

Home

GDS4AE - *Geographic Data Science for Applied Economists*

- [Dani Arribas-Bel](#) [[@darribas](#)].
- [Diego Puga](#) [[@ProfDiegoPuga](#)].

Contact

Dani Arribas-Bel - [D.Arribas-Bel \[at\] liverpool.ac.uk](mailto:D.Arribas-Bel@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.

Diego Puga - [diego.puga \[at\] cemfi.es](mailto:diego.puga@cemfi.es)

Professor
CEMFI,
Casado del Alisal 5,
28014 Madrid,
Spain.

Citation

If you use materials from this resource in your own work, we recommend the following citation:

```
@article{darribas_gds_course,  
  author = {Dani Arribas-Bel and Diego Puga},  
  title = {Geographic Data Science for Applied Economists},  
  year = 2021,  
  annote = {\href{https://darribas.org/gds4ae}}  
}
```

Overview

This resource provides an introduction to Geographic Data Science for applied economists using Python. It has been designed to be delivered within 15 hours of teaching, split into ten sessions of 1.5h each.

How to follow along

[GDS4AE](#) is best followed if you can interactively tinker with its content. To do that, you will need two things:

1. A computer set up with the Jupyter Lab environment and all the required libraries (please see the [Software stack](#) part in the [Infrastructure](#) section for instructions)

☰ Contents

Overview

[Overview](#)

[Infrastructure](#)

Content

[Introduction](#)

[Spatial Data](#)

[Geovisualisation](#)

[Spatial Feature Engineering.\(I\)](#)

[Spatial Feature Engineering.\(II\)](#)

[Spatial Networks.\(I\)](#)

[Spatial Networks.\(II\)](#)

[Transport costs](#)

[Visual challenges and opportunities](#)

Epilogue

[Datasets](#)

[Further Resources](#)

[Bibliography](#)

2. A local copy of the materials that you can run on your own computer (see the [repository](#) section in the [Infrastructure](#) section for instructions)

Blocks have different components:

- [Ahead of time...](#): materials to go on your own ahead of the live session
- [Hands-on coding](#): content for the live session
- [Next steps](#): a few pointers to continue your journey on the area the block covers

Content

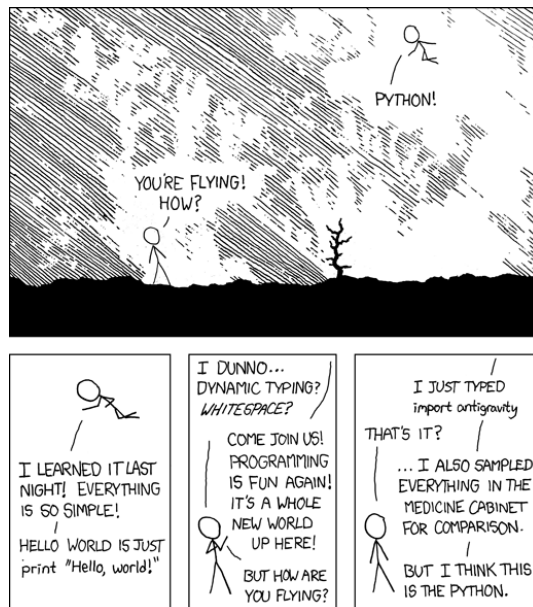
The structure of content is divided in nine blocks:

- [Introduction](#): get familiar with the computational environment of modern data science
- [Spatial Data](#): what do spatial data look like in Python?
- [Geovisualisation](#): make (good) data maps
- [Spatial Feature Engineering](#) ([Part I](#) and [Part II](#)): augment and massage your data using Geography before you feed them into your model
- [Spatial Networks](#) ([Part I](#) and [Part II](#)): understand, acquire and work with spatial graphs
- [Transport Costs](#): “getting there” doesn’t always cost the same
- [Visual challenges](#): all the details nobody told you (but should have) about visualising geographic data

Each block has its own section and is designed to be delivered in 1.5 hours approximately. The content of some of these blocks relies on external resources, all of them freely available. When that is the case, enough detail is provided in the to understand how additional material fits in.

Why Python?

There are several reasons why we have made this choice. Many of them are summarised nicely in [this article by The Economist](#) (paywalled).:w



Source: [XKCD](#)

Data

All the datasets used in this resource is freely available. Some of them have been developed in the context of the resource, others are borrowed from other resources. A full list of the datasets used, together with links to the original source, or to reproducible code to generate the data used is available in the [Datasets](#) page.

License

The materials in this course are published under a [Creative Commons BY-SA 4.0](#) license. This grants you the right to use them freely and (re-)distribute them so long as you give credit to the original creators (see the [Home page](#) for a suggested citation) and license derivative work under the same license.

Infrastructure

This page covers a few technical aspects on how the course is built, kept up to date, and how you can create a computational environment to run all the code it includes.

Software stack

This course is best followed if you can not only read its content but also interact with its code and even branch out to write your own code and play on your own. For that, you will need to have installed on your computer a series of interconnected software packages; this is what we call a *stack*.

Instructions on how to install a software stack that allows you to run the materials of this course depend on the operating system you are using. Detailed guides are available for the main systems on the following resource, provided by the [Geographic Data Science Lab](#):

Github repository

All the materials for this course and this website are available on the following Github repository:

If you are interested, you can download a compressed [.zip](#) file with the most up-to-date version of all the materials, including the HTML for this website at:

Icon made by [Freepik](#) from [www.flaticon.com](#)

Containerised backend

The course is developed, built and tested using the [gds_env](#), a containerised platform for Geographic Data Science. You can read more about the [gds_env](#) project at:

Binder

[Binder](#) is service that allows you to run scientific projects in the cloud for free. Binder can spin up “ephemeral” instances that allow you to run code on the browser without any local setup. It is possible to run the course on Binder by clicking on the button below:



Warning

It is important to note Binder instances are *ephemeral* in the sense that the data and content created in a session is **NOT** saved anywhere and is deleted as soon as the browser tab is closed.

Binder is also the backend this website relies on when you click on the rocket icon (🚀) on a page with code. Remember, you can play with the code interactively but, once you close the tab, all the changes are lost.

Introduction

Geographic Data Science

Note

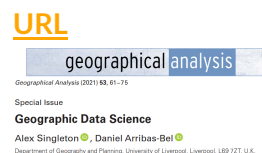
This section is adapted from [Block A](#) of the GDS Course [\[AB19\]](#).

Before we learn *how* to do Geographic Data Science or even *why* you would want to do it, let's start with *what* it is. We will rely on two resources:

- First, in this video, Dani Arribas-Bel covers the building blocks at the First [Spatial Data Science Conference](#), organised by [CARTO](#)



- Second, *Geographic Data Science*, by Alex Singleton and Dani Arribas-Bel [\[SAB19\]](#)



The computational stack

One of the core learning outcomes of this course is to get familiar with the modern computational environment that is used across industry and science to “do” Data Science. In this section, we will learn about ecosystem of concepts and tools that come together to provide the building blocks of much computational work in data science these days.

Source: [The Atlantic](#)



The Scientific Paper is Dissected
Here's what's next.

