spatial wdd3333

Gabriele Filomena and Elisabetta Pietrostefani

2024-08-12

Table of contents

Welcome

This is the website for "Web Mapping and Geovisualisation" (module **ENVS456**) at the University of Liverpool. This course is designed and delivered by Dr. Gabriele Filomena and Dr. Elisabetta Pietrostefani from the Geographic Data Science Lab at the University of Liverpool, United Kingdom. The module has two main aims. It seeks to provide hands-on experience and training in:

- The design and generation of web-based mapping and geographical information tools.
- The use of software to access, analyse and visualize web-based geographical information.

 A compilation of this web course is hosted as a GitHub repository that you can access:

- As an html website.
- As a GitHub repository.

Contact

Ugur Ursavas - gfilo [at] liverpool.ac.uk Lecturer in Geographic Data Science Office 1xx, Roxby Building, University of Liverpool - 74 Bedford St S, Liverpool, L69 7ZT, United Kingdom.

Elisabetta Pietrostefani - e.pietrostefani [at] liverpool.ac.uk Lecturer in Geographic Data Science Office 6xx, Roxby Building, University of Liverpool - 74 Bedford St S, Liverpool, L69 7ZT, United Kingdom.

Overview

Aims

This module aims to provide hands-on experience and training in: - The design and generation of (good looking) web-based mapping and geographical information tools. - The use of software to access, analyse and visualize web-based geographical information.

Learning Outcomes

By the end of the module, students should be able to:

(2) Visualise and represent geo-data through static and dynamic maps.
(3) Recognise and describe the component of web based mapping infrastructure.
(4) Collect Web-based data.
(5) Generate interactive maps and dashboards.
(6) Understand basic concepts of spatial network analysis.
(7) Manipulate geo-data through scripting in Python.

Feedback

Formal assessment. Two pieces of coursework (50%/50%). Equivalent to 2,500 words each

Verbal face-to-face feedback. Immediate face-to-face feedback will be provided during computer, discussion and clinic sessions in interaction with staff. This will take place in all live sessions during the semester. *Teams Forum*. Asynchronous written feedback will be provided via Teams. Students are encouraged to contribute by asking and answering questions relating to the module content. Staff will monitor the forum Monday to Friday 9am-5pm, but it will be open to students to make contributions at all times. Response time will vary depending on the complexity of the question and staff availability.

1 Introduction & Python Refresher

The **Lecture slides** can be found here.

This **lab**'s notebook can be downloaded from here.

1.1 Part I: Powerful Web Mapping Examples

This part of the lab has two main components: 1. The first one will require you to find a partner and work together with her/him 2. And the second one will involve group discussion.

1.1.1 Paired Activity

In pairs, find **three** examples where web maps are used to communicate an idea. Complete the following sheet for each example:

- **Substantive**
  - `Title`: Title of the map/project
  - `Author`: Who is behind the project?
  - `Big idea`: a "one-liner" on what the project tries to accomplish –
  - `Message`: what does the map try to get accross
- **Technical**
  - `URL`:
  - `Interactivity`: does the map let you interact with it in any way? Yes/No
  - `Zoomable`: can you explore the map at different scales? Yes/No
  - `Tooltips`:
  - `Basemap`: Is there an underlying map providing geographical context? Yes/No. If so, who is it provided by?
  - `Technology`: can you guess what technology does this map rely on?

Post each sheet as a separate item on the Teams channel for Lab No.1

1.1.1.1 Example

The project "WHO Coronavirus (COVID-19) Dashboard"

- **Substantive**
  - `Title`: WHO Coronavirus (COVID-19) Dashboard
  - `Author`: World Health Organization
  - `Big idea`: Shows confirmed COVID-19 cases and deaths by country to date
  - `Message`: The project displays a map of the world where COVID-19 cases are shown by country. This element is used to show which countries have had more cases (large trends). A drop down button allows us to visualise the map by a) Total per 100,000 population b) % change in the last 7 days c) newly reported in the last 7 days d) newly reported in the last 24 hours.
- **Technical**
  - `URL`: https://covid19.who.int/
  - `Interactivity`: Yes
  - `Zoomable`: Yes
  - `Tooltips`: Yes
  - `Basemap`: No
  - `Technology`: Unknown

Here are a couple of other COVID-19 examples of web-maps that where basemaps and technology is easier to spot.

