ax
       )
# Redimensionate X and Y axes to desired bounds
ax.set_ylim(18, 21)
ax.set_xlim(-102, -97)
```

```
(-102.0, -97.0)
```

../../_images/lab_D_58_1.png

Partial map

The approach above is straightforward, but not necessarily the most efficient one: not that, to generate a map of a potentially very small area, we effectively draw the entire (potentially very large) map, and discard everything except the section we want. This is not straightforward to notice at first sight, but what Python is doing in the code cell above is plottin the entire `db` object, and only then focusing the figure on the X and Y ranges specified in `set_xlim`/`set_ylim`.

Sometimes, this is required. For example, if we want to retain the same coloring used for the national map, but focus on the region around Mexico DF, this approach is the easiest one.

However, sometimes, we only need to plot the *geographical features* within a given range, and we either don't need to keep the national coloring (e.g. we are using a single color), or we want a classification performed *only* with the features in the region.

For these cases, it is computationally more efficient to select the data we want to plot first, and then display them through `plot`. For this, we can rely on the `cx` operator:

```
subset = db.cx[-102:-97, 18:21]
subset.plot()
```

```
<AxesSubplot:>
```

../../_images/lab_D_60_1.png

We query the range of spatial coordinates similarly to how we query indices with `loc`. Note however the result includes full geographic features, and hence the polygons with at least some area within the range are included fully. This results in a larger range than originally specified.

This approach is a "spatial slice". If you remember when we saw non-spatial slices (enabled by the `loc` operator), this is a similar approach but our selection criteria, instead of subsetting by indices of the table, are based on the spatial coordinates of the data represented in the table.

Since the result is a `GeoDataFrame` itself, we can create a choropleth that is based only on the data in the subset:

```
subset.plot(column="PCGDP2000",
            scheme="quantiles",
            k=7,
            cmap="YlGn",
            legend=False
           )
```

```
<AxesSubplot:>
```

../../_images/lab_D_62_1.png

Do-It-Yourself

Let's make a bunch of choropleths! In this section, you will practice the [concepts] and [code] we have learnt in this block. Happy hacking!

Data preparation

> **Note**
>
> The AHAH dataset was invented by a University of Liverpool team. If you want to find out more about the background and details of the project, have a look at the information page at the CDRC website.

We are going to use the Access to Healthy Assets and Hazards (AHAH) index. This is a score that ranks LSOAs (the same polygons we used in block C) by the proximity to features of the environment that are considered positive for health (assets) and negative (hazards). The resulting number gives us a sense of how "unhealthy" the environment of the LSOA is. The higher the score, the less healthy the area is assessed to be.

To download the Liverpool AHAH pack, please go over to:

> **Important**
>
> You will need a username and password to download the data. Create it for free at:
>
> https://data.cdrc.ac.uk/user/register

Liverpool AHAH GeoData pack

Once you have the `.zip` file on your computer, right-click and "Extract all". The resulting folder will contain all you need. For the sake of the example, let's assume you place the resulting folder in the same location as the notebook you are using. If that is the case, you can load up a `GeoDataFrame` of Liverpool neighborhoods with:

```
import geopandas
lsoas =
geopandas.read_file("Access_to_Healthy_Assets_and_Hazards_AHAH_E08000012/data/Access_to_Healthy_Assets_and_Hazards_AHAH/Local_Authority_Districts/E08000012/shapefiles/E08000012.shp")
```

Now, this gets us the geometries of the LSOAs, but not the AHAH data. For that, we need to read in the data and join it to `ahah`. Assuming the same location of the data as above, we can do as follows:

```
import pandas
ahah_data =
pandas.read_csv("Access_to_Healthy_Assets_and_Hazards_AHAH_E08000012/data/Access_to_Healthy_Assets_and_Hazards_AHAH/Local_Authority_Districts/E0800001/tables/E08000012.csv")
```

To read the data, and as follows to join it:

```
ahah = lsoas.join(ahah_data.set_index("lsoa11cd"), on="lsoa11cd")
```

Now we're ready to map using the `ahah` object.

Tasks

Task I: AHAH choropleths

Create the following choropleths and, where possible, complement them with a figure that displays the distribution of values using a KDE:

- Equal Interval with five classes
- Quantiles with five classes
- Fisher-Jenks with five classes
- Unique Values with the following setup:
  - Split the LSOAs in two classes: above and below the average AHAH score
  - Assign a qualitative label (`above` or `below`) to each LSOA
  - Create a unique value map for the labels you have just created

Task II: Zoom maps

Generate the following maps:

- Zoom of the [city centre of Liverpool] with he same color for every LSO
- Quantile map of AHAH for all of Liverpool, zoomed into [north of the city centre]
- Zoom to [north of the city centre] with a quantile map of AHAH for the section only

Concepts

This block is about how we pull off the trick to turn geography into numbers statistics can understand. At this point, we dive right into the more methods part of the course; so you can expect a bit of a ramp up in the conceptual sections. Take a deep breath and jump in, it's well worth the effort! At the same time, the coding side of each block will start looking more and more familiar because we are starting to repeat concepts and we will introduce less *new* building blocks and instead rely more and more on what we have seen, just adding small bits here and there.

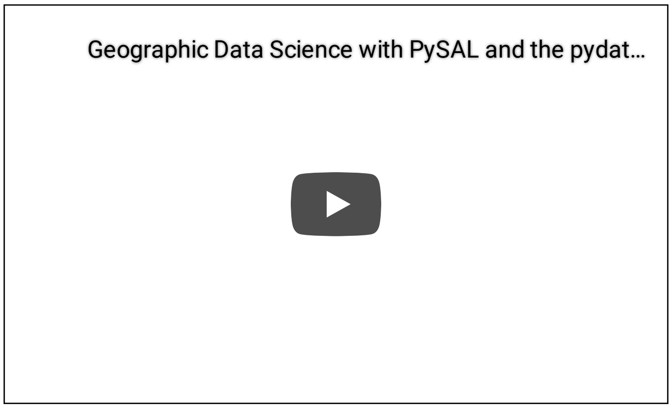
Space, formally

How do you express geographical relations between objects (e.g. areas, points) in a way that can be used in statistical analysis? This is exactly the core of what we get into in here. There are several ways, of course, but one of the most widespread approaches is what is termed spatial weights matrices. We motivate their role and define them in the following clip.

Slides

The slides used in the clip are available at:

- `[HTML]`
- `[PDF]`



Once you have watched the clip above, here's a quiz for you!

Imagine a geography of squared regions (ie. a grid) with the following structure:

regions

Each region is assigned an ID; so the most north-west region is 1, while the most south-east is 9. Here's a question:

> *What is the dimension of the Spatial Weights Matrix for the region above?*

Tip

Solution

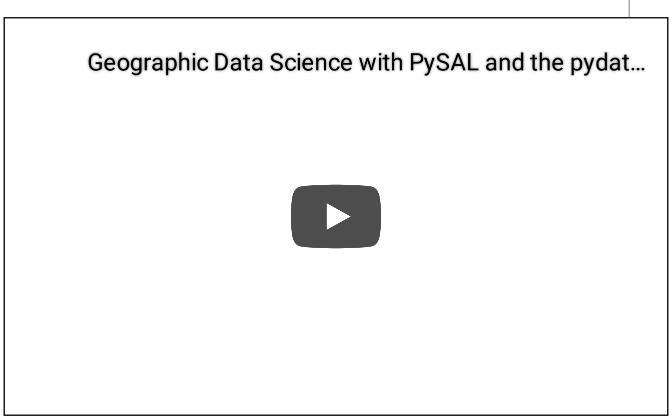
Types of Weights

Once we know what spatial weights are generally, in this clip we dive into some of the specific types we can build for our analyses.

Slides

The slides used in the clip are available at:

- `[HTML]`
- `[PDF]`



Here is a second question for you once you have watched the clip above:

> *What does the rook contiguity spatial weights matrix look like for the region abvoe? Can you write it down by hand?*

Solution

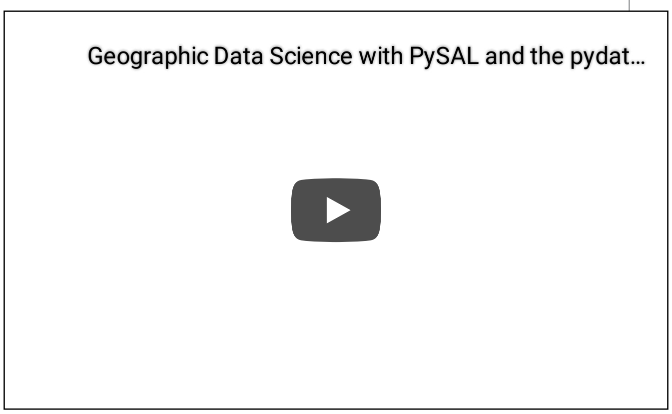
The Spatial Lag

We wrap up the the set of concepts in this block with one of the applications that makes spatial weights matrices so important: the spatial lag. Watch the clip to find out what it is and then jump over the [next part](#) to see how all of these ideas translate into delicious, juicy Python code!

Slides

The slides used in the clip are available at:

- `[HTML]`
- `[PDF]`



More materials

If you want a similar but slightly different take on spatial weights by Luc Anselin, one of the biggest minds in the field of spatial analysis, I strongly recommend you watch the following two clips, part of the course offered by the Spatial Data Center at the University of Chicago:

- Lecture on “Spatial Weights”

- Lecture on “Spatial Lag”, you can ignore the last five minutes as they are a bit more advanced

Further readings

If you liked what you saw in this section and would like to digg deeper into spatial weights, the following readings are good next steps:

- Spatial weights chapter on the GDS book (in progress) [RABWng].
- For a more advanced and detailed treatment, the chapters on spatial weights in the Anselin & Rey book [AR14] are the best source.

The chapter is available for free here

Hands-on

Spatial weights

In this session we will be learning the ins and outs of one of the key pieces in spatial analysis: spatial weights matrices. These are structured sets of numbers that formalize geographical relationships between the observations in a dataset. Essentially, a spatial weights matrix of a given geography is a positive definite matrix of dimensions N by N, where N is the total number of observations:

$$W = \begin{pmatrix} 0 & w_{12} & \dots & w_{1N} \\ w_{21} & \ddots & w_{ij} & \vdots \\ \vdots & w_{ji} & 0 & \vdots \\ w_{N1} & \dots & \dots & 0 \end{pmatrix}$$

where each cell w_{ij} contains a value that represents the degree of spatial contact or interaction between observations i and j. A fundamental concept in this context is that of *neighbor* and *neighborhood*. By convention, elements in the diagonal (w_{ij}) are set to zero. A *neighbor* of a given observation i is another observation with which i has some degree of connection. In terms of W, i and j are thus neighbors if $w_{ij} > 0$. Following this logic, the neighborhood of i will be the set of observations in the system with which it has certain connection, or those observations with a weight greater than zero.

There are several ways to create such matrices, and many more to transform them so they contain an accurate representation that aligns with the way we understand spatial interactions between the elements of a system. In this session, we will introduce the most commonly used ones and will show how to compute them with `PySAL`.

```
%matplotlib inline

import seaborn as sns
import pandas as pd
from pysal.lib import weights
from libpysal.io import open as psopen
import geopandas as gpd
import numpy as np
import matplotlib.pyplot as plt
```

Data

For this tutorial, we will use a dataset of Liverpool small areas (or Lower layer Super Output Areas, LSOAs) for Liverpool. The table is available as part of this course, so can be accessed remotely through the web. If you want to see how the table was created, a notebook is available here.

To make things easier, we will read data from a file posted online so, for now, you do not need to download any dataset:

```
# Read the file in
db = gpd.read_file("https://darribas.org/gds_course/content/data/liv_lsoas.gpkg")
# Index table on the LSOA ID
db = db.set_index("LSOA11CD", drop=False)
# Display summary
db.info()
```

> **Important**
>
> Make sure you are connected to the internet when you run this cell

```
<class 'geopandas.geodataframe.GeoDataFrame'>
Index: 298 entries, E01006512 to E01033768
Data columns (total 3 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   LSOA11CD  298 non-null    object
 1   MSOA11CD  298 non-null    object
 2   geometry  298 non-null    geometry
dtypes: geometry(1), object(2)
memory usage: 9.3+ KB
```

> **Alternative**
>
> Instead of reading the file directly off the web, it is possible to download it manually, store it on your computer, and read it locally. To do that, you can follow these steps:
>
> 1. Download the file by right-clicking on this link and saving the file
> 2. Place the file on the *same folder as the notebook* where you intend to read it
> 3. Replace the code in the cell above by:
>
> ```
> db = gpd.read_file("liv_lsoas.gpkg")
> ```

Building spatial weights in `PySAL`

Contiguity

Contiguity weights matrices define spatial connections through the existence of common boundaries. This makes it directly suitable to use with polygons: if two polygons share boundaries to some degree, they will be labeled as neighbors under these kinds of weights. Exactly how much they need to share is what differenciates the two approaches we will learn: queen and rook.

- **Queen**

Under the queen criteria, two observations only need to share a vortex (a single point) of their boundaries to be considered neighbors. Constructing a weights matrix under these principles can be done by running:

```
w_queen = weights.Queen.from_dataframe(db, idVariable="LSOA11CD")
w_queen
```

```
<libpysal.weights.contiguity.Queen at 0x7f4a2db6e650>
```

The command above creates an object `w_queen` of the class `W`. This is the format in which spatial weights matrices are stored in `PySAL`. By default, the weights builder (`Queen.from_dataframe`) will use the index of the table, which is useful so we can keep everything in line easily.

A `W` object can be queried to find out about the contiguity relations it contains. For example, if we would like to know who is a neighbor of observation `E01006690`:

```
w_queen['E01006690']
```

```
{'E01006759': 1.0,
 'E01006695': 1.0,
 'E01033763': 1.0,
 'E01006697': 1.0,
 'E01006692': 1.0,
 'E01006691': 1.0,
 'E01006720': 1.0}
```

This returns a Python dictionary that contains the ID codes of each neighbor as keys, and the weights they are assigned as values. Since we are looking at a raw queen contiguity matrix, every neighbor gets a weight of one. If we want to access the weight of a specific neighbor, `E01006691` for example, we can do recursive querying:

```
w_queen['E01006690']['E01006691']
```

```
1.0
```

`W` objects also have a direct way to provide a list of all the neighbors or their weights for a given observation. This is thanks to the `neighbors` and `weights` attributes:

```
w_queen.neighbors['E01006690']
```

```
['E01006759',
 'E01006695',
 'E01033763',
 'E01006697',
 'E01006692',
 'E01006691',
 'E01006720']
```

```
w_queen.weights['E01006690']
```

```
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
```

Once created, `W` objects can provide much information about the matrix, beyond the basic attributes one would expect. We have direct access to the number of neighbors each observation has via the attribute `cardinalities`. For example, to find out how many neighbors observation `E01006524` has:

```
w_queen.cardinalities['E01006524']
```

```
6
```

Since `cardinalities` is a dictionary, it is direct to convert it into a `Series` object:

```
queen_card = pd.Series(w_queen.cardinalities)
queen_card.head()
```

```
E01006512    6
E01006513    9
E01006514    5
E01006515    8
E01006518    5
dtype: int64
```

This allows, for example, to access quick plotting, which comes in very handy to get an overview of the size of neighborhoods in general:

```
sns.distplot(queen_card, bins=10)
```

```
<AxesSubplot:>
```

../../_images/lab_E_23_1.png

The figure above shows how most observations have around five neighbors, but there is some variation around it. The distribution also seems to follow a symmetric form, where deviations from the average occur both in higher and lower values almost evenly.

Some additional information about the spatial relationships contained in the matrix are also easily available from a `W` object. Let us tour over some of them:

```
# Number of observations
w_queen.n
```

```
298
```

```
# Average number of neighbors
w_queen.mean_neighbors
```

```
5.617449664429531
```

```
# Min number of neighbors
w_queen.min_neighbors
```

```
1
```

```
# Max number of neighbors
w_queen.max_neighbors
```

```
11
```

```
# Islands (observations disconnected)
w_queen.islands
```

```
[]
```

```
# Order of IDs (first five only in this case)
w_queen.id_order[:5]
```

```
['E01006512', 'E01006513', 'E01006514', 'E01006515', 'E01006518']
```

Spatial weight matrices can be explored visually in other ways. For example, we can pick an observation and visualize it in the context of its neighborhood. The following plot does exactly that by zooming into the surroundings of LSOA `E01006690` and displaying its polygon as well as those of its neighbors:

```
# Setup figure
f, ax = plt.subplots(1, figsize=(6, 6))
# Plot base layer of polygons
db.plot(ax=ax, facecolor='k', linewidth=0.1)
# Select focal polygon
# NOTE we pass both the area code and the column name
#      (`geometry`) within brackets!!!
focus = db.loc[['E01006690'], ['geometry']]
# Plot focal polygon
focus.plot(facecolor='red', alpha=1, linewidth=0, ax=ax)
# Plot neighbors
neis = db.loc[w_queen['E01006690'], :]
neis.plot(ax=ax, facecolor='lime', linewidth=0)
# Title
f.suptitle("Queen neighbors of `E01006690`")
# Style and display on screen
ax.set_ylim(388000, 393500)
ax.set_xlim(336000, 339500)
plt.show()
```

../../_images/lab_E_32_0.png

Note how the figure is built gradually, from the base map (L. 4-5), to the focal point (L. 9), to its neighborhood (L. 11-13). Once the entire figure is plotted, we zoom into the area of interest (L. 19-20).

- **Rook**

Rook contiguity is similar to and, in many ways, superseded by queen contiguity. However, since it sometimes comes up in the literature, it is useful to know about it. The main idea is the same: two observations are neighbors if they share some of their boundary lines. However, in the rook case, it is not enough with sharing only one point, it needs to be at least a segment of their boundary. In most applied cases, these differences usually boil down to how the geocoding was done, but in some cases, such as when we use raster data or grids, this approach can differ more substantively and it thus makes more sense.

From a technical point of view, constructing a rook matrix is very similar:

```
w_rook = weights.Rook.from_dataframe(db)
w_rook
```

```
<libpysal.weights.contiguity.Rook at 0x7f4a2d499790>
```

The output is of the same type as before, a `W` object that can be queried and used in very much the same way as any other one.

Distance

Distance based matrices assign the weight to each pair of observations as a function of how far from each other they are. How this is translated into an actual weight varies across types and variants, but they all share that the ultimate reason why two observations are assigned some weight is due to the distance between them.

- **K-Nearest Neighbors**

One approach to define weights is to take the distances between a given observation and the rest of the set, rank them, and consider as neighbors the k closest ones. That is exactly what the k-nearest neighbors (KNN) criterium does.

To calculate KNN weights, we can use a similar function as before and derive them from a shapefile:

```
knn5 = weights.KNN.from_dataframe(db, k=5)
knn5
```

```
<libpysal.weights.distance.KNN at 0x7f4a2d1c3c90>
```

Note how we need to specify the number of nearest neighbors we want to consider with the argument `k`. Since it is a polygon shapefile that we are passing, the function will automatically compute the centroids to derive distances between observations. Alternatively, we can provide the points in the form of an array, skipping this way the dependency of a file on disk:

```
# Extract centroids
cents = db.centroid
# Extract coordinates into an array
pts = pd.DataFrame({"X": cents.x,
                    "Y": cents.y
                   }).values
# Compute KNN weights
knn5_from_pts = weights.KNN.from_array(pts, k=5)
knn5_from_pts
```

```
<libpysal.weights.distance.KNN at 0x7f4a2d1980d0>
```

- **Distance band**

Another approach to build distance-based spatial weights matrices is to draw a circle of certain radious and consider neighbor every observation that falls within the circle. The technique has two main variations: binary and continuous. In the former one, every neighbor is given a weight of one, while in the second one, the weights can be further tweaked by the distance to the observation of interest.

To compute binary distance matrices in `PySAL`, we can use the following command:

```
w_dist1kmB = weights.DistanceBand.from_dataframe(db, 1000)
```

```
/opt/conda/lib/python3.7/site-packages/libpysal/weights/weights.py:172:
UserWarning: The weights matrix is not fully connected:
 There are 2 disconnected components.
  warnings.warn(message)
```

This creates a binary matrix that considers neighbors of an observation every polygon whose centroid is closer than 1,000 metres (1Km) of the centroid of such observation. Check, for example, the neighborhood of polygon `E01006690`:

```
w_dist1kmB['E01006690']
```

```
{'E01006691': 1.0,
 'E01006692': 1.0,
 'E01006695': 1.0,
 'E01006697': 1.0,
 'E01006720': 1.0,
 'E01006725': 1.0,
 'E01006726': 1.0,
 'E01033763': 1.0}
```

Note that the units in which you specify the distance directly depend on the CRS in which the spatial data are projected, and this has nothing to do with the weights building but it can affect it significantly. Recall how you can check the CRS of a `GeoDataFrame`:

```
db.crs
```

```
<Projected CRS: PROJCS["Transverse_Mercator",GEOGCS["GCS_OSGB 1936 ...>
Name: Transverse_Mercator
Axis Info [cartesian]:
- [east]: Easting (metre)
- [north]: Northing (metre)
Area of Use:
- undefined
Coordinate Operation:
- name: unnamed
- method: Transverse Mercator
Datum: OSGB 1936
- Ellipsoid: Airy 1830
- Prime Meridian: Greenwich
```

In this case, you can see the unit is expressed in metres (`m`), hence we set the threshold to 1,000 for a circle of 1km of radious.

An extension of the weights above is to introduce further detail by assigning different weights to different neighbors within the radious circle based on how far they are from the observation of interest. For example, we could think of assigning the inverse of the distance between observations i and j as w_{ij}. This can be computed with the following command:

```
w_dist1kmC = weights.DistanceBand.from_dataframe(db, 1000, binary=False)
```

```
/opt/conda/lib/python3.7/site-packages/scipy/sparse/data.py:117: RuntimeWarning:
divide by zero encountered in reciprocal
  return self._with_data(data ** n)
```

In `w_dist1kmC`, every observation within the 1km circle is assigned a weight equal to the inverse distance between the two:

$$w_{ij} = \frac{1}{d_{ij}}$$

This way, the further apart i and j are from each other, the smaller the weight w_{ij} will be.

Contrast the binary neighborhood with the continuous one for `E01006690`:

```
w_dist1kmC['E01006690']
```

```
{'E01006691': 0.001320115452290246,
 'E01006692': 0.0016898106255168294,
 'E01006695': 0.001120923796462639,
 'E01006697': 0.001403469553911711,
 'E01006720': 0.0013390451319917913,
 'E01006725': 0.001009044334260805,
 'E01006726': 0.0010528395831202145,
 'E01033763': 0.0012983249272553688}
```

Following this logic of more detailed weights through distance, there is a temptation to take it further and consider everyone else in the dataset as a neighbor whose weight will then get modulated by the distance effect shown above. However, although conceptually correct, this approach is not always the most computationally or practical one. Because of the nature of spatial weights matrices, particularly because of the fact their size is N by N, they can grow substantially large. A way to cope with this problem is by making sure they remain fairly *sparse* (with many zeros). Sparsity is typically ensured in the case of contiguity or KNN by construction but, with inverse distance, it needs to be imposed as, otherwise, the matrix could be potentially entirely dense (no zero values other than the diagonal). In practical terms, what is usually done is to impose a distance threshold beyond which no weight is assigned and interaction is assumed to be non-existent. Beyond being computationally feasible and scalable, results from this approach usually do not differ much from a fully “dense” one as the additional information that is included from further observations is almost ignored due to the small weight they receive. In this context, a commonly used threshold, although not always best, is that which makes every observation to have at least one neighbor.

Such a threshold can be calculated as follows:

```
min_thr = weights.min_threshold_distance(pts)
min_thr
```

```
939.7373992113434
```

Which can then be used to calculate an inverse distance weights matrix:

```
w_min_dist = weights.DistanceBand.from_dataframe(db, min_thr, binary=False)
```

Block weights

Block weights connect every observation in a dataset that belongs to the same category in a list provided ex-ante. Usually, this list will have some relation to geography an the location of the observations but, technically speaking, all one needs to create block weights is a list of memberships. In this class of weights, neighboring observations, those in the same group, are assigned a weight of one, and the rest receive a weight of zero.

In this example, we will build a spatial weights matrix that connects every LSOA with all the other ones in the same MSOA. See how the MSOA code is expressed for every LSOA:

```
db.head()
```

	LSOA11CD	MSOA11CD	geometry
LSOA11CD			
E01006512	E01006512	E02001377	POLYGON ((336103.358 389628.580, 336103.416 38...
E01006513	E01006513	E02006932	POLYGON ((335173.781 389691.538, 335169.798 38...
E01006514	E01006514	E02001383	POLYGON ((335495.676 389697.267, 335495.444 38...
E01006515	E01006515	E02001383	POLYGON ((334953.001 389029.000, 334951.000 38...
E01006518	E01006518	E02001390	POLYGON ((335354.015 388601.947, 335354.000 38...

To build a block spatial weights matrix that connects as neighbors all the LSOAs in the same MSOA, we only require the mapping of codes. Using `PySAL`, this is a one-line task:

```
w_block = weights.block_weights(db['MSOA11CD'])
```

```
/opt/conda/lib/python3.7/site-packages/libpysal/weights/weights.py:172:
UserWarning: The weights matrix is not fully connected:
 There are 61 disconnected components.
  warnings.warn(message)
```

In this case, `PySAL` does not allow to pass the argument `idVariable` as above. As a result, observations are named after the the order the occupy in the list:

```
w_block[0]
```

```
{218: 1.0, 219: 1.0, 220: 1.0, 292: 1.0}
```

The first element is neighbor of observations 218, 129, 220, and 292, all of them with an assigned weight of 1. However, it is possible to correct this by using the additional method `remap_ids`:

```
w_block.remap_ids(db.index)
```

Now if you try `w_bloc[0]`, it will return an error. But if you query for the neighbors of an observation by its LSOA id, it will work:

```
w_block['E01006512']
```

```
{'E01006747': 1.0, 'E01006748': 1.0, 'E01006751': 1.0, 'E01033763': 1.0}
```

Standardizing `W` matrices

In the context of many spatial analysis techniques, a spatial weights matrix with raw values (e.g. ones and zeros for the binary case) is not always the best suiting one for analysis and some sort of transformation is required. This implies modifying each weight so they conform to certain rules. `PySAL` has transformations baked right into the `W` object, so it is possible to check the state of an object as well as to modify it.

Consider the original queen weights, for observation `E01006690`:

```
w_queen['E01006690']
```

```
{'E01006759': 1.0,
 'E01006695': 1.0,
 'E01033763': 1.0,
 'E01006697': 1.0,
 'E01006692': 1.0,
 'E01006691': 1.0,
 'E01006720': 1.0}
```

Since it is contiguity, every neighbor gets one, the rest zero weight. We can check if the object `w_queen` has been transformed or not by calling the argument `transform`:

```
w_queen.transform
```

```
'O'
```

where `O` stands for "original", so no transformations have been applied yet. If we want to apply a row-based transformation, so every row of the matrix sums up to one, we modify the `transform` attribute as follows:

```
w_queen.transform = 'R'
```

Now we can check the weights of the same observation as above and find they have been modified:

```
w_queen['E01006690']
```

```
{'E01006759': 0.14285714285714285,
 'E01006695': 0.14285714285714285,
 'E01033763': 0.14285714285714285,
 'E01006697': 0.14285714285714285,
 'E01006692': 0.14285714285714285,
 'E01006691': 0.14285714285714285,
 'E01006720': 0.14285714285714285}
```

Save for precission issues, the sum of weights for all the neighbors is one:

```
pd.Series(w_queen['E01006690']).sum()
```

```
0.9999999999999998
```

Returning the object back to its original state involves assigning `transform` back to original:

```
w_queen.transform = 'O'
```

```python
w_queen['E01006690']
```

```
{'E01006759': 1.0,
 'E01006695': 1.0,
 'E01033763': 1.0,
 'E01006697': 1.0,
 'E01006692': 1.0,
 'E01006691': 1.0,
 'E01006720': 1.0}
```

`PySAL` supports the following transformations:

- `O`: original, returning the object to the initial state.
- `B`: binary, with every neighbor having assigned a weight of one.
- `R`: row, with all the neighbors of a given observation adding up to one.
- `V`: variance stabilizing, with the sum of all the weights being constrained to the number of observations.

Reading and Writing spatial weights in `PySAL`

Sometimes, if a dataset is very detailed or large, it can be costly to build the spatial weights matrix of a given geography and, despite the optimizations in the `PySAL` code, the computation time can quickly grow out of hand. In these contexts, it is useful to not have to re-build a matrix from scratch every time we need to re-run the analysis. A useful solution in this case is to build the matrix once, and save it to a file where it can be reloaded at a later stage if needed.

`PySAL` has a common way to write any kind of `W` object into a file using the command `open`. The only element we need to decide for ourselves beforehand is the format of the file. Although there are several formats in which spatial weight matrices can be stored (have a look at the list of supported ones by `PySAL`), we will focused on the two most commonly used ones:

- **`.gal`** files for contiguity weights

Contiguity spatial weights can be saved into a `.gal` file with the following commands:

```python
# Open file to write into
fo = psopen('imd_queen.gal', 'w')
# Write the matrix into the file
fo.write(w_queen)
# Close the file
fo.close()
```

The process is composed by the following three steps:

1. Open a target file for `w`riting the matrix, hence the `w` argument. In this case, if a file `imd_queen.gal` already exists, it will be overwritten, so be careful.
2. Write the `W` object into the file.
3. Close the file. This is important as some additional information is written into the file at this stage, so failing to close the file might have unintended consequences.

Once we have the file written, it is possible to read it back into memory with the following command:

```
w_queen2 = psopen('imd_queen.gal', 'r').read()
w_queen2
```

```
<libpysal.weights.weights.W at 0x7f4a2a8aa2d0>
```

Note how we now use `r` instead of `w` because we are `r`eading the file, and also notice how we open the file and, in the same line, we call `read()` directly.

- **`.gwt`** files for distance-based weights.

A very similar process to the one above can be used to read and write distance based weights. The only difference is specifying the right file format, `.gwt` in this case. So, if we want to write `w_dist1km` into a file, we will run:

```
# Open file
fo = psopen('imd_dist1km.gwt', 'w')
# Write matrix into the file
fo.write(w_dist1kmC)
# Close file
fo.close()
```

And if we want to read the file back in, all we need to do is:

```
w_dist1km2 = psopen('imd_dist1km.gwt', 'r').read()
```

```
/opt/conda/lib/python3.7/site-packages/libpysal/io/iohandlers/gwt.py:148:
RuntimeWarning: DBF relating to GWT was not found, proceeding with unordered string
ids.
  warn("DBF relating to GWT was not found, proceeding with unordered string ids.",
RuntimeWarning)
/opt/conda/lib/python3.7/site-packages/libpysal/weights/weights.py:172: UserWarning:
The weights matrix is not fully connected:
 There are 2 disconnected components.
  warnings.warn(message)
```

Note how, in this case, you will probably receive a warning alerting you that there was not a `DBF` relating to the file. This is because, by default, `PySAL` takes the order of the observations in a `.gwt` from a shapefile. If this is not provided, `PySAL` cannot entirely determine all the elements and hence the resulting `W` might not be complete (islands, for example, can be missing). To fully complete the reading of the file, we can remap the ids as we have seen above:

```
w_dist1km2.remap_ids(db.index)
```

Spatial Lag

One of the most direct applications of spatial weight matrices is the so-called *spatial lag*. The spatial lag of a given variable is the product of a spatial weight matrix and the variable itself:

$$Y_{sl} = WY$$

where Y is a Nx1 vector with the values of the variable. Recall that the product of a matrix and a vector equals the sum of a row by column element multiplication for the resulting value of a given row. In terms of the spatial lag:

$$y_{sl-i} = \sum_j w_{ij} y_j$$

If we are using row-standardized weights, w_{ij} becomes a proportion between zero and one, and y_{sl-i} can be seen as the average value of Y in the neighborhood of i.

For this illustration, we will use the area of each polygon as the variable of interest. And to make things a bit nicer later on, we will keep the log of the area instead of the raw measurement. Hence, let's create a column for it:

```
db["area"] = np.log(db.area)
```

The spatial lag is a key element of many spatial analysis techniques, as we will see later on and, as such, it is fully supported in `PySAL`. To compute the spatial lag of a given variable, `imd_score` for example:

```
# Row-standardize the queen matrix
w_queen.transform = 'R'
# Compute spatial lag of `imd_score`
w_queen_score = weights.lag_spatial(w_queen, db["area"])
# Print the first five elements
w_queen_score[:5]
```

```
array([12.40660189, 12.54225296, 12.28284814, 12.61675295, 12.55042815])
```

Line 4 contains the actual computation, which is highly optimized in `PySAL`. Note that, despite passing in a `pd.Series` object, the output is a `numpy` array. This however, can be added directly to the table `imd`:

```
db['w_area'] = w_queen_score
```

Moran Plot

The Moran Plot is a graphical way to start exploring the concept of spatial autocorrelation, and it is a good application of spatial weight matrices and the spatial lag. In essence, it is a standard scatter plot in which a given variable (`area`, for example) is plotted against *its own* spatial lag. Usually, a fitted line is added to include more information:

```
# Setup the figure and axis
f, ax = plt.subplots(1, figsize=(9, 9))
# Plot values
sns.regplot(x="area", y="w_area", data=db, ci=None)
# Display
plt.show()
```

../../_images/lab_E_100_0.png

In order to easily compare different scatter plots and spot outlier observations, it is common practice to standardize the values of the variable before computing its spatial lag and plotting it. This can be accomplished by substracting the average value and dividing the result by the standard deviation:

$$z_i = \frac{y - \bar{y}}{\sigma_y}$$

where z_i is the standardized version of y_i, $\bar{y}$ is the average of the variable, and σ its standard deviation.

Creating a standardized Moran Plot implies that average values are centered in the plot (as they are zero when standardized) and dispersion is expressed in standard deviations, with the rule of thumb of values greater or smaller than two standard deviations being *outliers*. A standardized Moran Plot also partitions the space into four quadrants that represent different situations:

1. High-High (*HH*): values above average surrounded by values above average.
2. Low-Low (*LL*): values below average surrounded by values below average.
3. High-Low (*HL*): values above average surrounded by values below average.
4. Low-High (*LH*): values below average surrounded by values above average.

These will be further explored once spatial autocorrelation has been properly introduced in subsequent blocks.

```
# Standardize the IMD scores
std_db = (db['area'] - db['area'].mean()) / db['area'].std()
# Compute the spatial lag of the standardized version and save is as a
# Series indexed as the original variable
std_w_db = pd.Series(weights.lag_spatial(w_queen, std_db), index=std_db.index)
# Setup the figure and axis
f, ax = plt.subplots(1, figsize=(9, 9))
# Plot values
sns.regplot(x=std_db, y=std_w_db, ci=None)
# Add vertical and horizontal lines
plt.axvline(0, c='k', alpha=0.5)
plt.axhline(0, c='k', alpha=0.5)
# Display
plt.show()
```

../../_images/lab_E_102_0.png

Do-It-Yourself

In this section, we are going to try

```
import geopandas
import contextily
from pysal.lib import examples
```

Task I: NYC tracts

In this task we will explore contiguity weights.To do it, we will load Census tracts for New York City. Census tracts are the geography the US Census Burearu uses for areas around 4,000 people. We will use a dataset prepared as part of the PySAL examples. Geographically, this is a set of polygons that cover all the area of the city of New York.

A bit of info on the dataset:

```
examples.explain("NYC Socio-Demographics")
```

NYC Education + Socio-Demographics

+
-

To check out the location of the files that make up the dataset, we can load it with `load_example` and inspect with `get_file_list`:

```
# Load example (this automatically downloads if not available)
nyc_data = examples.load_example("NYC Socio-Demographics")
# Print the paths to all the files in the dataset
nyc_data.get_file_list()
```

```
['/home/jovyan/pysal_data/NYC_Socio-Demographics/NYC_Tract_ACS2008_12.prj',
 '/home/jovyan/pysal_data/NYC_Socio-Demographics/NYC_Tract_ACS2008_12.dbf',
 '/home/jovyan/pysal_data/NYC_Socio-
Demographics/__MACOSX/._NYC_Tract_ACS2008_12.dbf',
 '/home/jovyan/pysal_data/NYC_Socio-
Demographics/__MACOSX/._NYC_Tract_ACS2008_12.shp',
 '/home/jovyan/pysal_data/NYC_Socio-
Demographics/__MACOSX/._NYC_Tract_ACS2008_12.shx',
 '/home/jovyan/pysal_data/NYC_Socio-
Demographics/__MACOSX/._NYC_Tract_ACS2008_12.prj',
 '/home/jovyan/pysal_data/NYC_Socio-Demographics/NYC_Tract_ACS2008_12.shp',
 '/home/jovyan/pysal_data/NYC_Socio-Demographics/NYC_Tract_ACS2008_12.shx']
```

And let's read the shapefile:

```
nyc = geopandas.read_file(nyc_data.get_path("NYC_Tract_ACS2008_12.shp"))
nyc.plot(figsize=(9, 9))
```

```
<AxesSubplot:>
```

../../_images/diy_E_7_1.png

Now with the `nyc` object ready to go, here a few tasks for you to complete:

- Create a contiguity matrix using the queen criterion
- Let's focus on [Central Park](https://en.wikipedia.org/wiki/Central_Park). The corresponding polygon is ID `142`. *How many neighbors does it have?*
- Try to reproduce the [zoom plot in the previous section](#).
- Create a block spatial weights matrix where every tract is connected to other tracts in the same borough. For that, use the `boroct2010` column of the `nyc` table.
- Compare the number of neighbors by tract for the two weights matrices, *which one has more? why?*

Task II: Japanese cities

In this task, you will be generating spatial weights matrices based on distance. We will test your skills on this using a [dataset](#) of Japanese urban areas provided by [OECD](#). Let’s get it ready for you to work on it directly.

The data is available over the web on the following addres and we can read it straight into a `GeoDataFrame`:

If you are interested in the original methodology, you can check out in [MMSV20]

```
jp_cities =
geopandas.read_file("http://www.oecd.org/cfe/regionaldevelopment/Japan.zip")
jp_cities.head()
```

	fuacode_si	fuaname	fuaname_en	class_code	iso3	name	
0	JPN19	Kagoshima	Kagoshima	3.0	JPN	Japan	MULTIF Z (((31.6293
1	JPN20	Himeji	Himeji	3.0	JPN	Japan	MULTIF Z (((34.6595
2	JPN50	Hitachi	Hitachi	3.0	JPN	Japan	PO ((36.9444
3	JPN08	Hiroshima	Hiroshima	3.0	JPN	Japan	MULTIF Z (((34.1993
4	JPN03	Toyota	Toyota	4.0	JPN	Japan	MULTIF Z (((34.7324

If we make a quick plot, we can see these are polygons covering the part of the Japanese geography that is considered urban by their analysis:

```
ax = jp_cities.plot(color="red", alpha=0.5, figsize=(9, 9))
contextily.add_basemap(ax, crs=jp_cities.crs)
```

../../_images/diy_E_12_0.png

For this example, we need two transformations: lon/lat coordinates to a geographical projection, and polygons to points. To calculate distances effectively, we need to ensure the coordinates of our geographic data are expressed in metres (or a similar measurement unit). The original dataset is expressed in lon/lat degrees:

```
jp_cities.crs
```

```
<Geographic 2D CRS: EPSG:4326>
Name: WGS 84
Axis Info [ellipsoidal]:
- Lat[north]: Geodetic latitude (degree)
- Lon[east]: Geodetic longitude (degree)
Area of Use:
- name: World
- bounds: (-180.0, -90.0, 180.0, 90.0)
Datum: World Geodetic System 1984
- Ellipsoid: WGS 84
- Prime Meridian: Greenwich
```

We can use the Japan Plane Rectangular CS XVII system (`EPSG:2459`), which is expressed in metres:

```
jp = jp_cities.to_crs(epsg=2459)
```

So the resulting table is in metres:

```
jp.crs
```

```
<Projected CRS: EPSG:2459>
Name: JGD2000 / Japan Plane Rectangular CS XVII
Axis Info [cartesian]:
- X[north]: Northing (metre)
- Y[east]: Easting (metre)
Area of Use:
- name: Japan - zone XVII
- bounds: (131.12, 24.4, 131.38, 26.01)
Coordinate Operation:
- name: Japan Plane Rectangular CS zone XVII
- method: Transverse Mercator
Datum: Japanese Geodetic Datum 2000
- Ellipsoid: GRS 1980
- Prime Meridian: Greenwich
```

Now, distances are easier to calculate between points than between polygons. Hence, we will convert the urban areas into their centroids:

```
jp.geometry = jp.geometry.centroid
```

So the result is a seet of points expressed in metres:

```
jp.plot()
```

```
<AxesSubplot:>
```

../../_images/diy_E_22_1.png

With these at hand, tackle the following challenges:

- Generate a spatial weights matrix with five nearest neighbors
- Generate a spatial weights matrix with a 100km distance band
- Compare the two in terms of average number of neighbors. *What are the main differences you can spot? In which cases do you think one criterion would be preferable over the other?*

Warning

The final task below is a bit more involved, so do not despair if you cannot get it to work completely!

Focus on Tokyo (find the row in the table through a query search as we saw when considering Index-based queries) and the 100km spatial weights generated above. Try to create a figure similar to the one in the lecture. Here's a recipe:

> **Tip**
>
> Be careful with the projections you are using and make sure to plot every dataset in a figure in the same projection!

1. Generate a buffer of 100Km around the Tokyo centroid
2. Start the plot with the Tokyo urban area polygon (`jp_cities`) in one color (e.g. red)
3. Add its neighbors in, say blue
4. Add their centroids in a third different color
5. Layer on top the buffer, making sure only the edge is colored
6. [Optional] Add a basemap

If all goes well, your figure should look, more or less, like:

```
/opt/conda/lib/python3.7/site-packages/libpysal/weights/weights.py:172: UserWarning:
The weights matrix is not fully connected:
 There are 9 disconnected components.
 There are 4 islands with ids: 14, 17, 30, 54.
  warnings.warn(message)
```

../../_images/diy_E_24_1.png

Task III: Spatial Lag

For this task, we will rely on the AHAH dataset. Create the spatial lag of the overall score, and generate a Moran plot. *Can you tell any overall pattern? What do you think it means?*

Check out the notes on how to read the AHAH dataset on the DIY section of block D to refresh your mind before starting the task.

Concepts

In this block we delve into a few statistical methods designed to characterise spatial patterns of data. How phenomena are distributed over space is at the centre of many important questions. From economic inequality, to the management of disease outbreaks, being able to statistically characterise the spatial pattern is the first step into understanding causes and thinking about solutions in the form of policy.

This section is split into a few more chunks than usual, each of them more byte size covering a single concept at a time. Each chunk builds on each other sequentially, so watch them in the order presented here. They are all non-trivial ideas, so focus all your brain power to understand them while tackling each of the sections!

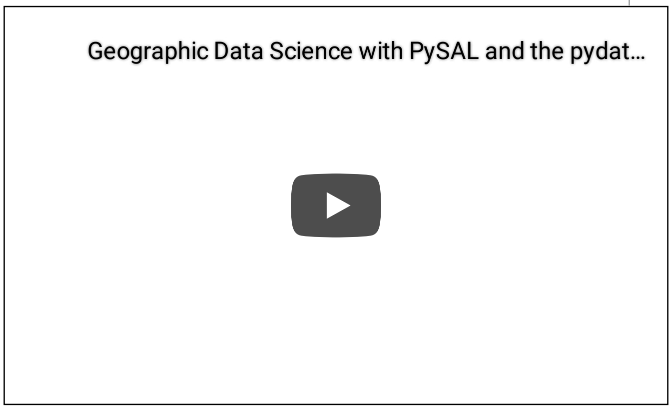
ESDA

ESDA stands for **E**xploratory **S**patial **D**ata **A**nalysis, and it is a family of techniques to explore and characterise spatial patterns in data. This clip introduces ESDA conceptually.

Slides

The slides used in the clip are available at:

- [HTML]
- [PDF]



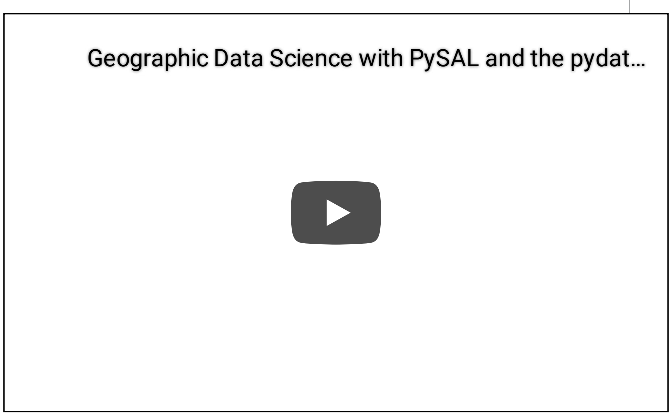
Spatial autocorrelation

In this clip, we define and explain spatial autocorrelation, a core concept to understand what ESDA is about. We also go over the different types and scales at which spatial autocorrelation can be relevant.

Slides

The slides used in the clip are available at:

- [HTML]
- [PDF]



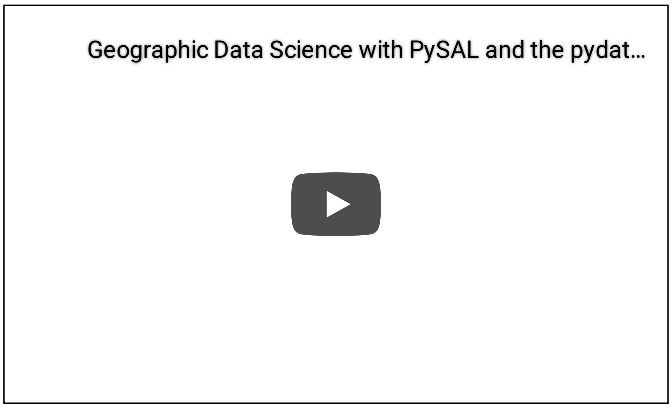
Global spatial autocorrelation

Here we discuss one of the first expressions to formalising spatial patterns: global spatial autocrrelation.

Slides

The slides used in the clip are available at:

- [HTML]
- [PDF]



Once you have seen the clip, you can play with this interactive online:

launch binder

Check out a quick overview on how to run it below on the expandable.

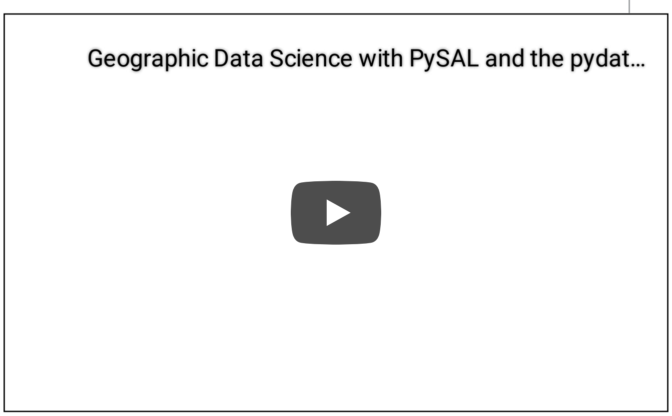
Local spatial autocorrelation

In this final clip, we discuss a more modern concept that takes the notion of spatial autocorrelation to a finer scale.

Slides

The slides used in the clip are available at:

- [HTML]
- [PDF]



If you like this clip and would like to know a bit more about local spatial autocorrelation, the chapter on local spatial autocorrelation in the GDS book (in progress) [RABWng] is a good “next step”.

The chapter is available for free here

Further readings

If this section was of your interest, there is plenty more you can read and explore. The following are good “next steps” to delve a bit deeper into exploratory spatial data analysis:

- Spatial autocorrelation chapter on the GDS book (in progress) [RABWng].
- Symanzik's chapter on ESDA in the Handbook of Regional Science [Sym14] introduces the main concepts behind ESDA
- Haining's chapter in the Handbook of Regional Science [Hai14] is a good historical perspective of the origins and motivations behind most of global and local measures of spatial autocorrelation.

The chapter is available for free here

Hands-on

Spatial autocorrelation and Exploratory Spatial Data Analysis

Spatial autocorrelation has to do with the degree to which the similarity in values between observations in a dataset is related to the similarity in locations of such observations. Not completely unlike the traditional correlation between two variables - which informs us about how the values in one variable change as a function of those in the other- and analogous to its time-series counterpart -which relates the value of a variable at a given point in time with those in previous periods-, spatial autocorrelation relates the value of the variable of interest in a given location, with values of the same variable in surrounding locations.

A key idea in this context is that of spatial randomness: a situation in which the location of an observation gives no information whatsoever about its value. In other words, a variable is spatially random if it is distributed following no discernible pattern over space. Spatial autocorrelation can thus be formally defined as the "absence of spatial randomness", which gives room for two main classes of autocorrelation, similar to the traditional case: *positive* spatial autocorrelation, when similar values tend to group together in similar locations; and *negative* spatial autocorrelation, in cases where similar values tend to be dispersed and further apart from each other.

In this session we will learn how to explore spatial autocorrelation in a given dataset, interrogating the data about its presence, nature, and strength. To do this, we will use a set of tools collectively known as Exploratory Spatial Data Analysis (ESDA), specifically designed for this purpose. The range of ESDA methods is very wide and spans from less sophisticated approaches like choropleths and general table querying, to more advanced and robust methodologies that include statistical inference and an explicit recognition of the geographical dimension of the data. The purpose of this session is to dip our toes into the latter group.

ESDA techniques are usually divided into two main groups: tools to analyze *global*, and *local* spatial autocorrelation. The former consider the overall trend that the location of values follows, and makes possible statements about the degree of *clustering* in the dataset. *Do values generally follow a particular pattern in their geographical distribution*? *Are similar values closer to other similar values than we would expect from pure chance?* These are some of the questions that tools for global spatial autocorrelation allow to answer. We will practice with global spatial autocorrelation by using Moran's I statistic.

Tools for *local* spatial autocorrelation instead focus on spatial instability: the departure of parts of a map from the general trend. The idea here is that, even though there is a given trend for the data in terms of the nature and strength of spatial association, some particular areas can diverege quite substantially from the general pattern. Regardless of the overall degree of concentration in the values, we can observe pockets of unusually high (low) values close to other high (low) values, in what we will call hot(cold)spots. Additionally, it is also possible to observe some high (low) values surrounded by low (high) values, and we will name these "spatial outliers". The main technique we will review in this session to explore local spatial autocorrelation is the Local Indicators of Spatial Association (LISA).