- *Ten simple rules for writing and sharing computational analyses in Jupyter Notebooks*, by Adam Rule et al. [\[RBZ+19\]](#)



[URL](#)

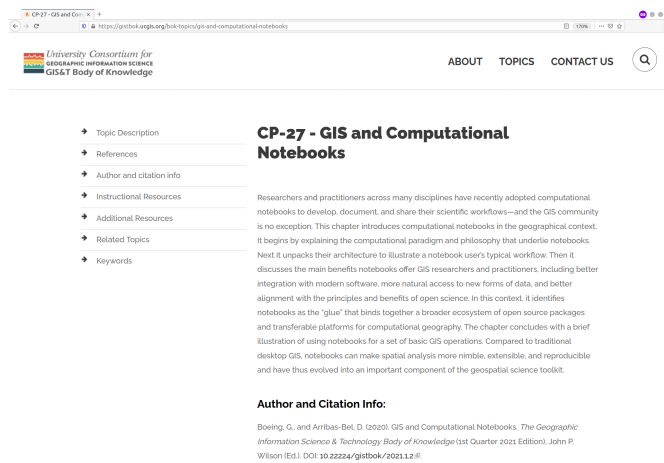
EDITORIAL

Ten simple rules for writing and sharing computational analyses in Jupyter Notebooks

Adam Rule¹, Amanda Birmingham², Crista Zuniga³, Ilkay Altintas⁴, Shih-Cheng Huang⁵, Rob Knight⁶, Niema Moshiri⁷, Mai H. Nguyen⁸, Sara Brin Rosenthal⁹, Fernando Pérez¹⁰, Peter W. Rose¹¹

1 Design Lab, UC San Diego, La Jolla, California, United States of America, **2** Center for Computational Biology and Bioinformatics, UC San Diego, La Jolla, California, United States of America, **3** Department of Pediatrics, UC San Diego, La Jolla, California, United States of America, **4** Data Science Hub, San Diego Supercomputer Center, UC San Diego, La Jolla, California, United States of America, **5** Departments of Bioengineering, and Computer Science and Engineering, and Center for Microbiome Innovation, UC San Diego, La Jolla, California, United States of America, **6** Bioinformatics and Systems Biology Graduate Program, UC San Diego, La Jolla, California, United States of America, **7** Department of Statistics and Berkeley Institute for Data Science, UC Berkeley, and Lawrence Berkeley National Laboratory, Berkeley, California, United States of America

- *GIS and Computational Notebooks*, by Geoff Boeing and Dani Arribas-Bel [\[BAB20\]](#)



[URL](#)

Now we are familiar with the conceptual pillars on top of which we will be working, let's switch gears into a more practical perspective. The following two clips cover the basics of Jupyter Lab, the frontend that glues all the pieces together, and Jupyter Notebooks, the file format, application, and protocol that allows us to record, store and share workflows.

Note

The clips are sourced from [BlockA](#) of the GDS Course [\[AB19\]](#)



Jupyter Notebooks



Spatial Data

□ Ahead of time...

This block is all about understanding spatial data, both conceptually and practically. Before your fingers get on the keyboard, the following readings will help you get going and familiar with core ideas:

- [Chapter 2](#) of the GDS Book [[RABWng](#)], which provides a conceptual overview of representing Geography in data
- [Chapter 3](#) of the GDS Book [[RABWng](#)], a sister chapter with a more applied perspective on how concepts are implemented in computer data structures

Additionally, parts of this block are based and source from [Block C](#) in the GDS Course [[AB19](#)].

□ Hands-on coding

(Geographic) tables

```
import pandas
import geopandas
import xarray, rioxarray
import contextily
import matplotlib.pyplot as plt
```

Points

Data

If you want to read more about the data sources behind this dataset, head to the [Datasets](#) section

[Local files](#)

[Online read](#)

Assuming you have the file locally on the path `../data/`:

```
pts = geopandas.read_file("../data/madrid_abb.gpkg")
```

i Point geometries from columns

```
pts.info()
```

```
<class 'geopandas.geodataframe.GeoDataFrame'>
RangeIndex: 18399 entries, 0 to 18398
Data columns (total 16 columns):
#   Column                Non-Null Count  Dtype
---  -
0   price                 18399 non-null  object
1   price_usd             18399 non-null  float64
2   loglp_price_usd       18399 non-null  float64
3   accommodates          18399 non-null  int64
4   bathrooms             18399 non-null  object
5   bedrooms             18399 non-null  float64
6   beds                 18399 non-null  float64
7   neighbourhood         18399 non-null  object
8   room_type            18399 non-null  object
9   property_type         18399 non-null  object
10  WiFi                 18399 non-null  object
11  Coffee               18399 non-null  object
12  Gym                  18399 non-null  object
13  Parking              18399 non-null  object
14  km_to_retiro         18399 non-null  float64
15  geometry              18399 non-null  geometry
dtypes: float64(5), geometry(1), int64(1), object(9)
memory usage: 2.2+ MB
```

```
pts.head()
```

	price	price_usd	log1p_price_usd	accommodates	bathrooms	bedrooms
0	\$60.00	60.0	4.110874	2	1 shared bath	1.0
1	\$31.00	31.0	3.465736	1	1 bath	1.0
2	\$60.00	60.0	4.110874	6	2 baths	3.0
3	\$115.00	115.0	4.753590	4	1.5 baths	2.0
4	\$26.00	26.0	3.295837	1	1 private bath	1.0

Lines

[Local files](#) [Online read](#)

Assuming you have the file locally on the path `../data/`:

```
pts = geopandas.read_file("../data/arturo_streets.gpkg")
```

```
lines.info()
```

```
<class 'geopandas.geodataframe.GeoDataFrame'>
RangeIndex: 66499 entries, 0 to 66498
Data columns (total 7 columns):
#   Column      Non-Null Count  Dtype
---  ---
0   OGC_FID      66499 non-null  object
1   dm_id        66499 non-null  object
2   dist_barri   66483 non-null  object
3   X            66499 non-null  float64
4   Y            66499 non-null  float64
5   value        5465 non-null   float64
6   geometry     66499 non-null  geometry
dtypes: float64(3), geometry(1), object(3)
memory usage: 3.6+ MB
```

```
lines.loc[0, "geometry"]
```

_build/jupyter_execute/content/pages/02-Spatial_data_16_0.svg

Polygons

```
<IPython.display.GeoJSON object>
```

[Local files](#) [Online read](#)