- "London School of Hygiene & Tropical Medicine - COVID-19 tracker"
- "Tracking Coronavirus in the United Kingdom: Latest Map and Case Count"

1.1.2 Class discussion

We will select a few examples posted and collectively discuss (some of) the following questions:

1. What makes them powerful, what "speaks" to us?
2. What could be improved, what is counter-intuitive?
3. What design elements do they rely on?
4. What technology do they use?

1.1.3 References

- For an excellent coverage of "visualisation literacy", Chapter 11 of Andy Kirk's "Data Visualisation" is a great start. Lab: Getting up to speed for web mapping
- A comprehensive overview of computational notebooks and how they relate to modern scientific work is available on Ch.1 of the GDS book.
- A recent overview of notebooks in Geography is available in Boeing & Arribas-Bel (2021)

1.2 Part II: Python/Pandas (Refresher)

Gabriele Filomena has prepared this notebook by readapting material shared on this repository.

1.2.1 Python

A quick overview of ubiquitous programming concepts including data types, for loops, if-then-else conditionals, and functions.

```
import numpy as np
import pandas as pd
```

```
# integers (int)
x = 100
type(x)
```

```
# floating-point numbers (float)
x = 100.5
type(x)
```

```
# sequence of characters (str)
x = 'Los Angeles, CA 90089'
len(x)
```

```
# list of items
x = [1, 2, 3, 'USC']
len(x)
```

```
# sets are unique
x = {2, 2, 3, 3, 1}
x
```

```
# tuples are immutable sequences
latlng = (34.019425, -118.283413)
type(latlng)
```

```
# you can unpack a tuple
lat, lng = latlng
type(lat)
```

```
# dictionary of key:value pairs
iceland = {'Country': 'Iceland', 'Population': 372520, 'Capital': 'Reykjavík', '% Foreign Po
type(iceland)
```

```
# you can convert types
x = '100'
print(type(x))
y = int(x)
print(type(y))
```

```
# you can loop through an iterable, such as a list or tuple
for coord in latlng:
    print('Current coordinate is:', coord)
```

```
# loop through a dictionary keys and values as tuples
for key, value in iceland.items():
    print(key, value)
```

```
# booleans are trues/falses
x = 101
x > 100
```

```
# use two == for equality and one = for assignment
x == 100
```

```
# if, elif, else for conditional branching execution
x = 101
if x > 100:
    print('Value is greater than 100.')
elif x < 100:
    print('Value is less than 100.')
else:
    print('Value is 100.')
```

```
# use functions to encapsulate and reuse bits of code
def convert_items(my_list, new_type=str):
    # convert each item in a list to a new type
    new_list = [new_type(item) for item in my_list]
    return new_list

l = [1, 2, 3, 4]
convert_items(l)
```

1.2.2 `pandas` Series and DataFrames

pandas has two primary data structures we will work with: `Series` and `DataFrame`.

1.2.2.1 Pandas Series

```
# a pandas series is based on a numpy array: it's fast, compact, and has more functionality
# it has an index which allows you to work naturally with tabular data
my_list = [8, 5, 77, 2]
my_series = pd.Series(my_list)
my_series
```

```
# look at a list-representation of the index
my_series.index.tolist()
```

```
# look at the series' values themselves
my_series.values
```

```
# what's the data type of the series' values?
type(my_series.values)
```

```
# what's the data type of the individual values themselves?
my_series.dtype
```

1.2.2.2 Pandas DataFrames

```
# a dict can contain multiple lists and label them
my_dict = {'hh_income'  : [75125, 22075, 31950, 115400],
           'home_value' : [525000, 275000, 395000, 985000]}
my_dict
```

```
# a pandas dataframe can contain one or more columns
# each column is a pandas series
# each row is a pandas series
# you can create a dataframe by passing in a list, array, series, or dict
df = pd.DataFrame(my_dict)
df
```

```
# the row labels in the index are accessed by the .index attribute of the DataFrame object
df.index.tolist()
```

```
# the column labels are accessed by the .columns attribute of the DataFrame object
df.columns
```

```
# the data values are accessed by the .values attribute of the DataFrame object
# this is a numpy (two-dimensional) array
df.values
```

1.2.3 Loading data in Pandas

Usually, you'll work with data by loading a dataset file into pandas. CSV is the most common format. But pandas can also ingest tab-separated data, JSON, and proprietary file formats like Excel .xlsx files, Stata, SAS, and SPSS.