```
%matplotlib inline

import seaborn as sns
import pandas as pd
from pysal.lib import weights
from pysal.explore import esda
from splot.esda import moran_scatterplot, lisa_cluster, plot_local_autocorrelation
import geopandas as gpd
import numpy as np
import contextily as ctx
import matplotlib.pyplot as plt

np.random.seed(123)
```

Data

For this session, we will use the results of the 2016 referendum vote to leave the EU, at the local authority level. In particular, we will focus on the spatial distribution of the vote to Leave, which ended up winning. From a technical point of view, you will be working with polygons which have a value (the percentage of the electorate that voted to Leave the EU) attached to them.

All the necessary data have been assembled for convenience in a single file that contains geographic information about each local authority in England, Wales and Scotland, as well as the vote attributes. The file is in the geospatial format [GeoPackage](), which presents several advantages over the more traditional shapefile (chief among them, the need of a single file instead of several). The file is available as a download from the course website.

```
# Read the file in
br = gpd.read_file("http://darribas.org/gds_course/content/data/brexit.gpkg")
```

Important

Make sure you are connected to the internet when you run this cell

Alternative

Instead of reading the file directly off the web, it is possible to download it manually, store it on your computer, and read it locally. To do that, you can follow these steps:

1. Download the file by right-clicking on this link and saving the file
2. Place the file on the *same folder as the notebook* where you intend to read it
3. Replace the code in the cell above by:

```
br = gpd.read_file("brexit.gpkg")
```

Now let's index it on the local authority IDs, while keeping those as a column too:

```
# Index table on the LAD ID
br = br.set_index("lad16cd", drop=False)
# Display summary
br.info()
```

```
<class 'geopandas.geodataframe.GeoDataFrame'>
Index: 380 entries, E06000001 to W06000024
Data columns (total 5 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   objectid   380 non-null    int64
 1   lad16cd    380 non-null    object
 2   lad16nm    380 non-null    object
 3   Pct_Leave  380 non-null    float64
 4   geometry   380 non-null    geometry
dtypes: float64(1), geometry(1), int64(1), object(2)
memory usage: 17.8+ KB
```

Preparing the data

Let's get a first view of the data:

```
# Plot polygons
ax = br.plot(alpha=0.5, color='red');
# Add background map, expressing target CRS so the basemap can be
# reprojected (warped)
ctx.add_basemap(ax, crs=br.crs)
```

../../_images/lab_F_9_0.png

Spatial weights matrix

As discused before, a spatial weights matrix is the way geographical space is formally encoded into a numerical form so it is easy for a computer (or a statistical method) to understand. We have seen already many of the conceptual ways in which we can define a spatial weights matrix, such as contiguity, distance-based, or block.

For this example, we will show how to build a queen contiguity matrix, which considers two observations as neighbors if they share at least one point of their boundary. In other words, for a pair of local authorities in the dataset to be considered neighbours under this W, they will need to be sharing border or, in other words, "touching" each other to some degree.

Technically speaking, we will approach building the contiguity matrix in the same way we did in Lab 5. We will begin with a `GeoDataFrame` and pass it on to the queen contiguity weights builder in `PySAL` (`ps.weights.Queen.from_dataframe`). We will also make sure our table of data is previously indexed on the local authority code, so the W is also indexed on that form.

```
# Create the spatial weights matrix
%time w = weights.Queen.from_dataframe(br, idVariable="lad16cd")
```

```
CPU times: user 4.09 s, sys: 258 ms, total: 4.35 s
Wall time: 4.34 s
```

```
/opt/conda/lib/python3.7/site-packages/libpysal/weights/weights.py:172:
UserWarning: The weights matrix is not fully connected:
 There are 7 disconnected components.
 There are 6 islands with ids: E06000046, E06000053, S12000013, S12000023,
S12000027, W06000001.
  warnings.warn(message)
```

Now, the `w` object we have just is of the same type of any other one we have created in the past. As such, we can inspect it in the same way. For example, we can check who is a neighbor of observation `E08000012`:

```
w['E08000012']
```

```
{'E08000011': 1.0, 'E08000014': 1.0, 'E06000006': 1.0}
```

However, the cell where we computed W returned a warning on "islands". Remember these are islands not necessarily in the geographic sense (although some of them will be), but in the mathematical sense of the term: local authorities that are not sharing border with any other one and thus do not have any neighbors. We can inspect and map them to get a better sense of what we are dealing with:

```
ax = br.plot(color='k', figsize=(9, 9))
br.loc[w.islands, :].plot(color='red', ax=ax);
```

../../_images/lab_F_16_0.png

In this case, all the islands are indeed "real" islands. These cases can create issues in the analysis and distort the results. There are several solutions to this situation such as connecting the islands to other observations through a different criterium (e.g. nearest neighbor), and then combining both spatial weights matrices. For convenience, we will remove them from the dataset because they are a small sample and their removal is likely not to have a large impact in the calculations.

Technically, this amounts to a subsetting, very much like we saw in the first weeks of the course, although in this case we will use the `drop` command, which comes in very handy in these cases:

```
br = br.drop(w.islands)
```

Once we have the set of local authorities that are not an island, we need to re-calculate the weights matrix:

```python
# Create the spatial weights matrix
# NOTE: this might take a few minutes as the geometries are
#       are very detailed
%time w = weights.Queen.from_dataframe(br, idVariable="lad16cd")
```

```
CPU times: user 2.99 s, sys: 88.6 ms, total: 3.08 s
Wall time: 3.06 s
```

And, finally, let us row-standardize it to make sure every row of the matrix sums up to one:

```python
# Row standardize the matrix
w.transform = 'R'
```

Now, because we have row-standardize them, the weight given to each of the four neighbors is 0.33 which, all together, sum up to one.

```python
w['E08000012']
```

```
{'E08000011': 0.3333333333333333,
 'E08000014': 0.3333333333333333,
 'E06000006': 0.3333333333333333}
```

Spatial lag

Once we have the data and the spatial weights matrix ready, we can start by computing the spatial lag of the percentage of votes that went to leave the EU. Remember the spatial lag is the product of the spatial weights matrix and a given variable and that, if W is row-standardized, the result amounts to the average value of the variable in the neighborhood of each observation.

We can calculate the spatial lag for the variable `Pct_Leave` and store it directly in the main table with the following line of code:

```python
br['w_Pct_Leave'] = weights.lag_spatial(w, br['Pct_Leave'])
```

Let us have a quick look at the resulting variable, as compared to the original one:

```python
br[['lad16cd', 'Pct_Leave', 'w_Pct_Leave']].head()
```

	lad16cd	Pct_Leave	w_Pct_Leave
lad16cd			
E06000001	E06000001	69.57	59.640000
E06000002	E06000002	65.48	60.526667
E06000003	E06000003	66.19	60.376667
E06000004	E06000004	61.73	60.488000
E06000005	E06000005	56.18	57.430000

The way to interpret the spatial lag (`w_Pct_Leave`) for say the first observation is as follow: Hartlepool, where 69,6% of the electorate voted to leave is surrounded by neighbouring local authorities where, on average, almost 60% of the electorate also voted to leave the EU. For the purpose of illustration, we can in fact check this is correct by querying the spatial weights matrix to find out Hartepool's neighbors:

```
w.neighbors['E06000001']
```

```
['E06000004', 'E06000047']
```

And then checking their values:

```
neis = br.loc[w.neighbors['E06000001'], 'Pct_Leave']
neis
```

```
lad16cd
E06000004    61.73
E06000047    57.55
Name: Pct_Leave, dtype: float64
```

And the average value, which we saw in the spatial lag is 61.8, can be calculated as follows:

```
neis.mean()
```

```
59.64
```

For some of the techniques we will be seeing below, it makes more sense to operate with the standardized version of a variable, rather than with the raw one. Standardizing means to substract the average value and divide by the standard deviation each observation of the column. This can be done easily with a bit of basic algebra in Python:

```
br['Pct_Leave_std'] = (br['Pct_Leave'] - br['Pct_Leave'].mean()) /
br['Pct_Leave'].std()
```

Finally, to be able to explore the spatial patterns of the standardized values, also called sometimes z values, we need to create its spatial lag:

```
br['w_Pct_Leave_std'] = weights.lag_spatial(w, br['Pct_Leave_std'])
```

Global Spatial autocorrelation

Global spatial autocorrelation relates to the overall geographical pattern present in the data. Statistics designed to measure this trend thus characterize a map in terms of its degree of clustering and summarize it. This summary can be visual or numerical. In this section, we will walk through an example of each of them: the Moran Plot, and Moran's I statistic of spatial autocorrelation.

Moran Plot

The moran plot is a way of visualizing a spatial dataset to explore the nature and strength of spatial autocorrelation. It is essentially a traditional scatter plot in which the variable of interest is displayed against its spatial lag. In order to be able to interpret values as above or below the mean, and their quantities in terms of standard deviations, the variable of interest is usually standardized by substracting its mean and dividing it by its standard deviation.

Technically speaking, creating a Moran Plot is very similar to creating any other scatter plot in Python, provided we have standardized the variable and calculated its spatial lag beforehand:

```
# Setup the figure and axis
f, ax = plt.subplots(1, figsize=(9, 9))
# Plot values
sns.regplot(x='Pct_Leave_std', y='w_Pct_Leave_std', data=br, ci=None)
# Add vertical and horizontal lines
plt.axvline(0, c='k', alpha=0.5)
plt.axhline(0, c='k', alpha=0.5)
# Display
plt.show()
```

../../_images/lab_F_41_0.png

The figure above displays the relationship between the standardized percentage which voted to Leave the EU (`Pct_Leave_std`) and its spatial lag which, because the W that was used is row-standardized, can be interpreted as the average percentage which voted to Leave in the surrounding areas of a given Local Authority. In order to guide the interpretation of the plot, a linear fit is also included in the post. This line represents the best linear fit to the scatter plot or, in other words, what is the best way to represent the relationship between the two variables as a straight line.

The plot displays a positive relationship between both variables. This is associated with the presence of *positive* spatial autocorrelation: similar values tend to be located close to each other. This means that the *overall trend* is for high values to be close to other high values, and for low values to be surrounded by other low values. This however does not mean that this is only situation in the dataset: there can of course be particular cases where high values are surrounded by low ones, and viceversa. But it means that, if we had to summarize the main pattern of the data in terms of how clustered similar values are, the best way would be to say they are positively correlated and, hence, clustered over space.

In the context of the example, this can be interpreted along the lines of: local authorities display positive spatial autocorrelation in the way they voted in the EU referendum. This means that local authorities with high percentage of Leave voters tend to be located nearby other local authorities where a significant share of the electorate also voted to Leave, and viceversa.

Moran's I

The Moran Plot is an excellent tool to explore the data and get a good sense of how much values are clustered over space. However, because it is a graphical device, it is sometimes hard to condense its insights into a more concise way. For these cases, a good approach is to come up with a statistical measure that summarizes the figure. This is exactly what Moran's I is meant to do.

Very much in the same way the mean summarizes a crucial element of the distribution of values in a non-spatial setting, so does Moran's I for a spatial dataset. Continuing the comparison, we can think of the mean as a single numerical value summarizing a histogram or a kernel density plot. Similarly, Moran's I captures much of the essence of the Moran Plot. In fact, there is an even close connection between the two: the value of Moran's I corresponds with the slope of the linear fit overlayed on top of the Moran Plot.

In order to calculate Moran's I in our dataset, we can call a specific function in `PySAL` directly:

```
mi = esda.Moran(br['Pct_Leave'], w)
```

Note how we do not need to use the standardized version in this context as we will not represent it visually.

The method `ps.Moran` creates an object that contains much more information than the actual statistic. If we want to retrieve the value of the statistic, we can do it this way:

```
mi.I
```

```
0.6228641407137806
```

The other bit of information we will extract from Moran's I relates to statistical inference: how likely is the pattern we observe in the map and Moran's I captures in its value to be generated by an entirely random process? If we considered the same variable but shuffled its locations randomly, would we obtain a map with similar characteristics?

The specific details of the mechanism to calculate this are beyond the scope of the session, but it is important to know that a small enough p-value associated with the Moran's I of a map allows to reject the hypothesis that the map is random. In other words, we can conclude that the map displays more spatial pattern that we would expect if the values had been randomly allocated to a particular location.

The most reliable p-value for Moran's I can be found in the attribute `p_sim`:

```
mi.p_sim
```

```
0.001
```

That is just 0.1% and, by standard terms, it would be considered statistically significant. We can quickly ellaborate on its intuition. What that 0.001 (or 0.1%) means is that, if we generated a large number of maps with the same values but randomly allocated over space, and calculated the Moran's I statistic for each of those maps, only 0.1% of them would display a larger (absolute) value than the one we obtain from the real data, and the other 99.9% of the random maps would receive a smaller (absolute) value of Moran’s I. If we remember again that the value of Moran’s I can also be interpreted as the slope of the Moran Plot, what we have is that, in this case, the particular spatial arrangement of values for the Leave votes is more concentrated than if the values had been allocated following a completely spatially random process, hence the statistical significance.

Once we have calculated Moran’s I and created an object like `mi`, we can use some of the functionality in `splot` to replicate the plot above more easily (remember, [D.R.Y.](D.R.Y.)):

```
moran_scatterplot(mi);
```

../../_images/lab_F_51_0.png

As a first step, the global autocorrelation analysis can teach us that observations do seem to be positively correlated over space. In terms of our initial goal to find spatial structure in the attitude towards Brexit, this view seems to align: if the vote had no such structure, it should not show a pattern over space -technically, it would show a random one.

Local Spatial autocorrelation

Moran’s I is good tool to summarize a dataset into a single value that informs about its degree of *clustering*. However, it is not an appropriate measure to identify areas within the map where specific values are located. In other words, Moran’s I can tell us values are clustered overall, but it will not inform us about *where* the clusters are. For that purpose, we need to use a *local* measure of spatial autocorrelation. Local measures consider each single observation in a dataset and operate on them, as oposed to on the overall data, as *global* measures do. Because of that, they are not good a summarizing a map, but they allow to obtain further insight.

In this session, we will consider Local Indicators of Spatial Association (LISAs), a local counter-part of global measures like Moran’s I. At the core of these method is a classification of the observations in a dataset into four groups derived from the Moran Plot: high values surrounded by high values (HH), low values nearby other low values (LL), high values among low values (HL), and viceversa (LH). Each of these groups are typically called “quadrants”. An illustration of where each of these groups fall into the Moran Plot can be seen below:

```
# Setup the figure and axis
f, ax = plt.subplots(1, figsize=(9, 9))
# Plot values
sns.regplot(x='Pct_Leave_std', y='w_Pct_Leave_std', data=br, ci=None)
# Add vertical and horizontal lines
plt.axvline(0, c='k', alpha=0.5)
plt.axhline(0, c='k', alpha=0.5)
plt.text(1.75, 0.5, "HH", fontsize=25)
plt.text(1.5, -1.5, "HL", fontsize=25)
plt.text(-2, 1, "LH", fontsize=25)
plt.text(-1.5, -2.5, "LL", fontsize=25)
# Display
plt.show()
```

../../_images/lab_F_54_0.png

So far we have classified each observation in the dataset depending on its value and that of its neighbors. This is only half way into identifying areas of unusual concentration of values. To know whether each of the locations is a *statistically significant* cluster of a given kind, we again need to compare it with what we would expect if the data were allocated in a completely random way. After all, by definition, every observation will be of one kind of another, based on the comparison above. However, what we are interested in is whether the strength with which the values are concentrated is unusually high.

This is exactly what LISAs are designed to do. As before, a more detailed description of their statistical underpinnings is beyond the scope in this context, but we will try to shed some light into the intuition of how they go about it. The core idea is to identify cases in which the comparison between the value of an observation and the average of its neighbors is either more similar (HH, LL) or dissimilar (HL, LH) than we would expect from pure chance. The mechanism to do this is similar to the one in the global Moran's I, but applied in this case to each observation, resulting then in as many statistics as original observations.

LISAs are widely used in many fields to identify clusters of values in space. They are a very useful tool that can quickly return areas in which values are concentrated and provide *suggestive* evidence about the processes that might be at work. For that, they have a prime place in the exploratory toolbox. Examples of contexts where LISAs can be useful include: identification of spatial clusters of poverty in regions, detection of ethnic enclaves, delineation of areas of particularly high/low activity of any phenomenon, etc.

In Python, we can calculate LISAs in a very streamlined way thanks to `PySAL`:

```
lisa = esda.Moran_Local(br['Pct_Leave'], w)
```

All we need to pass is the variable of interest -percentage of Leave votes- and the spatial weights that describes the neighborhood relations between the different observation that make up the dataset.

Because of their very nature, looking at the numerical result of LISAs is not always the most useful way to exploit all the information they can provide. Remember that we are calculating a statistic for every sigle observation in the data so, if we have many of them, it will be difficult to extract any meaningful pattern. Instead, what is typically done is to create a map, a cluster map as it is usually called, that extracts the significant observations (those that are highly unlikely to have come from pure chance) and plots them with a specific color depending on their quadrant category.

All of the needed pieces are contained inside the `lisa` object we have created above. But, to make the map making more straightforward, it is convenient to pull them out and insert them in the main data table, `br`:

```
# Break observations into significant or not
br['significant'] = lisa.p_sim < 0.05
# Store the quadrant they belong to
br['quadrant'] = lisa.q
```

Let us stop for second on these two steps. First, the `significant` column. Similarly as with global Moran's I, `PySAL` is automatically computing a p-value for each LISA. Because not every observation represents a statistically significant one, we want to identify those with a p-value small enough that rules out the possibility of obtaining a similar situation from pure chance. Following a similar reasoning as with global Moran's I, we select 5% as the threshold for statistical significance. To identify these values, we create a variable, `significant`, that contains `True` if the p-value of the observation is satisfies the condition, and `False` otherwise. We can check this is the case:

```
br['significant'].head()
```

```
lad16cd
E06000001    False
E06000002    False
E06000003    False
E06000004     True
E06000005    False
Name: significant, dtype: bool
```

And the first five p-values can be checked by:

```
lisa.p_sim[:5]
```

```
array([0.191, 0.087, 0.097, 0.039, 0.212])
```

Note how the third and fourth are smaller than 0.05, as the variable `significant` correctly identified.

Second, the quadrant each observation belongs to. This one is easier as it comes built into the `lisa` object directly:

```
br['quadrant'].head()
```

```
lad16cd
E06000001    1
E06000002    1
E06000003    1
E06000004    1
E06000005    1
Name: quadrant, dtype: int64
```

The correspondence between the numbers in the variable and the actual quadrants is as follows:

- 1: HH
- 2: LH
- 3: LL
- 4: HL

With these two elements, `significant` and `quadrant`, we can build a typical LISA cluster map combining the mapping skills with what we have learned about subsetting and querying tables:

We can create a quick LISA cluster map with `splot`:

```
lisa_cluster(lisa, br);
```

../../_images/lab_F_67_0.png

Or, if we want to have more control over what is being displayed, and how each component is presented, we can “cook” the plot ourselves:

```
# Setup the figure and axis
f, ax = plt.subplots(1, figsize=(9, 9))
# Plot insignificant clusters
ns = br.loc[br['significant']==False, 'geometry']
ns.plot(ax=ax, color='k')
# Plot HH clusters
hh = br.loc[(br['quadrant']==1) & (br['significant']==True), 'geometry']
hh.plot(ax=ax, color='red')
# Plot LL clusters
ll = br.loc[(br['quadrant']==3) & (br['significant']==True), 'geometry']
ll.plot(ax=ax, color='blue')
# Plot LH clusters
lh = br.loc[(br['quadrant']==2) & (br['significant']==True), 'geometry']
lh.plot(ax=ax, color='#83cef4')
# Plot HL clusters
hl = br.loc[(br['quadrant']==4) & (br['significant']==True), 'geometry']
hl.plot(ax=ax, color='#e59696')
# Style and draw
f.suptitle('LISA for Brexit vote', size=30)
f.set_facecolor('0.75')
ax.set_axis_off()
plt.show()
```

../../_images/lab_F_69_0.png

The map above displays the LISA results of the Brexit vote. In bright red, we find those local authorities with an unusual concentration of high Leave voters surrounded also by high levels of Leave vote. This corresponds with areas in the East of England, the Black Country, and East of London. In light red, we find the first type of *spatial outliers*. These are areas with high Leave vote but surrounded by areas with low support for leaving the EU (e.g. central London). Finally, in light blue we find the other type of spatial outlier: local authorities with low Leave support surrounded by other authorities with high support.

The substantive interpretation of a LISA map needs to relate its output to the original intention of the analyst who created the map. In this case, our original idea was to explore the spatial structure of support to leaving the EU. The LISA proves a fairly useful tool in this context. Comparing the LISA map above with the choropleth we started with, we can interpret the LISA as “simplification” of the detailed but perhaps too complicated picture in the choropleth that focuses the reader’s attention to the areas that display a particularly high concentration of (dis)similar values, helping the spatial structure of the vote emerge in a more explicit way. The result of this highlights the relevance that the East of England and the Midlands had in voting to Leave, as well as the regions of the map where there was a lot less excitement about Leaving.

The results from the LISA statistics can be connected to the Moran plot to visualise where in the scatter plot each type of polygon falls:

```
plot_local_autocorrelation(lisa, br, 'Pct_Leave');
```

../../_images/lab_F_72_0.png

Do-It-Yourself

In this block, the DIY section is more straightforward: we have a few tasks, but they are all around the same dataset. The tasks incorporates all the bits and pieces we have seen on the hands-on section.

Data preparation

For this section, we are going to revisit the AHAH dataset we saw in the DIY section of Block D. Please head over to the section to refresh your mind about how to load up the required data. Once you have successfully created the `ahah` object, move on to Task I.

Task I: get the dataset ready

With the `ahah` table on your figertips, complete all the other bits required for the ESDA analysis of spatial autocorrelation:

- Make sure your geography does not have islands

- Create a spatial weights matrix
- Standardise the spatial weights matrix
- Create the standardised version of the AHAH score
- Create the spatial lag of the main AHAH score

When creating your spatial weights matrix, think of one criterium to build it that you think would fit this variable (e.g. contiguity, distance-based, etc.), and apply it.

Task II: global spatial autocorrelation

Let's move on to the analytics:

- Visualise the main AHAH score with a Moran Plot
- Calculate Moran's I
- *What conclusions can you reach from the Moran Plot and Moran's I? What's the main spatial pattern?*

Task III: local spatial autocorrelation

Now that you have a good sense of the overall pattern in the AHAH dataset, let's move to the local scale:

- Calculate LISA statistics for the LSOA areas
- Make a map of significant clusters at the 5%
- *Can you identify hotspots or coldspots? If so, what do they mean? What about spatial outliers?*

- Create cluster maps for significance levels 1% and 10%; compare them with the one we obtained. *What are the main changes? Why?*

Warning

The last action is a bit more sophisticated, put all your brain power into it and you'll achieve it!

Concepts

This block is all about grouping; grouping of *similar* observations, areas, records… We start by discussing why grouping, or clustering in statistical parlance, is important and what it can do for us. Then we move on different types of clustering. We focus on two: one is traditional non-spatial clustering, or unsupervised learning, for which we cover the most popular technique; the other one is explicitly spatial clustering, or regionalisation, which imposes additional (geographic) constraints when grouping observations.

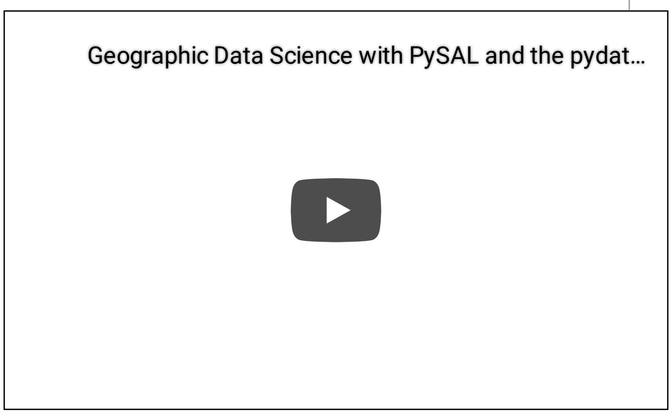
The need to group data

This video motivates the block: *what do we mean by "grouping data" and why is it useful?*

Slides

The slides used in the clip are available at:

- [HTML]
- [PDF]



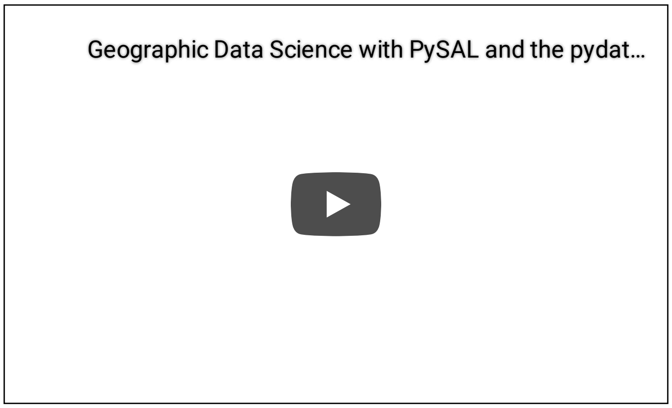
Non-spatial clustering

Non-spatial clustering is the most common form of data grouping. In this section, we cover the basics and mention a few approaches. We wrap it up with an example of clustering very dear to human geography: geodemographics.

Slides

The slides used in the clip are available at:

- [HTML]
- [PDF]



K-Means

In the clip above, we talk about K-Means, by far the most common clustering algorithm. Watch the video on the expandable to get the intuition behind the algorithm and better understand how it does its “magic”.

For a striking visual comparison of how K-Means compares to other clustering algorithms, check out this figure produced by the `scikit-learn` project, a Python package for machine learning (more on this later):

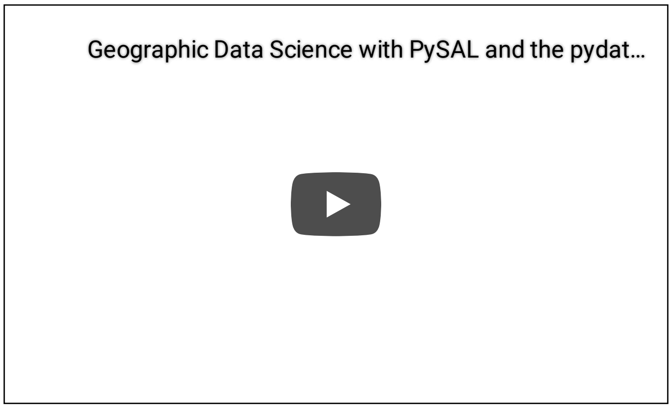
Geodemographics

If you are interested in Geodemographics, a very good reference to get a broader perspective on the idea, origins and history of the field is “The Predictive Postcode” [WB18], by Richard Webber and Roger Burrows. In particular, the first four chapters provide an excellent overview.

Regionalisation

Regionalisation is explicitly spatial clustering. We cover the conceptual basics in the following clip:

> **Slides**
>
> The slides used in the clip are available at:
>
> - `[HTML]`
> - `[PDF]`



If you are interested in the idea of regionalisation, a very good place to continue reading is Duque et al. (2007) [DRSurinach07], which was an important inspiration in structuring the clip.

Further readings

A similar coverage of clustering and regionalisation as provided here, but with a bit more detail, is available on the corresponding chapter of the GDS book (in progress) [RABWng].

The chapter is available for free here

Hands-on

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 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.

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.