Assuming you have the file locally on the path `../data/`:

```
polys = geopandas.read_file("../data/neighbourhoods.geojson")
```

```
polys.head()
```

	neighbourhood	neighbourhood_group	geometry
0	Palacio	Centro	MULTIPOLYGON (((-3.70584 40.42030, -3.70625 40...
1	Embajadores	Centro	MULTIPOLYGON (((-3.70384 40.41432, -3.70277 40...
2	Cortes	Centro	MULTIPOLYGON (((-3.69796 40.41929, -3.69645 40...
3	Justicia	Centro	MULTIPOLYGON (((-3.69546 40.41898, -3.69645 40...
4	Universidad	Centro	MULTIPOLYGON (((-3.70107 40.42134, -3.70155 40...

```
polys.query("neighbourhood_group == 'Retiro'")
```


	neighbourhood	neighbourhood_group	geometry
13	Pacífico	Retiro	MULTIPOLYGON (((-3.67015 40.40654, -3.67017 40...
14	Adelfas	Retiro	MULTIPOLYGON (((-3.67283 40.39468, -3.67343 40...
15	Estrella	Retiro	MULTIPOLYGON (((-3.66506 40.40647, -3.66512 40...
16	Ibiza	Retiro	MULTIPOLYGON (((-3.66916 40.41796, -3.66927 40...
17	Jerónimos	Retiro	MULTIPOLYGON (((-3.67874 40.40751, -3.67992 40...
18	Niño Jesús	Retiro	MULTIPOLYGON (((-3.66994 40.40850, -3.67012 40...

```
polys.neighbourhood_group.unique()
```

```
array(['Centro', 'Arganzuela', 'Retiro', 'Salamanca', 'Chamartín',
      'Moratalaz', 'Tetuán', 'Chamberí', 'Fuencarral - El Pardo',
      'Moncloa - Aravaca', 'Puente de Vallecas', 'Latina', 'Carabanchel',
      'Usera', 'Ciudad Lineal', 'Hortaleza', 'Villaverde',
      'Villa de Vallecas', 'Vicálvaro', 'San Blas - Canillejas',
      'Barajas'], dtype=object)
```

Surfaces

[Local files](#)

[Online read](#)

Assuming you have the file locally on the path `../data/`:




```
sat = xarray.open_rasterio("../data/madrid_scene_s2_10_tc.tif")
```

```
sat
```

xarray.DataArray (band: 3, y: 3681, x: 3129)

[34553547 values with dtype=uint8]

▼ Coordinates:

band	(band)	int64	1 2 3	
y	(y)	float64	4.499e+06 4.499e+06 ... 4.463e+06	
x	(x)	float64	4.248e+05 4.248e+05 ... 4.56e+05	

▼ Attributes:







```
transform: (10.0, 0.0, 424760.0, 0.0, -10.0, 4499370.0)
crs:       +init=epsg:32630
res:       (10.0, 10.0)
is_tiled:  0
nodatavals: (nan, nan, nan)
scales:    (1.0, 1.0, 1.0)
offsets:    (0.0, 0.0, 0.0)
AREA_OR_P... Area
```

```
sat.sel(band=1)
```

xarray.DataArray (y: 3681, x: 3129)

[11517849 values with dtype=uint8]

▼ Coordinates:

band	()	int64	1	 
y	(y)	float64	4.499e+06 4.499e+06 ... 4.463e+06	 
x	(x)	float64	4.248e+05 4.248e+05 ... 4.56e+05	 

▼ Attributes:




transform:	(10.0, 0.0, 424760.0, 0.0, -10.0, 4499370.0)
crs:	+init=epsg:32630
res:	(10.0, 10.0)
is_tiled:	0
nodatavals:	(nan, nan, nan)
scales:	(1.0, 1.0, 1.0)
offsets:	(0.0, 0.0, 0.0)
AREA_OR_P...	Area

```
sat.sel(  
    x=slice(430000, 440000), # x is ascending  
    y=slice(4480000, 4470000) # y is descending  
)
```

xarray.DataArray (band: 3, y: 1000, x: 1000)

[3000000 values with dtype=uint8]

▼ Coordinates:

band	(band)	int64	1 2 3	 
y	(y)	float64	4.48e+06 4.48e+06 ... 4.47e+06	 
x	(x)	float64	4.3e+05 4.3e+05 ... 4.4e+05 4.4e+05	 

▼ Attributes:

transform:	(10.0, 0.0, 424760.0, 0.0, -10.0, 4499370.0)
crs:	+init=epsg:32630
res:	(10.0, 10.0)
is_tiled:	0
nodatavals:	(nan, nan, nan)
scales:	(1.0, 1.0, 1.0)
offsets:	(0.0, 0.0, 0.0)
AREA_OR_P...	Area

Visualisation

```
polys.plot()
```

<AxesSubplot:>

_build/jupyter_execute/content/pages/02-Spatial_data_31_1.png

```
ax = lines.plot(linewidth=0.1, color="black")  
contextily.add_basemap(ax, crs=lines.crs)
```

_build/jupyter_execute/content/pages/02-Spatial_data_32_0.png

```
ax = pts.plot(color="red", figsize=(12, 12), markersize=0.1)
contextily.add_basemap(
    ax,
    crs = pts.crs,
    source = contextily.providers.CartoDB.DarkMatter
);
```

See more basemap options [here](#).

_build/jupyter_execute/content/pages/02-Spatial_data_34_0.png

```
sat.plot.imshow(figsize=(12, 12))
```

```
<matplotlib.image.AxesImage at 0x7fb126f3fa10>
```

_build/jupyter_execute/content/pages/02-Spatial_data_35_1.png

```
f, ax = plt.subplots(1, figsize=(12, 12))
sat.plot.imshow(ax=ax)
contextily.add_basemap(
    ax,
    crs=sat.rio.crs,
    source=contextily.providers.Stamen.TonerLabels,
    zoom=11
);
```

IMPORTANT

You will need version 1.1.0 of [contextily](#) to use label layers. Install it with:

```
pip install \
-U --no-deps \
contextily
```

```
/opt/conda/lib/python3.7/site-packages/pyproj/crs/crs.py:280: FutureWarning: '+init=
<authority>:<code>' syntax is deprecated. '<authority>:<code>' is the preferred
initialization method. When making the change, be mindful of axis order changes:
https://pyproj4.github.io/pyproj/stable/gotchas.html#axis-order-changes-in-proj-6
projstring = _prepare_from_string(projparams)
```

_build/jupyter_execute/content/pages/02-Spatial_data_37_1.png

Spatial operations

(Re-)Projections

```
pts.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
```

```
sat.rio.crs
```

```
CRS.from_epsg(32630)
```

```
pts.to_crs(sat.rio.crs).crs
```

```
<Projected CRS: EPSG:32630>
Name: WGS 84 / UTM zone 30N
Axis Info [cartesian]:
- [east]: Easting (metre)
- [north]: Northing (metre)
Area of Use:
- undefined
Coordinate Operation:
- name: UTM zone 30N
- method: Transverse Mercator
Datum: World Geodetic System 1984
- Ellipsoid: WGS 84
- Prime Meridian: Greenwich
```

```
sat.rio.reproject(pts.crs).rio.crs
```

```
CRS.from_epsg(4326)
```

```
# All into Web Mercator (EPSG:3857)
f, ax = plt.subplots(1, figsize=(12, 12))
## Satellite image
sat.rio.reproject(
    "EPSG:3857"
).plot.imshow(
    ax=ax
)
## Neighbourhoods
polys.to_crs(eps=3857).plot(
    linewidth=2,
    edgecolor="xkcd:lime",
    facecolor="none",
    ax=ax
)
## Labels
contextily.add_basemap( # No need to reproject
    ax,
    source=contextily.providers.Stamen.TonerLabels,
);
```