Below, notice what pandas's `read_csv` function does:

1. Recognize the header row and get its variable names.
2. Read all the rows and construct a pandas DataFrame (an assembly of pandas Series rows and columns).
3. Construct a unique index, beginning with zero.
4. Infer the data type of each variable (i.e., column).

```
# load a data file
# note the relative filepath! where is this file located?
# use dtype argument if you don't want pandas to guess your data types
df = pd.read_csv('../data/GTD_2022.csv', low_memory = False)
```

```
to_replace = [-9, -99, "-9", "-99"]
for value in to_replace:
    df = df.replace(value, np.NaN)

df['eventid'] = df['eventid'].astype("Int64")
```

```
# dataframe shape as rows, columns
df.shape
```

```
# or use len to just see the number of rows
len(df)
```

```
# view the dataframe's "head"
df.head()
```

```
# view the dataframe's "tail"
df.tail()
```

```
# column data types
df.dtypes
```

```
# or
for dt in df.columns[:10]:
    print(dt, type(dt))
```

1.2.4 Selecting and slicing data from a DataFrame

```
# CHEAT SHEET OF COMMON TASKS
# Operation                       Syntax           Result
#------------------------------------------------------------
# Select column by name           df[col]          Series
# Select columns by name          df[col_list]     DataFrame
# Select row by label             df.loc[label]    Series
# Select row by integer location  df.iloc[loc]     Series
# Slice rows by label             df.loc[a:c]      DataFrame
# Select rows by boolean vector   df[mask]         DataFrame
```

1.2.4.1 Select DataFrame's column(s) by name

```
# select a single column by column name
# this is a pandas series
df['country']
```

```
# select multiple columns by a list of column names
# this is a pandas dataframe that is a subset of the original
df[['country_txt', 'year']]
```

```
# create a new column by assigning df['new_col'] to some values
# people killed every perpetrator
df['killed_per_attacker'] = df['nkill'] / df['nperps']

# inspect the results
df[['country', 'year', 'nkill', 'nperps', 'killed_per_attacker']].head(15)
```

1.2.4.2 Select row(s) by label

```
# use .loc to select by row label
# returns the row as a series whose index is the dataframe column names
df.loc[0]
```

```
# use .loc to select single value by row label, column name
df.loc[15, 'gname'] #group name
```

```
# slice of rows from label 5 to label 7, inclusive
# this returns a pandas dataframe
df.loc[5:7]
```

```
# slice of rows from label 17 to label 27, inclusive
# slice of columns from country_txt to city, inclusive
df.loc[17:27, 'country_txt':'city']
```

```
# subset of rows from with labels in list
# subset of columns with names in list
df.loc[[1, 350], ['country', 'gname']]
```

```
# you can use a column of identifiers as the index (indices do not *need* to be unique)
df_gname = df.set_index('gname')
df_gname.index.is_unique
```

```
df_gname.head(3)
```

```
# .loc works by label, not by position in the dataframe
try:
    df_gname.loc[0]
except KeyError as e:
    print('label not found')
```

```
# the index now contains gname values, so you have to use .loc accordingly to select by row l
df_gname.loc['Taliban'].head()
```

1.2.4.3 Select by (integer) position - Independent from actual Index

```
# get the row in the zero-th position in the dataframe
df.iloc[0]
```

```
# you can slice as well
# note, while .loc is inclusive, .iloc is not
# get the rows from position 0 up to but not including position 3 (ie, rows 0, 1, and 2)
df.iloc[0:3]
```

```
# get the value from the row in position 3 and the column in position 2 (zero-indexed)
df.iloc[3, 6] #country_txt
```