```
%matplotlib inline

import seaborn as sns
import pandas as pd
from pysal.lib import weights
import geopandas as gpd
import contextily as cx
import numpy as np
import matplotlib.pyplot as plt
from sklearn import cluster
```

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.

Let's start by reading the main table of MSOAs in:

```
# Read the file in
abb = gpd.read_file("https://darribas.org/gds_course/content/data/london_abb.gpkg")
```

Important

Make sure you are connected to the internet when you run this cell

Alternative

Instead of reading the file directly off the web, it is possible to download it manually, store it on your computer, and read it locally. To do that, you can follow these steps:

1. Download the file by right-clicking on this link and saving the file
2. Place the file on the *same folder as the notebook* where you intend to read it
3. Replace the code in the cell above by:

```
abb = gpd.read_file("london_abb.gpkg")
```

```
# Inspect the structure of the table
abb.info()
```

```
<class 'geopandas.geodataframe.GeoDataFrame'>
RangeIndex: 353 entries, 0 to 352
Data columns (total 18 columns):
 #   Column                       Non-Null Count  Dtype
---  ------                       --------------  -----
 0   MSOA_CODE                    353 non-null    object
 1   accommodates                 353 non-null    float64
 2   bathrooms                    353 non-null    float64
 3   bedrooms                     353 non-null    float64
 4   beds                         353 non-null    float64
 5   number_of_reviews            353 non-null    float64
 6   reviews_per_month            353 non-null    float64
 7   review_scores_rating         353 non-null    float64
 8   review_scores_accuracy       353 non-null    float64
 9   review_scores_cleanliness    353 non-null    float64
 10  review_scores_checkin        353 non-null    float64
 11  review_scores_communication  353 non-null    float64
 12  review_scores_location       353 non-null    float64
 13  review_scores_value          353 non-null    float64
 14  property_count               353 non-null    int64
 15  BOROUGH                      353 non-null    object
 16  GSS_CODE                     353 non-null    object
 17  geometry                     353 non-null    geometry
dtypes: float64(13), geometry(1), int64(1), object(3)
memory usage: 49.8+ KB
```

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:

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

Later in the section, we will also use what AirBnb calls neighborhoods. Let’s load them in so they are ready when we need them.

```
boroughs =
gpd.read_file("https://darribas.org/gds_course/content/data/london_inner_boroughs.geojs
on")
```

Important

Make sure you are connected to the internet when you run this cell

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 readable by humans and by many different platforms. As you can see above, reading them in Python is exactly the same as reading a shapefile, for example.

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:

```
# 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()
```

../../_images/lab_G_12_0.png

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.
- The resulting object, `axs`, is not a single one but a grid (or array) of axis. Because of this, we can't plot directly on `axs`, but instead we need access each individual axis.
- To make that step easier, we *unpack* the grid into a flat list (array) for the axes of each subplot with `flatten` (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`:

```
_ = sns.pairplot(abb[ratings], kind='reg', diag_kind='hist')
```

../../_images/lab_G_14_0.png

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.

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 streamlined 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:

```
kmeans5 = cluster.KMeans(n_clusters=5, random_state=12345)
```

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(`n_clusters`) and the random state, a number that ensures every run of K-Means, which remember relies on random initialisations, is the same and thus reproducible.

To actually run the algorithm on the attributes, we need to call the `fit` method in `kmeans5`:

```
# 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:

```
k5cls.labels_
```

```
array([4, 1, 1, 1, 3, 3, 1, 0, 4, 0, 3, 0, 4, 0, 0, 0, 1, 3, 3, 4, 4, 4,
       2, 2, 2, 2, 2, 4, 4, 2, 4, 4, 0, 1, 3, 3, 3, 3, 1, 3, 0, 3, 3, 3,
       1, 3, 3, 3, 1, 1, 0, 2, 4, 3, 3, 3, 3, 1, 1, 1, 4, 1, 1, 3, 0, 1,
       1, 1, 1, 1, 1, 1, 1, 3, 1, 3, 1, 1, 3, 3, 3, 3, 1, 1, 3, 1, 0, 0,
       0, 3, 3, 0, 3, 3, 0, 3, 3, 4, 3, 4, 0, 2, 4, 4, 0, 3, 3, 0, 4, 1,
       3, 3, 3, 3, 3, 3, 0, 3, 3, 3, 1, 3, 0, 3, 3, 3, 3, 3, 3, 1, 1, 4,
       0, 4, 0, 0, 4, 3, 3, 0, 3, 0, 1, 3, 2, 0, 0, 4, 4, 2, 4, 4, 0, 0,
       0, 0, 4, 0, 0, 0, 0, 1, 3, 3, 0, 1, 1, 1, 1, 1, 1, 1, 3, 0, 1, 1,
       1, 1, 1, 3, 1, 3, 1, 1, 1, 1, 1, 1, 1, 1, 3, 2, 0, 3, 0, 3, 1, 3,
       1, 3, 3, 3, 1, 3, 3, 1, 1, 0, 1, 4, 1, 1, 1, 1, 2, 3, 0, 1, 3, 1,
       1, 3, 2, 1, 1, 0, 3, 4, 3, 0, 3, 3, 0, 4, 0, 1, 4, 4, 0, 4, 4, 3,
       4, 0, 0, 3, 3, 3, 0, 3, 3, 0, 1, 3, 1, 1, 1, 0, 1, 1, 1, 3, 3, 3,
       3, 0, 0, 4, 3, 0, 4, 2, 1, 4, 0, 2, 4, 2, 1, 4, 0, 4, 2, 4, 0, 4,
       4, 3, 0, 4, 2, 0, 3, 3, 3, 3, 3, 1, 3, 3, 1, 3, 1, 1, 0, 3, 1, 4,
       1, 3, 3, 1, 3, 1, 4, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 4, 0, 1, 2,
       0, 0, 4, 4, 4, 0, 4, 2, 4, 0, 2, 4, 0, 2, 2, 4, 0, 4, 3, 0, 4, 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:

```
abb['k5cls'] = k5cls.labels_
```

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:

```
# 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()
# Add title
plt.title('AirBnb Geodemographic classification for Inner London')
# Display the map
plt.show()
```

../../_images/lab_G_27_0.png

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.

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:

```
k5sizes = abb.groupby('k5cls').size()
k5sizes
```

```
k5cls
0     72
1     98
2     23
3    104
4     56
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:

```
_ = k5sizes.plot.bar()
```

../../_images/lab_G_31_0.png

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 on the `groupby` operator which we will combine it with the function `mean`:

```
# Calculate the mean by group
k5means = abb.groupby('k5cls')[ratings].mean()
# Show the table transposed (so it's not too wide)
k5means.T
```

k5cls	0	1	2	
review_scores_rating	92.134328	95.330624	88.322160	93.727
review_scores_accuracy	9.472732	9.717272	9.149055	9.605
review_scores_cleanliness	9.214409	9.478406	8.907681	9.328
review_scores_checkin	9.588242	9.785712	9.413322	9.679
review_scores_communication	9.627248	9.804255	9.444095	9.722
review_scores_location	9.546235	9.539375	9.454598	9.443
review_scores_value	9.220018	9.531206	8.901364	9.384

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.

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.

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:

```
w = weights.Queen.from_dataframe(abb)
```

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:

```
sagg13 = cluster.AgglomerativeClustering(n_clusters=13, connectivity=w.sparse)
sagg13
```

```
AgglomerativeClustering(connectivity=<353x353 sparse matrix of type '<class
'numpy.float64'>'
	with 1978 stored elements in Compressed Sparse Row format>,
                        n_clusters=13)
```

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

```
# Run the clustering algorithm
sagg13cls = sagg13.fit(abb[ratings])
```

And then we append the labels to the table:

```
abb['sagg13cls'] = sagg13cls.labels_
```

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:

```
# 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()
# Add title
plt.title('AirBnb-based boroughs for Inner London')
# Display the map
plt.show()
```

../../_images/lab_G_45_0.png

Comparing organic and administrative delineations

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`:

```python
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:

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

Which we can plot:

```python
# 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()
# Add title
plt.title('AirBnb-based boroughs for Inner London')
```

```
Text(0.5, 1.0, 'AirBnb-based boroughs for Inner London')
```

../../_images/lab_G_51_1.png

The delineation above provides a view into the geography of AirBnb properties. Each region delineated contains houses that, according to our regionalisation algorithm, are more similar with each other than those in the neighboring areas. Now let’s compare this geography that we have organically drawn from our data with that of the official set of administrative boundaries. For example, with the London boroughs.

Remember we read these at the beginning of the notebook:

```python
boroughs.head()
```

	NAME	GSS_CODE	HECTARES	NONLD_AREA	ONS_INN
0	Lambeth	E09000022	2724.940	43.927	
1	Southwark	E09000028	2991.340	105.139	
2	Lewisham	E09000023	3531.706	16.795	
3	Greenwich	E09000011	5044.190	310.785	
4	Wandsworth	E09000032	3522.022	95.600	

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

```
# Setup figure and ax
f, ax = plt.subplots(1, figsize=(6, 6))
# Plot boundary lines
boroughs.plot(ax=ax,
              linewidth=0.5,
              edgecolor='k',
              facecolor='white'
             )
# Remove axis
ax.set_axis_off()
# Add title
plt.title('Administrative boroughs for Inner London')
```

```
Text(0.5, 1.0, 'Administrative boroughs for Inner London')
```

../../_images/lab_G_55_1.png

In order to more easily compare the administrative and the “regionalized” boundary lines, we can overlay them:

The code to create this figure is hidden to facilitate the flow of the narrative but you can toggle it open. It combines building blocks we have seen previously in this course

../../_images/lab_G_57_0.png

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.

Do-It-Yourself

```
import geopandas
import contextily
```

Task I: NYC Geodemographics

We are going to try to get at the (geographic) essence of New York City. For that, we will rely on the same set up Census tracts for New York City we used a few blocks ago. Once you have the `nyc` object loaded, create a geodemographic classification using the following variables:

- `european`: Total Population White
- `asian`: Total Population Asian American
- `american`: Total Population American Indian
- `african`: Total Population African American
- `hispanic`: Total Population Hispanic
- `mixed`: Total Population Mixed race
- `pacific`: Total Population Pacific Islander

For this, make sure you standardise the table by the size of each tract. That is, compute a column with the total population as the sum of all the ethnic groups and divide each of them by that column. This way, the values will range between 0 (no population of a given ethnic group) and 1 (all the population in the tract is of that group).

Once this is ready, get to work with the following tasks:

1. Pick a number of clusters (e.g. 10)
2. Run K-Means for that number of clusters
3. Plot the different clusters on a map
4. Analyse the results:
   - *What do you find?*
   - *What are the main characteristics of each cluster?*
   - *How are clusters distributed geographically?*
   - *Can you identify some groups concentrated on particular areas (e.g. China Town, Little Italy)?*

Task II: Regionalisation of Dar Es Salaam

For this task we will travel to Tanzania’s Dar Es Salaam. We are using a dataset assembled to describe the built environment of the city centre. Let’s load up the dataset before anything:

```
# Read the file in
db =
geopandas.read_file("http://darribas.org/gds_course/content/data/dar_es_salaam.geojson"
)
```

> **Important**
>
> Make sure you are connected to the internet when you run this cell

> **Alternative**
>
> Instead of reading the file directly off the web, it is possible to download it manually, store it on your computer, and read it locally. To do that, you can follow these steps:
>
> 1. Download the file by right-clicking on this link and saving the file
> 2. Place the file on the *same folder as the notebook* where you intend to read it
> 3. Replace the code in the cell above by:
>
> ```
> br = geopandas.read_file("dar_es_salaam.geojson")
> ```

Geographically, this is what we are looking at:

../../_images/diy_G_7_0.png

We can inspect the table:

```
db.info()
```

```
<class 'geopandas.geodataframe.GeoDataFrame'>
RangeIndex: 1291 entries, 0 to 1290
Data columns (total 7 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   index              1291 non-null   object
 1   id                 1291 non-null   object
 2   street_length      1291 non-null   float64
 3   street_linearity   1291 non-null   float64
 4   building_density   1291 non-null   float64
 5   building_coverage  1291 non-null   float64
 6   geometry           1291 non-null   geometry
dtypes: float64(4), geometry(1), object(2)
memory usage: 70.7+ KB
```

Two main aspects of the built environment are considered: the street network and buildings. To capture those, the following variables are calculated at for the H3 hexagonal grid system, zoom level 8:

- Building density: number of buildings per hexagon
- Building coverage: proportion of the hexagon covered by buildings
- Street length: total length of streets within the hexagon

- Street linearity: a measure of how regular the street network is

With these at hand, your task is the following:

> *Develop a regionalisation that partitions Dar Es Salaam based on its built environment*

For that, you can follow these suggestions:

These are only guidelines, feel free to improvise and go beyond what's set. The sky is the limit!

- Create a spatial weights matrix to capture spatial relationships between hexagons
- Set up a regionalisation algorithm with a given number of clusters (e.g. seven)
- Generate a geography that contains only the boundaries of each region and visualise it (ideally with a satellite image as basemap for context)
- Rinse and repeat with several combinations of variables and number of clusters
- Pick your best. *Why have you selected it? What does it show? What are the main groups of areas based on the built environment?*

Concepts

In this block, we focus on a particular type of geometry: points. As we will see, points can represent a very particular type of spatial entity. We explore how that is the case and what are its implications, and then wrap up with a particular machine learning technique that allows us to identify clusters of points in space.

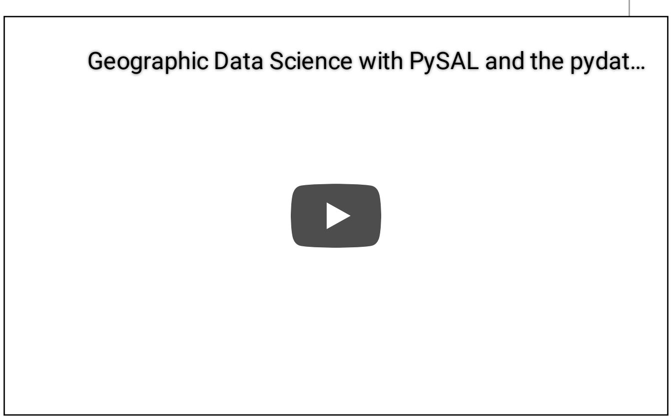
Point patterns

Collections of points referencing geographical locations are sometimes called *point patterns*. In this section, we talk about what's special about point patterns and how they differ from other collections of geographical features such as polygons.

Slides

The slides used in the clip are available at:

- [[HTML]]
- [[PDF]]



Once you have gone over the clip above, watch the one below, featuring Luc Anselin from the University of Chicago providing an overview of point patterns. This will provide a wider perspective on the particular nature of points, but also on their relevance for many disciplines, from ecology to economic geography..

If you want to delve deeper into point patterns, watch the video on the expandable below, which features Luc Anselin delivering a longer (and slightly more advanced) lecture on point patterns.

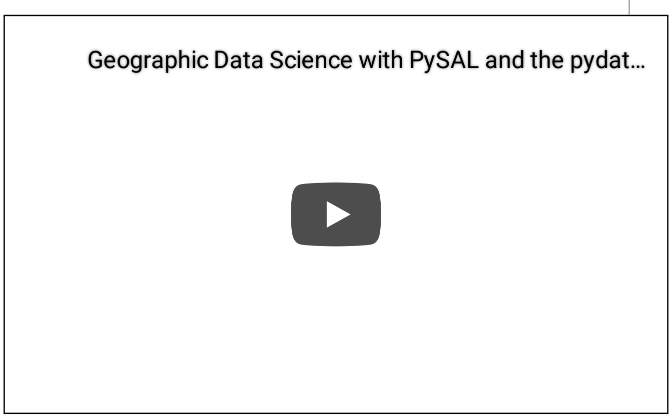
Visualisating Points

Once we have a better sense of what makes points special, we turn to visualising point patterns. Here we cover three main strategies: one to one mapping, aggregation, and smoothing.

Slides

The slides used in the clip are available at:

- [HTML]
- [PDF]



We will put all of these ideas to visualising points into practice on the Hands-on section.

Clustering Points

As we have seen in this course, “cluster” is a hard to define term. In Block G we used it as the outcome of an unsupervised learning algorithm. In this context, we will use the following definition:

> *Concentrations/agglomerations of points over space, significantly more so than in the rest of the space considered*

Spatial/Geographic clustering has a wide literature going back to spatial mathematics and statistics and, more recently, machine learning. For this section, we will cover one algorithm from the latter discipline which has become very popular in the geographic context in the last few years: Density-Based Spatial Clustering of Applications with Noise, or DBSCAN [EKS+96].

Wath the clip below to get the intuition of the algorithm first:

Let’s complement and unpack the clip above in the context of this course. The video does a very good job at explaining how the algorithm works, and what general benefits that entails. Here are two *additional* advantages that are not picked up in the clip:

1. **It is not necessarily spatial**. In fact, the original design was for the area of “data mining” and “knowledge discovery in databases”, which historically does not work with spatial data. Instead, think of purchase histories of consumers, or warehouse stocks: DBSCAN was designed to pick up patterns of similar behaviour in those contexts. Note also that this means you can use DBSCAN not only with two dimensions (e.g. longitude and latitude), but with many more (e.g. product variety) and its mechanics will work in the same way.
2. **Fast and scalable**. For similar reasons, DBSCAN is very fast and can be run in relatively large databases without problem. This contrasts with much of the traditional point pattern methods, that rely heavily on simulation and thus are trickier to scale feasibly. This is one of the reasons why DBSCAN has been widely adopted in Geographic Data Science: it is relatively straightforward to apply and will run fast, even on large datasets, meaning you can iterate over ideas quickly to learn more about your data.

DBSCAN also has a few drawbacks when compared to some of the techniques we have seen earlier in this course. Here are two prominent ones:

1. **It is not based on a probabilistic model**. Unlike the LISAs, for example, there is no underlying model that helps us characterise the pattern the algorithms returns. There is no “null hypothesis” to reject, no inferential model and thus no statistical significance. In some cases, this is an important drawback if we want to ensure what we are observing (and the algorithm is picking up) is not a random pattern.
2. **Agnostic about the underlying process**. Because there is no inferential model and the algorithm imposes very little prior structure to identify clusters, it is also hard to learn anything about the underlying process that gave rise to the pattern picked up by the algorithm. This is by no means a unique feature of DBSCAN, but one that is always good to keep in mind as we are moving from exploratory analysis to more confirmatory approaches.

Further readings

If this section was of your interest, there is plenty more you can read and explore. A good “next step” is the Points chapter on the GDS book (in progress) [RABWng].

The chapter is available for free here

Hands-on

Points

> **Important**
>
> This is an adapted version, with a bit less content and detail, of the chapter on points by Rey, Arribas-Bel and Wolf (*in progress*) [RABWng]. Check out the full chapter, available for free at:
>
> https://geographicdata.science/book/notebooks/08_point_pattern_analysis.htm

Points are spatial entities that can be understood in two fundamentally different ways. On the one hand, points can be seen as fixed objects in space, which is to say their location is taken as given (*exogenous*). In this case, analysis of points is very similar to that of other types of spatial data such as polygons and lines. On the other hand, points can be seen as the occurence of an event that could theoretically take place anywhere but only manifests in certain locations. This is the approach we will adopt in the rest of the notebook.

When points are seen as events that could take place in several locations but only happen in a few of them, a collection of such events is called a *point pattern*. In this case, the location of points is one of the key aspects of interest for analysis. A good example of a point pattern is crime events in a city: they could technically happen in many locations but we usually find crimes are committed only in a handful of them. Point patterns can be *marked*, if more attributes are provided with the location, or *unmarked*, if only the coordinates of where the event occured are provided. Continuing the crime example, an unmarked pattern would result if only the location where crimes were committed was used for analysis, while we would be speaking of a marked point pattern if other attributes, such as the type of crime, the extent of the damage, etc. was provided with the location.

Point pattern analysis is thus concerned with the description, statistical characerization, and modeling of point patterns, focusing specially on the generating process that gives rise and explains the observed data. *What's the nature of the distribution of points? Is there any structure we can statistically discern in the way locations are arranged over space? Why do events occur in those places and not in others?* These are all questions that point pattern analysis is concerned with.

This notebook aims to be a gentle introduction to working with point patterns in Python. As such, it covers how to read, process and transform point data, as well as several common ways to visualize point patterns.

```
%matplotlib inline

import numpy as np
import pandas as pd
import geopandas as gpd
import contextily as cx
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.cluster import DBSCAN
from ipywidgets import interact, fixed
```

Data

Photographs

We are going to dip our toes in the lake of point data by looking at a sample of geo-referenced photographs in Tokyo. The dataset comes from the GDS Book [RABWng] and contains photographs voluntarily uploaded to the Flickr service.

You can read more about the dataset on its page at the GDS Book website

Let's read the dataset first:

```
# Read remote file
tokyo =
pd.read_csv("https://geographicdata.science/book/_downloads/7fb86b605af15b3c9cbd9bfc
bead23e9/tokyo_clean.csv")
```

Important

Make sure you are connected to the internet when you run this cell

> **Alternative**
>
> Instead of reading the file directly off the web, it is possible to download it manually, store it on your computer, and read it locally. To do that, you can follow these steps:
>
> 1. Download the file by right-clicking on this link and saving the file
> 2. Place the file on the *same folder as the notebook* where you intend to read it
> 3. Replace the code in the cell above by:
>
> ```
> tokyo = pd.read_csv("tokyo_clean.csv")
> ```

Administrative areas

We will later use administrative areas for aggregation. Let’s load them upfront first. These are provided with the course and available online:

```
# Read the file in
#areas =
gpd.read_file("https://darribas.org/gds_course/content/data/tokyo_admin_boundaries.g
eojson")
areas = gpd.read_file("../data/tokyo_admin_boundaries.geojson")
```

> **Important**
>
> Make sure you are connected to the internet when you run this cell

> **Alternative**
>
> Instead of reading the file directly off the web, it is possible to download it manually, store it on your computer, and read it locally. To do that, you can follow these steps:
>
> 1. Download the file by right-clicking on this link and saving the file
> 2. Place the file on the *same folder as the notebook* where you intend to read it
> 3. Replace the code in the cell above by:
>
> ```
> areas = gpd.read_file("tokyo_admin_boundaries.geojson")
> ```

The final bit we need to get out of the way is attaching the administrative area code where a photo is located to each area. This can be done with a GIS operation called “spatial join”.

Click the cell below if you are interested in finding out how it works. In the interest of the narrative of this section, we present it collapsed

Now we are good to go!

Visualization of a Point Pattern

We will spend the rest of this notebook learning different ways to visualize a point pattern. In particular, we will consider to main strategies: one relies on aggregating the points into polygons, while the second one is based on creating continuous surfaces using kernel density estimation.

One-to-one

The first approach we review here is the one-to-one approach, where we place a dot on the screen for every point to visualise. In Python, one way to do this is with the `scatter` method in the Pandas visualisation layer:

```
# Plot a dot for every image
tokyo.plot.scatter("longitude", "latitude")
```

```
<AxesSubplot:xlabel='longitude', ylabel='latitude'>
```

../../_images/lab_H_14_1.png

However this does not give us much geographical context and, since there are many points, it is hard to see any pattern in areas of high density. Let’s tweak the dot display and add a basemap:

```
# Plot photographs with smaller, more translucent dots
ax = tokyo.plot.scatter("longitude",
                        "latitude",
                        s=0.25,
                        c="xkcd:bright yellow",
                        alpha=0.5,
                        figsize=(9, 9)
                       )
# remove axis
ax.set_axis_off()
# Add dark basemap
cx.add_basemap(ax,
               crs="EPSG:4326",
               source=cx.providers.CartoDB.DarkMatter
              )
```

../../_images/lab_H_16_0.png

Points meet polygons

The approach presented above works until a certain number of points to plot; tweaking dot transparency and size only gets us so far and, at some point, we need to shift the focus. Having learned about visualizing lattice (polygon) data, an option is to “turn” points into polygons and apply techniques like choropleth mapping to visualize their spatial distribution. To do that, we will overlay a polygon layer on top of the point pattern, *join* the points to the polygons by assigning to each point the polygon where they fall into, and create a choropleth of the counts by polygon.

This approach is intuitive but of course raises the following question: *what polygons do we use to aggregate the points?* Ideally, we want a boundary delineation that matches as closely as possible the point generating process and partitions the space into areas with a similar internal intensity of points. However, that is usually not the case, no less because one of the main reasons we typically want to visualize the point pattern is to learn about such generating process, so we would typically not know a priori whether a set of polygons match it. If we cannot count on the ideal set of polygons to begin with, we can adopt two more realistic approaches: using a set of pre-existing irregular areas or create a artificial set of regular polygons. Let’s explore both.

Irregular lattices

To exemplify this approach, we will use the administrative areas we have loaded above. Let’s add them to the figure above to get better context (unfold the code if you are interested in seeing exactly how we do this):

![../../_images/lab_H_19_0.png]

Now we need to know how many photographs each are contains. Our photograph table already contains the area ID, so all we need to do here is counting by area and attaching the count to the `areas` table. We rely here on the `groupby` operator which takes all the photos in the table and "groups" them "by" their administrative ID. Once grouped, we apply the method `size`, which counts how many elements each group has and returns a column indexed on the LSOA code with all the counts as its values. We end by assigning the counts to a newly created column in the `areas` table.

```
# Create counts
photos_by_area = tokyo.groupby("admin_area").size()
# Assign counts into a table of its own
# and joins it to the areas table
areas = areas.join(pd.DataFrame({"photo_count": photos_by_area}),
                   on="GID_2")
```

The lines above have created a new column in our `areas` table that contains the number of photos that have been taken within each of the polygons in the table.

At this point, we are ready to map the counts. Technically speaking, this is a choropleth just as we have seen many times before:

Tip

Check out Block D if you need a refresher of choropleth maps

```
# Set up figure and axis
f, ax = plt.subplots(1, figsize=(9, 9))
# Plot the equal interval choropleth and add a legend
areas.plot(column='photo_count',
           scheme='quantiles',
           ax=ax,
           legend=True,
           legend_kwds={"loc": 4}
          )
# Remove the axes
ax.set_axis_off()
# Set the title
ax.set_title("Quantile map of photo counts by administrative boundary")
# Add dark basemap
cx.add_basemap(ax,
               crs="EPSG:4326",
               source=cx.providers.CartoDB.DarkMatterNoLabels
              )
# Draw map
plt.show()
```

![../../_images/lab_H_23_0.png]

The map above clearly shows a concentration of photos in the centre of Tokyo. However, it is important to remember that the map is showing *raw* counts of tweets. In the case to photos, as with many other phenomena, it is crucial to keep in mind the "container geography" (see Block D for a refresher of the term). In this case, different administrative areas have different sizes. Everything else equal, a larger polygon may contain more photos, simply because it covers a larger space. To obtain a more accurate picture of the *intensity* of photos by area, what we would like to see is a map of the *density* of photos, not of raw counts. To do this, we can divide the count per polygon by the area of the polygon.

Let's first calculate the area in Sq. metres of each administrative delineation:

```
areas["area_sqm"] = areas.to_crs(epsg=2459).area * 1e-6
```

Note how we need to convert our polygons into a projected CRS. Same as we did with the [Japanese functional urban areas](), we use the [Japan Plane Rectangular CS XVII system]()

Also, we multiply the area by `1e-6` to express the area in squared Km instead of sw. metres

And we can add the photo density as well:

```
areas["photo_density"] = areas["photo_count"] / areas["area_sqm"]
```

With the density at hand, creating the new choropleth is similar as above (check the code in the expandable):

../../_images/lab_H_29_0.png

The pattern in the raw counts is similar to that of density, but we can see how some peripheral, large areas are "downgraded" when correcting for their size, while some smaller polygons in the centre display a higher value.

Regular lattices: hex-binning

Sometimes we either do not have any polygon layer to use or the ones we have are not particularly well suited to aggregate points into them. In these cases, a sensible alternative is to create an artificial topology of polygons that we can use to aggregate points. There are several ways to do this but the most common one is to create a grid of hexagons. This provides a regular topology (every polygon is of the same size and shape) that, unlike circles, cleanly exhausts all the space without overlaps and has more edges than squares, which alleviates edge problems.

Python has a simplified way to create this hexagon layer *and* aggregate points into it in one shot thanks to the method `hexbin`, which is available in every axis object (e.g. `ax`). Let us first see how you could create a map of the hexagon layer alone:

```
# Setup figure and axis
f, ax = plt.subplots(1, figsize=(9, 9))
# Add hexagon layer that displays count of points in each polygon
hb = ax.hexbin(tokyo["longitude"],
               tokyo["latitude"],
               gridsize=50,
              )
# Add a colorbar (optional)
plt.colorbar(hb)
```

```
<matplotlib.colorbar.Colorbar at 0x7f0f11fdb2d0>
```

../../_images/lab_H_32_1.png

See how all it takes is to set up the figure and call `hexbin` directly using the set of coordinate columns (`tokyo["longitude"]` and `tokyo["latitude"]`). Additional arguments we include is the number of hexagons by axis (`gridsize`, 50 for a 50 by 50 layer), and the transparency we want (80%). Additionally, we include a colorbar to get a sense of how counts are mapped to colors. Note that we need to pass the name of the object that includes the `hexbin` (`hb` in our case), but keep in mind this is optional, you do not need to always create one.

Once we know the basics, we can dress it up a bit more for better results (expand to see code):

../../_images/lab_H_34_0.png

Kernel Density Estimation

Using a hexagonal binning can be a quick solution when we do not have a good polygon layer to overlay the points directly and some of its properties, such as the equal size of each polygon, can help alleviate some of the problems with a “bad” irregular topology (one that does not fit the underlying point generating process). However, it does not get around the issue of the modifiable areal unit problem (M.A.U.P., see Block D: at the end of the day, we are still imposing arbitrary boundary lines and aggregating based on them, so the possibility of mismatch with the underlying distribution of the point pattern is very real.

One way to work around this problem is to avoid aggregating into another geography altogether. Instead, we can aim at estimating the *continuous* observed probability distribution. 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.

The actual algorithm to estimate a kernel density is not trivial but its application in Python is rather simplified by the use of Seaborn. KDE’s however are fairly computationally intensive. When you have a large point pattern like we do in the `tokyo` example (10,000 points), its computation can take a bit long. To get around this issue, we create a random subset, which retains the overall structure of the pattern, but with much fewer points. Let’s take a subset of 1,000 random points from our original table:

```
# Take a random subset of 1,000 rows from `tokyo`
tokyo_sub = tokyo.sample(1000, random_state=12345)
```

Note we need to specify the size of the resulting subset (1,000), and we also add a value for `random_state`; this ensures that the sample is always the same and results are thus reproducible.

Same as above, let us first see how to create a quick KDE. For this we rely on Seaborn’s `kdeplot`:

```
sns.kdeplot(tokyo_sub["longitude"],
            tokyo_sub["latitude"],
            n_levels=50,
            shade=True,
            cmap='BuPu'
           )
```

```
<AxesSubplot:xlabel='longitude', ylabel='latitude'>
```

../../_images/lab_H_38_1.png

Seaborn greatly streamlines the process and boils it down to a single line. The method `sns.kdeplot` (which we can also use to create a KDE of a single variable) takes the X and Y coordinate of the points as the only compulsory attributes. In addition, we specify the number of levels we want the color gradient to have (`n_levels`), whether we want to color the space in between each level (`share`, yes), and the colormap of choice.

Once we know how the basic logic works, we can insert it into the usual mapping machinery to create a more complete plot. The main difference here is that we now have to tell `sns.kdeplot` where we want the surface to be added (`ax` in this case). Toggle the expandable to find out the code that produces the figure below:

../../_images/lab_H_40_0.png

Clusters of points

In this final section, we will learn a method to identify clusters of points, based on their density across space. To do this, we will use the widely used `DBSCAN` algorithm. For this method, a cluster is a concentration of at least `m` points, each of them within a distance of `r` of at least another point in the cluster. Points in the dataset are then divided into three categories:

- *Noise*, for those points outside a cluster.
- *Cores*, for those points inside a cluster whith at least `m` points in the cluster within distance `r`.
- *Borders* for points inside a cluster with less than `m` other points in the cluster within distance `r`.

Both `m` and `r` need to be prespecified by the user before running `DBSCAN`. This is a critical point, as their value can influence significantly the final result. Before exploring this in greater depth, let us get a first run at computing `DBSCAN` in Python.

Basics

The heavy lifting is done by the method `DBSCAN`, part of the excellent machine learning library `scikit-learn`. Running the algorithm is similar to how we ran K-Means when [clustering](). We first set up the details:

```
# Set up algorithm
algo = DBSCAN(eps=100, min_samples=50)
```

We decide to consider a cluster photos with more than 50 photos within 100 metres from them, hence we set the two parameters accordingly. Once ready, we *"fit"* it to our data, but note that we first need to express the longitude and latitude of our points in metres (see code for that on the side cell).

```
algo.fit(tokyo[["X_metres", "Y_metres"]])
```

```
DBSCAN(eps=100, min_samples=50)
```

```
## Express points in metres
# Convert lon/lat into Point objects +
set CRS
pts =
gpd.points_from_xy(tokyo["longitude"],

tokyo["latitude"],
                             crs="EPSG:4326"
                            )
# Convert lon/lat points to Japanese CRS
in metres
pts = gpd.GeoDataFrame({"geometry":
pts}).to_crs(epsg=2459)
# Extract coordinates from point objects
into columns
tokyo["X_metres"] = pts.geometry.x
tokyo["Y_metres"] = pts.geometry.y
```

Once fit, we can recover the labels:

```
algo.labels_
```

```
array([-1, -1, -1, ...,  8, -1, -1])
```

And the list of points classified as cores:

```
# Print only the first five values
algo.core_sample_indices_[:5]
```

```
array([12, 25, 28, 46, 63])
```

The `labels_` object always has the same length as the number of points used to run `DBSCAN`. Each value represents the index of the cluster a point belongs to. If the point is classified as *noise*, it receives a -1. Above, we can see that the first five points are effectively not part of any cluster. To make thinks easier later on, let us turn the labels into a `Series` object that we can index in the same way as our collection of points:

```
lbls = pd.Series(algo.labels_, index=tokyo.index)
```

Now we already have the clusters, we can proceed to visualize them. There are many ways in which this can be done. We will start just by coloring points in a cluster in red and noise in grey:

```
# Setup figure and axis
f, ax = plt.subplots(1, figsize=(6, 6))
# Assign labels to tokyo table dynamically and
# subset points that are not part of any cluster (noise)
noise = tokyo.assign(lbls=lbls)\
             .query("lbls == -1")
# Plot noise in grey
ax.scatter(noise["X_metres"],
           noise["Y_metres"],
           c='grey',
           s=5,
           linewidth=0
          )
# Plot all points that are not noise in red
# NOTE how this is done through some fancy indexing, where
#      we take the index of all points (tw) and substract from
#      it the index of those that are noise
ax.scatter(tokyo.loc[tokyo.index.difference(noise.index),
                     "X_metres"
                    ],
           tokyo.loc[tokyo.index.difference(noise.index),
                     "Y_metres"
                    ],
           c="red",
           linewidth=0
          )
# Display the figure
plt.show()
```

../../_images/lab_H_53_0.png

This is a first good pass. The algorithm is able to identify a few clusters with high density of photos. However, as we mentioned [when discussing DBSCAN](), this is all contingent on the parameters we arbitrarily set. Depending on the maximum radious (`eps`) we set, we will pick one type of cluster or another: a higher (lower) radious will translate in less (more) local clusters. Equally, the minimum number of points required for a cluster (`min_samples`) will affect the implicit size of the cluster. Both parameters need to be set before running the algorithm, so our decision will affect the final outcome quite significantly.

For an illustration of this, let's run through a case with very different parameter values. For example, let's pick a larger radious (e.g. 500m) and a smaller number of points (e.g. 10):

```
# Set up algorithm
algo = DBSCAN(eps=500, min_samples=10)
# Fit to Tokyo projected points
algo.fit(tokyo[["X_metres", "Y_metres"]])
# Store labels
lbls = pd.Series(algo.labels_, index=tokyo.index)
```

And let's now visualise the result (toggle the expandable to see the code):

../../_images/lab_H_57_0.png

The output is now very different, isn't it? This exemplifies how different parameters can give rise to substantially different outcomes, even if the same data and algorithm are applied.

Advanced plotting

> **Warning**
>
> Please keep in mind this final section of the tutorial is **OPTIONAL**, so do not feel forced to complete it. This will not be covered in the assignment and you will still be able to get a good mark without completing it (also, including any of the following in the assignment does NOT guarantee a better mark).

As we have seen, the choice of parameters plays a crucial role in the number, shape and type of clusters founds in a dataset. To allow an easier exploration of these effects, in this section we will turn the computation and visualization of `DBSCAN` outputs into a single function. This in turn will allow us to build an interactive tool later on.

Below is a function that accomplishes just that:

```python
def clusters(db, eps, min_samples):
    '''
    Compute and visualize DBSCAN clusters
    ...

    Arguments
    ---------
    db          : (Geo)DataFrame
                  Table with at least columns `X` and `Y` for point coordinates
    eps         : float
                  Maximum radious to search for points within a cluster
    min_samples : int
                  Minimum number of points in a cluster
    '''
    algo = DBSCAN(eps=eps, min_samples=min_samples)
    algo.fit(db[['X_metres', 'Y_metres']])
    lbls = pd.Series(algo.labels_, index=db.index)

    f, ax = plt.subplots(1, figsize=(6, 6))
    noise = db.loc[lbls==-1, ['X_metres', 'Y_metres']]
    ax.scatter(noise['X_metres'], noise['Y_metres'], c='grey', s=5, linewidth=0)
    ax.scatter(db.loc[db.index.difference(noise.index), 'X_metres'], \
               db.loc[db.index.difference(noise.index), 'Y_metres'], \
              c='red', linewidth=0)
    return plt.show()
```

The function takes the following three arguments:

1. `db`: a `(Geo)DataFrame` containing the points on which we will try to find the clusters.
2. `eps`: a number (maybe with decimals, hence the `float` label in the documentation of the function) specifying the maximum distance to look for neighbors that will be part of a cluster.
3. `min_samples`: a count of the minimum number of points required to form a cluster.

Let us see how the function can be used. For example, let us replicate the plot above, with a minimum of 10 points and a maximum radious of 500 metres:

```python
clusters(tokyo, 500, 10)
```

../../_images/lab_H_61_0.png

Voila! With just one line of code, we can create a map of `DBSCAN` clusters. How cool is that?

However, this could be even more interesting if we didn't have to write each time the parameters we want to explore. To change that, we can create a quick interactive tool that will allow us to modify both parameters with sliders. To do this, we will use the library `ipywidgets`. Let us first do it and then we will analyse it bit by bit:

```python
interact(
    clusters,                 # Method to make interactive
    db=fixed(tokyo),          # Data to pass on db (does not change)
    eps=(50, 500, 50),        # Range start/end/step of eps
    min_samples=(50, 300, 50) # Range start/end/step of min_samples
);
```

Phew! That is cool, isn't it? Once passed the first excitement, let us have a look at how we built it, and how you can modify it further on. A few points on this:

- First, `interact` is a method that allows us to pass an arbitrary function (like `clusters`) and turn it into an interactive widget where we modify the values of its parameters through sliders, drop-down menus, etc.
- What we need to pass to `interact` is the name of the function we would like to make interactive (`clusters` in this case), and all the parameters it will take.
- Since in this case we do not wish to modify the dataset that is used, we pass `tokyo` as the `db` argument in `clusters` and fixate it by passing it first to the `fixed` method.
- Then both the radious `eps` and the minimum cluster size `min_samples` are passed. In this case, we do want to allow interactivity, so we do not use `fixed`. Instead, we pass a tuple that specifies the range and the step of the values we will allow to be used.
- In the case of `eps`, we use `(50, 500, 50)`, which means we want `r` to go from 50 to 500, in jumps of 50 units at a time. Since these are specified in metres, we are saying we want the range to go from 50 to 500 metres in increments of 50 metres.
- In the case of `min_samples`, we take a similar approach and say we want the minimum number of points to go from 50 to 300, in steps of 50 points at a time.

The above results in a little interactive tool that allows us to play easily and quickly with different values for the parameters and to explore how they affect the final outcome.

Do-It-Yourself

```
import pandas, geopandas, contextily
```

Task I: AirBnb distribution in Beijing

In this task, you will explore patterns in the distribution of the location of AirBnb properties in Beijing. For that, we will use data from the same provider as we did for the clustering block: Inside AirBnb. We are going to read in a file with the locations of the properties available as of August 15th. 2019:

```
url = ("http://data.insideairbnb.com/china/beijing/beijing/"\
       "2019-08-15/visualisations/listings.csv")
url
```

```
'http://data.insideairbnb.com/china/beijing/beijing/2019-08-
15/visualisations/listings.csv'
```

```
abb = pandas.read_csv(url)
```

Important

Make sure you are connected to the internet when you run this cell

Alternative

Instead of reading the file directly off the web, it is possible to download it manually, store it on your computer, and read it locally. To do that, you can follow these steps:

1. Download the file by right-clicking on [this link] and saving the file
2. Place the file on the *same folder as the notebook* where you intend to read it
3. Replace the code in the cell above by:

```
abb = pandas.read_csv("listings.csv")
```

This gives us a table with the following information:

```
abb.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 34400 entries, 0 to 34399
Data columns (total 16 columns):
 #   Column                          Non-Null Count  Dtype  
---  ------                          --------------  -----  
 0   id                              34400 non-null  int64  
 1   name                            34399 non-null  object 
 2   host_id                         34400 non-null  int64  
 3   host_name                       34393 non-null  object 
 4   neighbourhood_group             0 non-null      float64
 5   neighbourhood                   34400 non-null  object 
 6   latitude                        34400 non-null  float64
 7   longitude                       34400 non-null  float64
 8   room_type                       34400 non-null  object 
 9   price                           34400 non-null  int64  
 10  minimum_nights                  34400 non-null  int64  
 11  number_of_reviews               34400 non-null  int64  
 12  last_review                     20961 non-null  object 
 13  reviews_per_month               20961 non-null  float64
 14  calculated_host_listings_count  34400 non-null  int64  
 15  availability_365                34400 non-null  int64  
dtypes: float64(4), int64(7), object(5)
memory usage: 4.2+ MB
```

Also, for an ancillary geography, we will use the neighbourhoods provided by the same source:

```
url = ("http://data.insideairbnb.com/china/beijing/beijing/"\
       "2019-08-15/visualisations/neighbourhoods.geojson")
url
```

```
'http://data.insideairbnb.com/china/beijing/beijing/2019-08-
15/visualisations/neighbourhoods.geojson'
```

```
neis = geopandas.read_file(url)
```

Important

Make sure you are connected to the internet when you run this cell

Alternative

Instead of reading the file directly off the web, it is possible to download it manually, store it on your computer, and read it locally. To do that, you can follow these steps:

1. Download the file by right-clicking on [this link]() and saving the file
2. Place the file on the *same folder as the notebook* where you intend to read it
3. Replace the code in the cell above by:

```
neis = geopandas.read_file("neighbourhoods.geojson")
```

```
neis.info()
```

```
<class 'geopandas.geodataframe.GeoDataFrame'>
RangeIndex: 16 entries, 0 to 15
Data columns (total 3 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   neighbourhood       16 non-null     object
 1   neighbourhood_group  0 non-null      object
 2   geometry            16 non-null     geometry
dtypes: geometry(1), object(2)
memory usage: 512.0+ bytes
```

With these at hand, get to work with the following challenges:

- Create a Hex binning map of the property locations
- Compute and display a kernel density estimate (KDE) of the distribution of the properties
- Using the neighbourhood layer:
    - Obtain a count of property by neighbourhood (nothe the neighbourhood name is present in the property table and you can connect the two tables through that)
    - Create a raw count choropleth
    - Create a choropleth of the density of properties by polygon

Task II: Clusters of Indian cities

For this one, we are going to use a dataset on the location of populated places in India provided by `http://geojson.xyz`. The original table covers the entire world so, to get it ready for you to work on it, we need to prepare it:

```
url = ("https://d2ad6b4ur7yvpq.cloudfront.net/naturalearth-3.3.0/"\
       "ne_50m_populated_places_simple.geojson")
url
```

```
'https://d2ad6b4ur7yvpq.cloudfront.net/naturalearth-
3.3.0/ne_50m_populated_places_simple.geojson'
```

Let’s read the file in and keep only places from India:

```
places = geopandas.read_file(url).query("adm0name == 'India'")
```

Important

Make sure you are connected to the internet when you run this cell

Alternative

Instead of reading the file directly off the web, it is possible to download it manually, store it on your computer, and read it locally. To do that, you can follow these steps:

1. Download the file by right-clicking on this link and saving the file
2. Place the file on the *same folder as the notebook* where you intend to read it
3. Replace the code in the cell above by:

```
places = geopandas.read_file("ne_50m_populated_places_simple.geojson")
```

By defaul, place locations come expressed in longitude and latitude. Because you will be working with distances, it makes sense to convert the table into a system expressed in metres. For India, this can be the "Kalianpur 1975 / India zone I" (`EPSG:24378`) projection.

```
places_m = places.to_crs(epsg=24378)
```

This is what we have to work with then:

```
ax = places_m.plot(color="xkcd:bright yellow",
                   figsize=(9, 9)
                  )
contextily.add_basemap(ax,
                       crs=places_m.crs,
                       source=contextily.providers.CartoDB.DarkMatter
                      )
```

../../_images/diy_H_24_0.png

With this at hand, get to work:

- Use the DBSCAN algorithm to identify clusters
- Start with the following parameters: at least five cities for a cluster (`min_samples`) and a maximum of 1,000Km (`eps`)
- Obtain the clusters and plot them on a map. *Does it pick up any interesting pattern?*
- Based on the results above, tweak the values of both parameters to find a cluster of southern cities, and another one of cities in the North around New Dehli

By Dani Arribas-Bel