_build/jupyter_execute/content/pages/02-Spatial_data_44_0.png

Centroids

```
polys.centroid
```

```
/opt/conda/lib/python3.7/site-packages/ipykernel_launcher.py:1: UserWarning:
Geometry is in a geographic CRS. Results from 'centroid' are likely incorrect. Use
'GeoSeries.to_crs()' to re-project geometries to a projected CRS before this
operation.
```

```
"""Entry point for launching an IPython kernel.
```

```
0    POINT (-3.71398 40.41543)
1    POINT (-3.70237 40.40925)
2    POINT (-3.69674 40.41485)
3    POINT (-3.69657 40.42367)
4    POINT (-3.70698 40.42568)
...
123  POINT (-3.59135 40.45656)
124  POINT (-3.59723 40.48441)
125  POINT (-3.55847 40.47613)
126  POINT (-3.57889 40.47471)
127  POINT (-3.60718 40.46415)
Length: 128, dtype: geometry
```

```
lines.centroid
```

Note the warning that geometric operations with non-project CRS object result in biases.

```

0      POINT (444133.737 4482808.936)
1      POINT (444192.064 4482878.034)
2      POINT (444134.563 4482885.414)
3      POINT (445612.661 4479335.686)
4      POINT (445606.311 4479354.437)
...
66494  POINT (451980.378 4478407.920)
66495  POINT (436975.438 4473143.749)
66496  POINT (442218.600 4478415.561)
66497  POINT (442213.869 4478346.700)
66498  POINT (442233.760 4478278.748)
Length: 66499, dtype: geometry

```

```

ax = polys.plot(color="purple")
polys.centroid.plot(
    ax=ax, color="lime", markersize=1
)

```

```

/opt/conda/lib/python3.7/site-packages/ipykernel_launcher.py:2: UserWarning:
Geometry is in a geographic CRS. Results from 'centroid' are likely incorrect. Use
'GeoSeries.to_crs()' to re-project geometries to a projected CRS before this
operation.

```

<AxesSubplot:>

_build/jupyter_execute/content/pages/02-Spatial_data_49_2.png

Spatial joins

```

sj = geopandas.sjoin(
    lines,
    polys.to_crs(lines.crs)
)

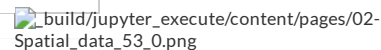
```

```
sj.info()
```

```

<class 'geopandas.geodataframe.GeoDataFrame'>
Int64Index: 69420 entries, 0 to 66438
Data columns (total 10 columns):
#   Column                Non-Null Count  Dtype
---  -
0   OGC_FID                69420 non-null  object
1   dm_id                  69420 non-null  object
2   dist_barri             69414 non-null  object
3   X                      69420 non-null  float64
4   Y                      69420 non-null  float64
5   value                  5769 non-null   float64
6   geometry               69420 non-null  geometry
7   index_right            69420 non-null  int64
8   neighbourhood          69420 non-null  object
9   neighbourhood_group    69420 non-null  object
dtypes: float64(3), geometry(1), int64(1), object(5)
memory usage: 5.8+ MB

```

_build/jupyter_execute/content/pages/02-Spatial_data_53_0.png

More information about spatial joins in [geopandas](#) is available on its [documentation page](#)

Areas

```

areas = polys.to_crs(
    epsg=25830
).area * 1e-6 # Km2
areas.head()

```

```

0    1.471037
1    1.033253
2    0.592049
3    0.742031
4    0.947616
dtype: float64

```

Distances

```
cemfi = geopandas.tools.geocode(
    "Calle Casado del Alisal, 5, Madrid"
).to_crs(epsg=25830)
cemfi
```

	geometry	address
0	POINT (441473.624 4473943.520)	Calle de Casado del Alisal 5, 28014 Madrid, Sp...

```
polys.to_crs(
    cemfi.crs
).distance(
    cemfi.geometry
)
```


```
/opt/conda/lib/python3.7/site-packages/geopandas/base.py:39: UserWarning: The
indices of the two GeoSeries are different.
  warn("The indices of the two GeoSeries are different.")
```

```
0      1487.894214
1           NaN
2           NaN
3           NaN
4           NaN
...
123          NaN
124          NaN
125          NaN
126          NaN
127          NaN
Length: 128, dtype: float64
```

```
d2cemfi = polys.to_crs(
    cemfi.crs
).distance(
    cemfi.geometry[0] # NO index
)
d2cemfi.head()
```

```
0      1487.894214
1      567.196279
2      275.166923
3      645.807884
4      1191.537001
dtype: float64
```

□ Next steps

_build/jupyter_execute/content/pages/02-Spatial_data_61_0.png

If you are interested in following up on some of the topics explored in this block, the following pointers might be useful:

- Although we have seen here **geopandas** only, all non-geographic operations on geo-tables are really thanks to **pandas**, the workhorse for tabular data in Python. Their [official documentation](#) is an excellent first stop. If you prefer a book, McKinney (2012) [[McK12](#)] is a great one.
- For more detail on geographic operations on geo-tables, the [Geopandas official documentation](#) is a great place to continue the journey.
- Surfaces, as covered here, are really an example of multi-dimensional labelled arrays. The library we use, **xarray** represents the cutting edge for working with these data structures in Python, and [their documentation](#) is a great place to wrap your head around how data of this type can be manipulated. For geographic extensions (CRS handling, reprojections, etc.), we have used **rioxarray** under the hood, and [its documentation](#) is also well worth checking.

Geovisualisation

□ Ahead of time...

This block is all about visualising statistical data on top of a geography. Although this task looks simple, there are a few technical and conceptual building blocks that it helps to understand before we try to make our own maps. Aim to complete the following readings by the time we get our hands on the keyboard:

- [Block D](#) of the GDS course [[AB19](#)], which provides an introduction to choropleths (statistical maps)
- [Chapter 5](#) of the GDS Book [[RABWng](#)], discussing choropleths in more detail

□ Hands-on coding

```
import geopandas
import xarray, rioxarray
import contextily
import seaborn as sns
from pysal.viz import mapclassify as mc
from legendgram import legendgram
import matplotlib.pyplot as plt
import palettable.matplotlib as palmpl
```

[Local files](#)

[Online read](#)

Data

If you want to read more about the data sources behind this dataset, head to the [Datasets](#) section

Assuming you have the file locally on the path `../data/`:

```
db = geopandas.read_file("../data/cambodia_regional.gpkg")
```

```
db.info()
```

```
<class 'geopandas.geodataframe.GeoDataFrame'>
RangeIndex: 198 entries, 0 to 197
Data columns (total 6 columns):
#   Column      Non-Null Count  Dtype  
---  -
0   adm2_name    198 non-null    object  
1   adm2_altnm   122 non-null    object  
2   motor_mean   198 non-null    float64  
3   walk_mean    198 non-null    float64  
4   no2_mean     198 non-null    float64  
5   geometry     198 non-null    geometry
dtypes: float64(3), geometry(1), object(2)
memory usage: 9.4+ KB
```

build/jupyter_execute/content/pages/03-Geovisualisation_7_0.png

We will use the average measurement of [nitrogen dioxide](#) (`no2_mean`) by region throughout the block.