1.2.4.4 Select/filter by value

You can subset or filter a dataframe for based on the values in its rows/columns.

```
# filter the dataframe by urban areas with more than 25 million residents
df[df['nkill'] > 30].head()
```

```
# you can chain multiple conditions together
# pandas logical operators are: | for or, & for and, ~ for not
# these must be grouped by using parentheses due to order of operations
df[['country','nkill', 'nwound']][(df['nkill'] > 200) & (df['nwound'] > 10)].head()
# columns on the left-hand side are here used to slice the resulting output
```

```
# ~ means not... it essentially flips trues to falses and vice-versa
df[['country','nkill', 'nwound']][~(df['nkill'] > 200) & (df['nwound'] > 10)]
```

1.2.5 Grouping and summarizing

```
# group by terroristic group name
groups = df.groupby('gname')
```

```
# what is the median number of people killed per event across the different groups?
groups['nkill'].median().sort_values(ascending=False)
```

```
# look at several columns' medians by group
groups[['nkill', 'nwound', 'nperps']].median()
```

```
# you can create a new dataFrame by directly passing columns between "[[ ]]", after the grou
# to do so, you also need to pass a function that can deal with the values (e.g. sum..etc)
western_europe = df[df.region_txt == 'Western Europe']
western_europe.groupby('country_txt')[['nkill', 'nwound']].sum().sort_values('nkill', ascend
```

1.2.6 Indexes

Each `DataFrame` has an index. Indexes do not have to be unique (but that would be for the best)

```
# resetting index (when loading a .csv file pandas creates an index automatically, from 0 to
df.reset_index(drop = True).sort_index().head() # this does not assign the new index though,
```

```
#this does assign the new index to your df
df = df.reset_index(drop = True).sort_index()
df.head()
```

```
# index isn't unique
df.index.is_unique
```

```
# you can set a new index
# drop -> Delete columns to be used as the new index.
# append ->  whether to append columns to existing index.
df = df.set_index('eventid', drop=True, append=False)
df.index.name = None # remove the index "name"
df.head()

# this index is not ideal, but it's the original source's id
```

1.3 Part III: Geospatial Vector data in Python

Gabriele Filomena has prepared this notebook by readapting material shared on this repository. Copyright (c) 2018, Joris Van den Bossche.

```
%matplotlib inline

import geopandas as gpd
```

1.3.1 Importing geospatial data

GeoPandas builds on Pandas types `Series` and `Dataframe`, by incorporating information about geographical space.

- `GeoSeries`: a Series object designed to store shapely geometry object
- `GeoDataFrame`: object is a pandas DataFrame that has a column with geometry (that contains a *Geoseries*)

We can use the GeoPandas library to read many of GIS file formats (relying on the `fiona` library under the hood, which is an interface to GDAL/OGR), using the `gpd.read_file` function. For example, let's start by reading a shapefile with all the countries of the world (adapted from http://www.naturalearthdata.com/downloads/110m-cultural-vectors/110m-admin-0-countries/, zip file is available in the `/data` directory), and inspect the data:

```
countries = gpd.read_file("../data/ne_countries.zip")
# or if the archive is unpacked:
# countries = gpd.read_file("../data/ne_countries.shp")
```

```
countries.head()
```

```
countries.plot()
```

We observe that:

- Using `.head()` we can see the first rows of the dataset, just like we can do with Pandas.
- There is a `geometry` column and the different countries are represented as polygons
- We can use the `.plot()` (matplotlib) method to quickly get a *basic* visualization of the data

1.3.2 What's a GeoDataFrame?

We used the GeoPandas library to read in the geospatial data, and this returned us a `GeoDataFrame`:

```
type(countries)
```

A GeoDataFrame contains a tabular, geospatial dataset:

- It has a 'geometry' column that holds the geometry information (or features in GeoJSON).
- The other columns are the **attributes** (or properties in GeoJSON) that describe each of the geometries.

Such a `GeoDataFrame` is just like a pandas `DataFrame`, but with some additional functionality for working with geospatial data: * A `geometry` attribute that always returns the column with the geometry information (returning a `GeoSeries`). The column name itself does not necessarily need to be 'geometry', but it will always be accessible as the `geometry` attribute. * It has some extra methods for working with spatial data (area, distance, buffer, intersection, …) see here, for example.

```
countries.geometry.head()
```

```
type(countries.geometry)
```

```
countries.geometry.area
```

It's still a `DataFrame`, so we have all the `pandas` functionality available to use on the geospatial dataset, and to do data manipulations with the attributes and geometry information together. For example, we can calculate the average population over all countries (by accessing the 'pop_est' column, and calling the `mean` method on it):

```
countries['pop_est'].mean()
```

```
africa = countries[countries['continent'] == 'Africa']
```

```
africa.plot();
```

The rest of the tutorial is going to assume you already know some pandas basics, but we will try to give hints for that part for those that are not familiar.

Important:

- A `GeoDataFrame` allows to perform typical tabular data analysis together with spatial operations
- A `GeoDataFrame` (or *Feature Collection*) consists of:
  - **Geometries** or **features**: the spatial objects
  - **Attributes** or **properties**: columns with information about each spatial object

1.3.3 Geometries: Points, Linestrings and Polygons

Spatial **vector** data can consist of different types, and the 3 fundamental types are:

- **Point** data: represents a single point in space.
- **Line** data ("LineString"): represented as a sequence of points that form a line.
- **Polygon** data: represents a filled area.

And each of them can also be combined in multi-part geometries (See https://shapely.readthedocs.io/en/stable/n objects for extensive overview).

For the example we have seen up to now, the individual geometry objects are Polygons:

```
print(countries.geometry[2])
```

Let's import some other datasets with different types of geometry objects.

A dateset about cities in the world (adapted from http://www.naturalearthdata.com/downloads/110m-cultural-vectors/110m-populated-places/, zip file is available in the `/data` directory), consisting of `Point` data:

```
cities = gpd.read_file("../data/ne_cities.zip")
```

```
print(cities.geometry[0])
```

And a dataset of rivers in the world (from http://www.naturalearthdata.com/downloads/50m-physical-vectors/50m-rivers-lake-centerlines/, zip file is available in the `/data` directory) where each river is a `(Multi-)LineString`:

```
rivers = gpd.read_file("../data/ne_rivers.zip")
```

```
print(rivers.geometry[0])
```

1.3.4 The `shapely` library

The individual geometry objects are provided by the `shapely` library

```
from shapely.geometry import Point, Polygon, LineString
```

```
type(countries.geometry[0])
```

To construct one ourselves:

```
p = Point(0, 0)
```

```
print(p)
```

```
polygon = Polygon([(1, 1), (2,2), (2, 1)])
```

```
polygon.area
```

```
polygon.distance(p)
```

Important:

Single geometries are represented by `shapely` objects:

- If you access a single geometry of a GeoDataFrame, you get a shapely geometry object

- Those objects have similar functionality as geopandas objects (GeoDataFrame/GeoSeries). For example:
  - `single_shapely_object.distance(other_point)` -> distance between two points
  - `geodataframe.distance(other_point)` -> distance for each point in the geodataframe to the other point

1.3.5 Plotting

```
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt

fig, ax = plt.subplots(1, 1, figsize=(15, 10))
countries.plot(ax = ax, edgecolor='k', facecolor='none')
rivers.plot(ax=ax)
cities.plot(ax=ax, color='red')
ax.set(xlim=(-20, 60), ylim=(-40, 40))
```

1.3.6 Creating GeoDataFrames (withouth specifying the CRS)

```
gpd.GeoDataFrame({
    'geometry': [Point(1, 1), Point(2, 2)],
    'attribute1': [1, 2],
    'attribute2': [0.1, 0.2]})
```

```
# Creating a GeoDataFrame from an existing dataframe
# For example, if you have lat/lon coordinates in two columns:
df = pd.DataFrame(
    {'City': ['Buenos Aires', 'Brasilia', 'Santiago', 'Bogota', 'Caracas'],
     'Country': ['Argentina', 'Brazil', 'Chile', 'Colombia', 'Venezuela'],
     'Latitude': [-34.58, -15.78, -33.45, 4.60, 10.48],
     'Longitude': [-58.66, -47.91, -70.66, -74.08, -66.86]})
```

```
gdf = gpd.GeoDataFrame(df, geometry=gpd.points_from_xy(df.Longitude, df.Latitude))
gdf
```

2 Practice

Throughout the exercises in this course, we will work with several datasets about the city of Paris.