To make visualisation a bit easier below, we create an additional column with values rescaled:

```
db["no2_viz"] = db["no2_mean"] * 1e5
```

This way, numbers are larger and will fit more easily on legends:

```
db[["no2_mean", "no2_viz"]].describe()
```

	no2_mean	no2_viz
count	198.000000	198.000000
mean	0.000032	3.236567
std	0.000017	1.743538
min	0.000014	1.377641
25%	0.000024	2.427438
50%	0.000029	2.922031
75%	0.000034	3.390426
max	0.000123	12.323324

Choropleths

_build/jupyter_execute/content/pages/03-Geovisualisation_14_0.png

A classification problem

```
db["no2_viz"].unique().shape
```

```
(198,)
```

```
sns.displot(
    db, x="no2_viz", kde=True, aspect=2
);
```

_build/jupyter_execute/content/pages/03-Geovisualisation_17_0.png

! Attention

To build an intuition behind each classification algorithm more easily, we create a helper method (`plot_classi`) that generates a visualisation of a given classification.

Toggle the cell below if you are interested in the code behind it.

- Equal intervals

```
classi = mc.EqualInterval(db["no2_viz"], k=7)
classi
```

```
EqualInterval
  Interval      Count
-----
[ 1.38,  2.94] |    103
( 2.94,  4.50] |     80
( 4.50,  6.07] |      6
( 6.07,  7.63] |      1
( 7.63,  9.20] |      3
( 9.20, 10.76] |      0
(10.76, 12.32] |      5
```



_build/jupyter_execute/content/pages/03-Geovisualisation_22_0.png

- Quantiles

```
classi = mc.Quantiles(db["no2_viz"], k=7)
classi
```

```
Quantiles
  Interval      Count
-----
[ 1.38,  2.24] |     29
( 2.24,  2.50] |     28
( 2.50,  2.76] |     28
( 2.76,  3.02] |     28
( 3.02,  3.35] |     28
( 3.35,  3.76] |     28
( 3.76, 12.32] |     29
```



_build/jupyter_execute/content/pages/03-Geovisualisation_25_0.png

- Fisher-Jenks

```
classi = mc.FisherJenks(db["no2_viz"], k=7)
classi
```

```
FisherJenks
  Interval      Count
-----
[ 1.38,  2.06] |     20
( 2.06,  2.69] |     58
( 2.69,  3.30] |     62
( 3.30,  4.19] |     42
( 4.19,  5.64] |      7
( 5.64,  9.19] |      4
( 9.19, 12.32] |      5
```



_build/jupyter_execute/content/pages/03-Geovisualisation_28_0.png

- Fisher-Jenks

```
classi = mc.FisherJenks(db["no2_viz"], k=7)
classi
```

FisherJenks

Interval	Count
[1.38, 2.06]	20
(2.06, 2.69]	58
(2.69, 3.30]	62
(3.30, 4.19]	42
(4.19, 5.64]	7
(5.64, 9.19]	4
(9.19, 12.32]	5

_build/jupyter_execute/content/pages/03-Geovisualisation_31_0.png

Now let's dig into the internals of `classi`:

`classi`

FisherJenks

Interval	Count
[1.38, 2.06]	20
(2.06, 2.69]	58
(2.69, 3.30]	62
(3.30, 4.19]	42
(4.19, 5.64]	7
(5.64, 9.19]	4
(9.19, 12.32]	5

`classi.k`

7

`classi.bins`

```
array([ 2.05617382,  2.6925931 ,  3.30281182,  4.19124954,  5.63804861,
        9.19190206, 12.32332434])
```

`classi.yb`

```
array([2, 3, 3, 1, 1, 2, 1, 1, 1, 0, 0, 3, 2, 1, 1, 1, 3, 1, 1, 1, 2, 0,
       0, 4, 2, 1, 3, 1, 0, 0, 0, 1, 2, 2, 6, 5, 4, 2, 1, 3, 2, 3, 2, 1,
       2, 3, 2, 3, 1, 1, 3, 1, 2, 3, 3, 1, 3, 3, 1, 0, 1, 1, 3, 2, 0, 0,
       2, 1, 0, 0, 0, 2, 0, 1, 3, 3, 3, 2, 3, 2, 3, 1, 2, 3, 1, 1, 1, 1,
       2, 1, 2, 2, 1, 2, 2, 2, 1, 3, 2, 3, 2, 2, 2, 1, 2, 3, 3, 2, 0, 3,
       1, 0, 1, 2, 1, 1, 2, 1, 2, 6, 5, 6, 2, 2, 3, 6, 3, 4, 3, 4, 2, 3,
       0, 2, 5, 6, 4, 5, 2, 2, 2, 1, 1, 1, 2, 1, 2, 3, 3, 2, 2, 2, 3, 2,
       1, 1, 3, 4, 2, 1, 3, 1, 2, 3, 4, 0, 1, 1, 2, 1, 2, 2, 2, 2, 1, 2,
       2, 2, 0, 0, 1, 2, 3, 3, 3, 3, 3, 2, 1, 2, 1, 1, 1, 2, 2, 1, 3, 1])
```

How many colors?



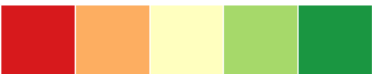
Attention

The code used to generate this figure uses more advanced features than planned for this course.

If you want to inspect it, toggle the cell below.



Using the *right* color

-  **Categories, non-ordered**
-  Graduated, **sequential**
-  Graduated, **divergent**

For a safe choice, make sure to visit [ColorBrewer](#)

Choropleths on Geo-Tables

How can we create classifications from data on geo-tables? Two ways:

- Directly within **plot** (only for some algorithms)

```
db.plot(
    "no2_viz", scheme="quantiles", k=7, legend=True
);
```



- Manually attaching the data (for any algorithm)

```
classi = mc.Quantiles(db["no2_viz"], k=7)
db.assign(
    classes=classi.yb
).plot("classes");
```

See [this tutorial](#) for more details on fine tuning choropleths manually



Legendgrams:

```
f, ax = plt.subplots(figsize=(9, 9))
classi = mc.Quantiles(db["no2_viz"], k=7)
db.assign(
    classes=classi.yb
).plot("classes", ax=ax)
legendgram(
    f,                # Figure object
    ax,               # Axis object of the map
    db["no2_viz"],    # Values for the histogram
    classi.bins,      # Bin boundaries
    pal=palettable.Viridis_7, # color palette (as palettable object)
    legend_size=(.5,.2), # legend size in fractions of the axis
    loc = 'lower right', # matplotlib-style legend locations
    clip = (2,10)      # clip the displayed range of the histogram
)
ax.set_axis_off();
```



Surface visualisation

[Local files](#) [Online read](#)

Assuming you have the file locally on the path `./data/`:

Data

If you want to read more about the data sources behind this dataset, head to the [Datasets](#) section

```
grid = xarray.open_rasterio(
    "../data/cambodia_s5_no2.tif"
).sel(band=1)
```

- (Implicit) continuous equal interval

```
grid.where(
    grid != grid.rio.nodata
).plot(cmap="viridis");
```

_build/jupyter_execute/content/pages/03-Geovisualisation_53_0.png

```
grid.where(
    grid != grid.rio.nodata
).plot(cmap="viridis", robust=True);
```

_build/jupyter_execute/content/pages/03-Geovisualisation_54_0.png

- Discrete equal interval

```
grid.where(
    grid != grid.rio.nodata
).plot(cmap="viridis", levels=7)
```

```
/opt/conda/lib/python3.7/site-packages/xarray/plot/plot.py:970:
MatplotlibDeprecationWarning: Passing parameters norm and vmin/vmax simultaneously
is deprecated since 3.3 and will become an error two minor releases later. Please
pass vmin/vmax directly to the norm when creating it.
    primitive = ax.pcolormesh(x, y, z, **kwargs)
```

```
<matplotlib.collections.QuadMesh at 0x7f2eb886d450>
```

_build/jupyter_execute/content/pages/03-Geovisualisation_56_2.png

- Combining with `mapclassify`

```
grid_nona = grid.where(
    grid != grid.rio.nodata
)

classi = mc.Quantiles(
    grid_nona.to_series().dropna(), k=7
)

grid_nona.plot(
    cmap="viridis", levels=classi.bins
)
plt.title(classi.name);
```

```
/opt/conda/lib/python3.7/site-packages/xarray/plot/plot.py:970:
MatplotlibDeprecationWarning: Passing parameters norm and vmin/vmax simultaneously
is deprecated since 3.3 and will become an error two minor releases later. Please
pass vmin/vmax directly to the norm when creating it.
    primitive = ax.pcolormesh(x, y, z, **kwargs)
```

_build/jupyter_execute/content/pages/03-Geovisualisation_58_1.png

```
/opt/conda/lib/python3.7/site-packages/xarray/plot/plot.py:970:
MatplotlibDeprecationWarning: Passing parameters norm and vmin/vmax simultaneously
is deprecated since 3.3 and will become an error two minor releases later. Please
pass vmin/vmax directly to the norm when creating it.
    primitive = ax.pcolormesh(x, y, z, **kwargs)
```

_build/jupyter_execute/content/pages/03-Geovisualisation_59_1.png

```
/opt/conda/lib/python3.7/site-packages/xarray/plot/plot.py:970:
MatplotlibDeprecationWarning: Passing parameters norm and vmin/vmax simultaneously
is deprecated since 3.3 and will become an error two minor releases later. Please
pass vmin/vmax directly to the norm when creating it.
    primitive = ax.pcolormesh(x, y, z, **kwargs)
```

_build/jupyter_execute/content/pages/03-Geovisualisation_60_1.png

```
/opt/conda/lib/python3.7/site-packages/xarray/plot/plot.py:970:
MatplotlibDeprecationWarning: Passing parameters norm and vmin/vmax simultaneously
is deprecated since 3.3 and will become an error two minor releases later. Please
pass vmin/vmax directly to the norm when creating it.
    primitive = ax.pcolormesh(x, y, z, **kwargs)
```

_build/jupyter_execute/content/pages/03-Geovisualisation_61_1.png

□ Next steps

If you are interested in statistical maps based on classification, here are two recommendations to check out next:

- On the technical side, the [documentation for mapclassify](#) (including its [tutorials](#)) provides more detail and illustrates more classification algorithms than those reviewed in this block
- On a more conceptual note, Cynthia Brewer’s “Designing better maps” [[Bre15](#)] is an excellent blueprint for good map making.

Spatial Feature Engineering (I)

Map Matching

□ Ahead of time...

Feature Engineering is a common term in machine learning that refers to the processes and transformations involved in turning data from the state in which the modeller access them into what is then fed to a model. This can take several forms, from standardisation of the input data, to the derivation of numeric scores that better describe aspects (*features*) of the data we are using.

Spatial Feature Engineering refers to operations we can use to derive “views” or summaries of our data that we can use in models, *using space* as the key medium to create them.

There is only one reading to complete for this block, [Chapter 12](#) of the GDS Book [[RABWng](#)]. The first block of Spatial Feature Engineering in this course loosely follows the first part of the chapter ([Map Matching](#)), so focus on this first sections for the block.

□ Hands-on coding

```
import pandas
import geopandas
import xarray, rioxarray
import contextily
import numpy as np
import matplotlib.pyplot as plt
```

[Local files](#)

[Online read](#)

Data

If you want to read more about the data sources behind this dataset, head to the [Datasets](#) section

Assuming you have the file locally on the path `../data/`:

```
regions = geopandas.read_file("../data/cambodia_regional.gpkg")
cities = geopandas.read_file("../data/cambodian_cities.geojson")
pollution = xarray.open_rasterio(
    "../data/cambodia_s5_no2.tif"
).sel(band=1)
friction = xarray.open_rasterio(
    "../data/cambodia_2020_motorized_friction_surface.tif"
).sel(band=1)
```

Check both geo-tables and the surface are in the same CRS:

```
regions.crs.to_epsg() == \
cities.crs.to_epsg() == \
pollution.rio.crs.to_epsg()
```

True

Polygons to points

In which region is a city?

```
sj = geopandas.sjoin(
    cities,
    regions
)
```

```
# City name | Region name
sj[["UC_NM_MN", "adm2_name"]]
```

	UC_NM_MN	adm2_name
0	Sampov Lun	Sampov Lun
1	Khum Pech Chenda	Phnum Proek
2	Poipet	Paoy Paet
3	Sisophon	Serei Saophoan
4	Battambang	Battambang
5	Siem Reap	Siem Reap
6	Sihanoukville	Preah Sihanouk
7	N/A	Trapeang Prasat
8	Kampong Chhnang	Kampong Chhnang
9	Phnom Penh	Tuol Kouk
10	Kampong Cham	Kampong Cham

Points to polygons

If we were after the number of cities per region, it is a similar approach, with a (groupby) twist at the end:

```
regions.set_index(
    "adm2_name"
).assign(
    city_count=sj.groupby("adm2_name").size()
).info()
```

```
<class 'geopandas.geodataframe.GeoDataFrame'>
Index: 198 entries, Mongkol Borei to Administrative unit not available
Data columns (total 6 columns):
#   Column          Non-Null Count  Dtype
---  ---
0   adm2_altnm      122 non-null    object
1   motor_mean      198 non-null    float64
2   walk_mean       198 non-null    float64
3   no2_mean        198 non-null    float64
4   geometry        198 non-null    geometry
5   city_count      11 non-null     float64
dtypes: float64(4), geometry(1), object(1)
memory usage: 10.8+ KB
```

Note

1. We `set_index` to align both tables
2. We `assign` to create a new column

If you want no missing values, you can `fillna(0)` since you know missing data are zeros

Surface to points

Consider attaching to each city in `cities` the pollution level, as expressed in `pollution`.