Here, we start with the following datasets:

- The administrative districts of Paris (https://opendata.paris.fr/explore/dataset/quartier_paris/): `paris_districts_utm.geojson`
- Real-time (at the moment I downloaded them ..) information about the public bicycle sharing system in Paris (vélib, https://opendata.paris.fr/explore/dataset/stations-velib-disponibilites-en-temps-reel/information/): `data/paris_bike_stations_mercator.gpkg`

Both datasets are provided as spatial datasets using a GIS file format.

Excercise 1:

We will start by exploring the bicycle station dataset (available as a GeoPackage file: `data/paris_bike_stations_mercator.gpkg`)

- Read the stations datasets into a GeoDataFrame called `stations`.
- Check the type of the returned object
- Check the first rows of the dataframes. What kind of geometries does this datasets contain?
- How many features are there in the dataset?

Hints

- Use `type(..)` to check any Python object type
- The `gpd.read_file()` function can read different geospatial file formats. You pass the file name as first argument.
- Use the `.shape` attribute to get the number of features

Exercise 2:

- Make a quick plot of the `stations` dataset.
- Make the plot a bit larger by setting the figure size to (12, 6) (hint: the `plot` method accepts a `figsize` keyword).

Exercise 3:

Next, we will explore the dataset on the administrative districts of Paris (available as a GeoJSON file: `../data/paris_districts_utm.geojson`)

- Read the dataset into a GeoDataFrame called `districts`.
- Check the first rows of the dataframe. What kind of geometries does this dataset contain?
- How many features are there in the dataset? (hint: use the `.shape` attribute)
- Make a quick plot of the `districts` dataset (set the figure size to (12, 6)).

Exercise 4:

What are the largest districts (biggest area)?

- Calculate the area of each district.
- Add this area as a new column to the `districts` dataframe.
- Sort the dataframe by the area column from largest to smallest values (descending).

Hints

- Adding a column can be done by assigning values to a column using the same square brackets syntax: `df['new_col'] = values`
- To sort the rows of a DataFrame, use the `sort_values()` method, specifying the colum to sort on with the `by='col_name'` keyword. Check the help of this method to see how to sort ascending or descending.

2.1 Part IV: Coordinate reference systems & Projections

Gabriele Filomena has prepared this notebook by readapting material shared on this repository.


```
countries = gpd.read_file("../data/ne_countries.zip")
cities = gpd.read_file("../data/ne_cities.zip")
rivers = gpd.read_file("../data/ne_rivers.zip")
```

2.1.1 Coordinate reference systems

Up to now, we have used the geometry data with certain coordinates without further wondering what those coordinates mean or how they are expressed.

> The **Coordinate Reference System (CRS)** relates the coordinates to a specific location on earth.

For an in-depth explanation, see https://docs.qgis.org/2.8/en/docs/gentle_gis_introduction/coordinate_referer

2.1.1.1 Geographic coordinates

Degrees of latitude and longitude.

E.g. 48°51 N, 2°17 E

The most known type of coordinates are geographic coordinates: we define a position on the globe in degrees of latitude and longitude, relative to the equator and the prime meridian. With this system, we can easily specify any location on earth. It is used widely, for example in GPS. If you inspect the coordinates of a location in Google Maps, you will also see latitude and longitude.

Attention!

in Python we use (lon, lat) and not (lat, lon)

- Longitude: [-180, 180]{{1}}
- Latitude: [-90, 90]{{1}}

2.1.2 Projected coordinates

`(x, y)` coordinates are usually in meters or feet

Although the earth is a globe, in practice we usually represent it on a flat surface: think about a physical map, or the figures we have made with Python on our computer screen. Going from the globe to a flat map is what we call a *projection*.