_build/jupyter_execute/content/pages/04-Spatial_feature_eng_i_19_0.png

```
from rasterstats import point_query

city_pollution = point_query(
    cities,
    pollution.values,
    affine=pollution.rio.transform(),
    nodata=pollution.rio.nodata
)
city_pollution
```

```
[3.9397064813333136e-05,
 3.4949825609644426e-05,
 3.825255125820345e-05,
 4.103826573585785e-05,
 3.067677208474005e-05,
 5.108273256655399e-05,
 2.2592785882580366e-05,
 4.050414400882722e-05,
 2.4383652926989897e-05,
 0.0001285838935209779,
 3.258245740282522e-05]
```

The code for generating this figure is a bit more advanced as it fiddles with text, but if you want to explore it you can toggle it on

And we can map these on the city locations:

```
ax = cities.assign(
    pollution=city_pollution
).plot(
    "pollution",
    cmap="YlOrRd",
    legend=True
)

contextily.add_basemap(
    ax=ax, crs=cities.crs,
);
```

_build/jupyter_execute/content/pages/04-Spatial_feature_eng_i_22_0.png

Surface to polygons

Instead of transferring to points, we want to aggregate all the information in a surface that falls *within* a polygon.

For this case, we will use the motorised friction surface. The question we are asking thus is: *what is the average degree of friction of each region?* Or, in other words: *what regions are harder to get through with motorised transport?*



Again, we can rely on **rasterstats**:

```
from rasterstats import zonal_stats
regional_pollution = pandas.DataFrame(
    zonal_stats(
        regions,
        pollution.values,
        affine=pollution.rio.transform(),
        nodata=pollution.rio.nodata
    ),
    index=regions.index
)
regional_pollution.head()
```

The output is returned from **zonal_stats** as a list of dicts. To make it more manageable, we convert it into a **pandas.DataFrame**.

	min	max	mean	count
0	0.000016	0.000045	0.000029	50
1	0.000019	0.000052	0.000034	72
2	0.000022	0.000049	0.000037	19
3	0.000017	0.000037	0.000026	41
4	0.000018	0.000033	0.000025	3

This can then also be mapped onto the polygon geography:



Surface to surface

If we want to align the **pollution** surface with that of **friction**, we need to resample them to make them “fit on the same frame”.

```
pollution.shape
```

```
(138, 152)
```

```
friction.shape
```

```
(574, 636)
```

This involves either moving one surface to the frame of the other one, or both into an entirely new one. For the sake of the illustration, we will do the latter and select a frame that is 300 by 400 pixels. Note this involves stretching (upsampling) **pollution**, while compressing (downsampling) **friction**. For this, we use **datashader**.

```
import datashader as ds
canvas = ds.Canvas(plot_height=400, plot_width=300)
tgt_pollution = canvas.raster(pollution)
tgt_friction = canvas.raster(friction)
```

```
tgt_pollution.shape
```



```
(400, 300)
```

```
tgt_pollution.shape == tgt_friction.shape
```

```
True
```

! Attention

The following methods involve modelling and are thus more sophisticated. Take these as a conceptual introduction with an empirical illustration, but keep in mind there are extensive literatures on each of them and these cover some of the simplest cases.

Points to points

For this example, we will assume that, instead of a surface with pollution values, we only have available a sample of points and we would like to obtain estimates for other locations.

See [this section](#) of Chapter 12 of the GDS Book [[RABWng](#)] for more details on the technique

For that we will first generate 100 random points within the extent of `pollution` which we will take as the location of our measurement stations:

```
rs = np.random.RandomState(
    np.random.MT19937(np.random.SeedSequence(123456789))
)

bb = pollution.rio.bounds()
station_xs = np.random.uniform(bb[0], bb[2], 100)
station_ys = np.random.uniform(bb[1], bb[3], 100)
stations = geopandas.GeoSeries(
    geopandas.points_from_xy(station_xs, station_ys),
    crs=pollution.rio.crs
)
```

i Note

The code in this cell contains bits that are a bit more advanced, do not despair if not everything makes sense!

Our station values come from the `pollution` surface, but we assume we do not have access to the latter, and we would like to obtain estimates for the location of the cities:



We will need the location and the pollution measurements for every station as separate arrays. Before we do that, since we will be calculating distances, we convert our coordinates to `asystem` expressed in metres.

```
stations_mt = stations.to_crs(epsg=5726)
station_xys = np.array(
    [stations_mt.geometry.x, stations_mt.geometry.y]
).T
```

We also need to extract the pollution measurements for each station location:

```
station_measurements = np.array(
    point_query(
        stations,
        pollution.values,
        affine=pollution.rio.transform(),
        nodata=pollution.rio.nodata
    )
)
```

And finally, we will also need the locations of each city expressed in the same coordination system:

```
cities_mt = cities.to_crs(epsg=5726)
city_xys = np.array(
    [cities_mt.geometry.x, cities_mt.geometry.y]
).T
```

For this illustration, we will use a k -nearest neighbors regression that estimates the value for each target point (**cities** in our case) as the average weighted by distance of its k nearest neighbors. In this illustration we will use $k = 10$.

```
from sklearn.neighbors import KNeighborsRegressor

model = KNeighborsRegressor(
    n_neighbors=10, weights="distance"
).fit(station_xys, station_measurements)
```

Note how **sklearn** relies only on array data structures, so we first had to express all the required information in that format

Once we have trained the model, we can use it to obtain predictions for each city location:

```
predictions = model.predict(city_xys)
```

Points to surface

Imagine we do not have a surface like **pollution** but we needed it. In this context, if you have measurements from some locations, such as in **stations**, we can use the approach reviewed above to generate a surface. The trick to do this is to realise that we can generate a *uniform* grid of target locations that we can then express as a surface.

We will set as our target locations those of the pixels in the target surface we have seen [above](#):

```
tgt_friction_mt = tgt_friction.rio.set_crs(
    "EPSG:4326"
).rio.reproject("EPSG:5726")
```

```
xys = tgt_friction_mt.coords.to_index()
```

To obtain pollution estimates at each location, we can **predict** with **model**:

```
predictions_grid = model.predict(
    np.array(xys.to_list())
)
```

And with these at hand, we can convert them into a surface:

```
predictions_series = pandas.Series(
    predictions_grid,
    index=xys
)
predictions_surface = xarray.DataArray().from_series(
    predictions_series
)
```

```
f, axs = plt.subplots(1, 2, figsize=(16, 6))

tgt_pollution.where(tgt_pollution>0).plot(ax=axs[0])
predictions_surface.plot(ax=axs[1])

plt.show()
```

_build/jupyter_execute/content/pages/04-Spatial_feature_eng_i_64_0.png

Tip

If performance is a concern, you can build the array of XY coordinates directly from the original object:

```
predictions_grid =
model.predict(
    np.array([

tgt_friction.coords[
    "x"].values,

tgt_friction.coords[
    "y"].values,
    ])
)
```

Polygons to polygons

Next steps

If you are interested in learning more about spatial feature engineering through map matching, the following pointers might be useful to delve deeper into specific types of “data transfer”:

- The [datashader](#) library is a great option to transfer geo-tables into surfaces, providing tooling to perform these operations in a highly efficient and performant way.
- When aggregating surfaces into geo-tables, the library [rasterstats](#) contains most if not all of the machinery you will need.
- For transfers from polygon to polygon geographies, [tobler](#) is your friend. Its official documentation contains examples for different use cases.

Spatial Feature Engineering (II)

Map Synthesis

□ Ahead of time...

In this second part of Spatial Feature Engineering, we turn to Map Synthesis. There is only one reading to complete for this block, [Chapter 12](#) of the GDS Book [[RABWng](#)]. This block of Spatial Feature Engineering in this course loosely follows the second part of the chapter ([Map Synthesis](#)).

□ Hands-on coding

“Dataset enhancement through space”:

- [] Distance buffer counts (within and between datasets)
- [] Ring buffers (within and between datasets)
- [] Clustering as map synthesis (example with DBSCAN and LISAs)

□ Next steps

Spatial Networks (I)

Spatial Networks (II)

Transport costs

Visual challenges and opportunities

Datasets

This section covers the datasets required to run the course interactively. For archival reasons, all of those listed here have been mirrored in the repository for this course so, if you have [downloaded the course](#), you already have a local copy of them.




Madrid

Airbnb properties

Source

This dataset has been sourced from the course "[Spatial Modelling for Data Scientists](#)". The file imported here corresponds to the [v0.1.0](#) version.

This dataset contains a pre-processed set of properties advertised on the Airbnb website within the region of Madrid (Spain), together with house characteristics.

-  Data file [madrid_abb.gpkg](#)
-  Code used to generate the file [\[URL\]](#).
-  Further information [\[URL\]](#).



This dataset is licensed under a [CC0 1.0 Universal Public Domain Dedication](#).

Airbnb neighbourhoods

Source

This dataset has been directly sourced from the website [Inside Airbnb](#). The file was imported on February 10th 2021.

This dataset contains neighbourhood boundaries for the city of Madrid, as provided by Inside Airbnb.




-  Data file [neighbourhoods.geojson](#)
-  Further information [\[URL\]](#).



This dataset is licensed under a [CC0 1.0 Universal Public Domain Dedication](#).

Arturo

This dataset contains the street layout of Madrid as well as scores of habitability, where available, associated with street segments. The data originate from the [Arturo Project](#), by [300,000Km/s](#), and the available file here is a slimmed down version of their official [street layout](#) distributed by the project.

-  Data file [arturo_streets.gpkg](#)
-  Code used to generate the file [\[Page\]](#).
-  Further information [\[URL\]](#).






This dataset is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](#).

Sentinel 2 - 120m mosaic

This dataset contains four scenes for the region of Madrid (Spain) extracted from the [Digital Twin Sandbox Sentinel-2 collection](#), by the SentinelHub. Each scene corresponds to the following dates in 2019:

- January 1st
- April 1st
- July 10th
- November 17th

Each scene includes red, green, blue and near-infrared bands.




-  Data files ([Jan 1st](#), [Apr 1st](#), [Jul 10th](#), [Nov 27th](#))
-  Code used to generate the file [\[Page\]](#).
-  Further information [\[URL\]](#).



This dataset is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](#).

Sentinel 2 - 10m GHS composite

This dataset contains a scene for the region of Madrid (Spain) extracted from the [GHS Composite S2](#), by the European Commission.

-  Data file [madrid_scene_s2_10_tc.tif](#)
-  Code used to generate the file [\[Page\]](#).
-  Further information [\[URL\]](#).






This dataset is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](#).

Cambodia

Pollution

Surface with NO_2 measurements (tropospheric column) information attached from Sentinel 5.




-  Data file [cambodia_s5_no2.tif](#)
-  Code used to generate the file [\[Page\]](#).
-  Further information [\[URL\]](#).

Friction surfaces

This dataset is an extraction of the following two data products by Weiss et al. (2020) [\[WNVR+20\]](#) and distributed through the [Malaria Atlas Project](#):

- Global friction surface enumerating land-based travel walking-only speed without access to motorized transport for a nominal year 2019 (Minutes required to travel one metre)
- Global friction surface enumerating land-based travel speed with access to motorized transport for a nominal year 2019 (Minutes required to travel one metre)

Each is provided on a separate file.

-  Data files ([Motorized](#) and [Walking](#))
-  Code used to generate the file [\[Page\]](#).
-  Further information [\[URL\]](#).

Regional aggregates

Source

This dataset relies on boundaries from the [Humanitarian Data Exchange](#). The file is provided by the World Food Programme through the Humanitarian Data Exchange and was accessed on February 15th 2021.

[Pollution](#) and [friction](#) aggregated at Level 2 (municipality) administrative boundaries for Cambodia.

- Data file [cambodia_regional.gpkg](#)
- Code used to generate the file [\[Page\]](#).



This dataset is licensed under a [Creative Commons Attribution 4.0 International License](#).

Cambodian cities

Extract from the Urban Centre Database (UCDB), version 1.2, of the centroid for Cambodian cities.

- Data file [cambodian_cities.geojson](#)
- Code used to generate the file [\[Page\]](#).
- Further information [\[URL\]](#).



This dataset is licensed under a [Creative Commons Attribution 4.0 International License](#).

Further Resources

If this course is successful, it will leave you wanting to learn more about using Python for (Geographic) Data Science. See below a few resources that are good “next steps”.

Courses

- The “Automating GIS processes”, by Vuokko Heikinheimo and Henrikki Tenkanen is a great overview of GIS with a modern Python stack:

<https://autogis-site.readthedocs.io/>

- The “GDS Course” by Dani Arribas-Bel [\[AB19\]](#) is an introductory level overview of Geographic Data Science, including notebooks, slides and video clips.

https://darribas.org/gds_course

Books

- “Python for Geographic Data Analysis”, by Henrikki Tenkanen, Vuokko Heikinheimo and David Whipp:

<https://pythongis.org/>

- “Geographic Data Science in Python”, by Sergio J. Rey, Dani Arribas-Bel and Levi J. Wolf:

<https://geographicdata.science>

Bibliography

[AB19]

Dani Arribas-Bel. A course on geographic data science. *The Journal of Open Source Education*, 2019. doi:<https://doi.org/10.21105/jose.00042>.

[BAB20]

[Bre15]

Geoff Boeing and Dani Arribas-Bel. Gis and computational notebooks. In John P. Wilson, editor, *The Geographic Information Science & Technology Body of Knowledge*. UCGIS, 2020.

Cynthia Brewer. *Designing better Maps: A Guide for GIS users*. ESRI press, 2015. [McK12]

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

[RABWng]

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

[RBZ+19]

Adam Rule, Amanda Birmingham, Cristal Zuniga, Ilkay Altintas, Shih-Cheng Huang, Rob Knight, Niema Moshiri, Mai H Nguyen, Sara Brin Rosenthal, Fernando Pérez, and others. Ten simple rules for writing and sharing computational analyses in jupyter notebooks. *PLoS Comput Biol*, 2019. [doi:https://doi.org/10.1371/journal.pcbi.1007007](https://doi.org/10.1371/journal.pcbi.1007007).

[SAB19]

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

[WNVR+20]

DJ Weiss, A Nelson, CA Vargas-Ruiz, K Gligorić, S Bavadekar, E Gabrilovich, A Bertozzi-Villa, J Rozier, HS Gibson, T Shekel, and others. Global maps of travel time to healthcare facilities. *Nature Medicine*, 26(12):1835–1838, 2020.

By Dani Arribas-Bel & Diego Puga



Data Science Studio by [Dani Arribas-Bel](#) and [Diego Puga](#) is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](#).