We project the surface of the earth onto a 2D plane so we can express locations in cartesian x and y coordinates, on a flat surface. In this plane, we then typically work with a length unit such as meters instead of degrees, which makes the analysis more convenient and effective.

However, there is an important remark: the 3 dimensional earth can never be represented perfectly on a 2 dimensional map, so projections inevitably introduce distortions. To minimize such errors, there are different approaches to project, each with specific advantages and disadvantages.

Some projection systems will try to preserve the area size of geometries, such as the Albers Equal Area projection. Other projection systems try to preserve angles, such as the Mercator projection, but will see big distortions in the area. Every projection system will always have some distortion of area, angle or distance.

Projected size vs actual size (Mercator projection):

2.1.3 Coordinate Reference Systems in Python / GeoPandas

A GeoDataFrame or GeoSeries has a `.crs` attribute which holds (optionally) a description of the coordinate reference system of the geometries:

```
countries.crs
```

For the `countries` dataframe, it indicates that it uses the EPSG 4326 / WGS84 lon/lat reference system, which is one of the most used for geographic coordinates.

It uses coordinates as latitude and longitude in degrees, as can you be seen from the x/y labels on the plot:

```
countries.plot()
```

The `.crs` attribute returns a `pyproj.CRS` object. To specify a CRS, we typically use some string representation:

- **EPSG code** Example: `EPSG:4326` = WGS84 geographic CRS (longitude, latitude)

For more information, see also http://geopandas.readthedocs.io/en/latest/projections.html.

2.1.3.1 Transforming to another CRS

We can convert a GeoDataFrame to another reference system using the `to_crs` function.

For example, let's convert the countries to the World Mercator projection (http://epsg.io/3395):

```
# remove Antartica, as the Mercator projection cannot deal with the poles
countries = countries[(countries['name'] != "Antarctica")]
countries_mercator = countries.to_crs(epsg=3395)  # or .to_crs("EPSG:3395")
countries_mercator.plot()
```

Note the different scale of x and y.

2.1.3.2 Why using a different CRS?

There are sometimes good reasons you want to change the coordinate references system of your dataset, for example:

- Different sources with different CRS -> need to convert to the same crs.

- Different countries/geographical areas with different CRS.
- Mapping (distortion of shape and distances).
- Distance / area based calculations -> ensure you use an appropriate projected coordinate system expressed in a meaningful unit such as meters or feet (**not degrees!**).

Important:

All the calculations (e.g. distance, spatial operations, etc.) that take place in `GeoPandas` and `Shapely` assume that your data is represented in a 2D cartesian plane, and thus the result of those calculations will only be correct if your data is properly projected.

2.2 Practice

Again, we will go back to the Paris datasets. Up to now, we provided the datasets in an appropriate projected CRS for the exercises. But the original data were actually using geographic coordinates. In the following exercises, we will start from there.

Going back to the Paris districts dataset, this is now provided as a GeoJSON file (`"../data/paris_districts.geojson"`) in geographic coordinates.

For converting the layer to projected coordinates, we will use the standard projected CRS for France is the RGF93 / Lambert-93 reference system, referenced by the `EPSG:2154` number.

Exercise: Projecting a GeoDataFrame

- Read the districts datasets (`../data/paris_districts.geojson"`) into a GeoDataFrame called `districts`.
- Look at the CRS attribute of the GeoDataFrame. Do you recognize the EPSG number?
- Make a plot of the `districts` dataset.
- Calculate the area of all districts.
- Convert the `districts` to a projected CRS (using the `EPSG:2154` for France). Call the new dataset `districts_RGF93`.
- Make a similar plot of `districts_RGF93`.
- Calculate the area of all districts again with `districts_RGF93` (the result will now be expressed in m^2).

Hints

- The CRS information is stored in the `.crs` attribute of a GeoDataFrame.
- Making a simple plot of a GeoDataFrame can be done with the `.plot()` method.
- Converting to a different CRS can be done with the `.to_crs()` method, and the CRS can be specified as an EPSG number using the `epsg` keyword.

3 Panel spatial lags

This document shows how one can calculate spatial lag of a series of variables over several periods of time, assuming the geography (e.g., W) remains constant.

```
import pandas
import geopandas
from libpysal import graph
```

NOTE - This implementation relies on the new `graph` structures for spatial weights in PySAL. For that reason, a recent version of the library is required.

3.1 Data

- Tabular data

Note we drop the names as they're irrelevant here (we have unique IDs) and index the table on region ID and year. The resulting table contains only the variables to lag as columns.

```
panel = (
    pandas.read_csv(
        'spatial_lag_panel_data.csv',
        encoding = 'ISO-8859-9' # Turkish encoding
    )
    .set_index(['asdf_id', 'year'])
    .drop(columns=['shapeName'])
)
```

- Geographic data

```
geo = geopandas.read_file('TUR_ADM1.geojson').set_index('asdf_id')
```

```
ERROR 1: PROJ: proj_create_from_database: Open of /opt/conda/envs/gds/share/proj failed
```

3.2 Lag computation

First, we compute the spatial weights we will use. In this example, we pick queen contiguity, although other criteria are available and possibly valid too.

```
w = (
    graph.Graph.build_contiguity(geo, rook=False)
    .transform('R')
)
```

Now we're ready to compute the lags. We approach this as a nested `for` loop, where we iterate through every year and, within that, through every variable. To make computation more efficient, we first generate the frame where results will be stored (`lags`).

```
lags = pandas.DataFrame(index=panel.index, columns=panel.columns)

for year in panel.index.get_level_values('year').unique():
    for var in lags.columns:
        vals = panel.loc[pandas.IndexSlice[:, year], var]
        lags.loc[vals.index, var] = w.lag(vals)
```

We can now write the lagged values to disk:

```
lags.to_csv('lagged.csv')
```

4 Build a queen contiguity matrix from a regular 3x3

```
import numpy as np
import pandas as pd
import seaborn as sns
sns.set_style("darkgrid")
sns.set_context(context="paper", font_scale=1.5, rc=None)
sns.set(font="serif")
import seaborn

import geopandas as gpd
import matplotlib.pyplot as plt

import libpysal
from libpysal  import weights
from pysal.explore import esda
import esda
from esda.moran import Moran, Moran_Local

import splot
from splot.esda import moran_scatterplot, plot_moran, lisa_cluster
from splot.libpysal import plot_spatial_weights

from giddy.directional import Rose
import os

import contextily
```

```
from numpy.random import seed
seed(12345678)
```

```
os.chdir('F:/projects/2024/informal')
```

```
geojson_data = gpd.read_file("F:/projects/2024/informal/merged2.geojson")
```

```
geojson_data
```

	id	Level_x	asdf_id	gqid_x	shapeGroup_x	shapeID_x	shapeISO_x	sha
0	0	ADM1	0	0	TUR	TUR-ADM1-80719077B77822815	TR-01	Ad
1	1	ADM1	1	1	TUR	TUR-ADM1-80719077B28599679	TR-02	Ad
2	2	ADM1	2	2	TUR	TUR-ADM1-80719077B84550223	TR-03	Af
3	3	ADM1	3	3	TUR	TUR-ADM1-80719077B65173278	TR-04	Ağ
4	4	ADM1	4	4	TUR	TUR-ADM1-80719077B72380009	TR-05	An
...	...	...	...	...	...	...	...	...
76	76	ADM1	76	76	TUR	TUR-ADM1-80719077B83749854	TR-77	Ya
77	77	ADM1	77	77	TUR	TUR-ADM1-80719077B67647683	TR-66	Yo
78	78	ADM1	78	78	TUR	TUR-ADM1-80719077B51620989	TR-67	Zo
79	79	ADM1	79	79	TUR	TUR-ADM1-80719077B759750	TR-73	Şır
80	80	ADM1	80	80	TUR	TUR-ADM1-80719077B19552530	TR-63	Şa

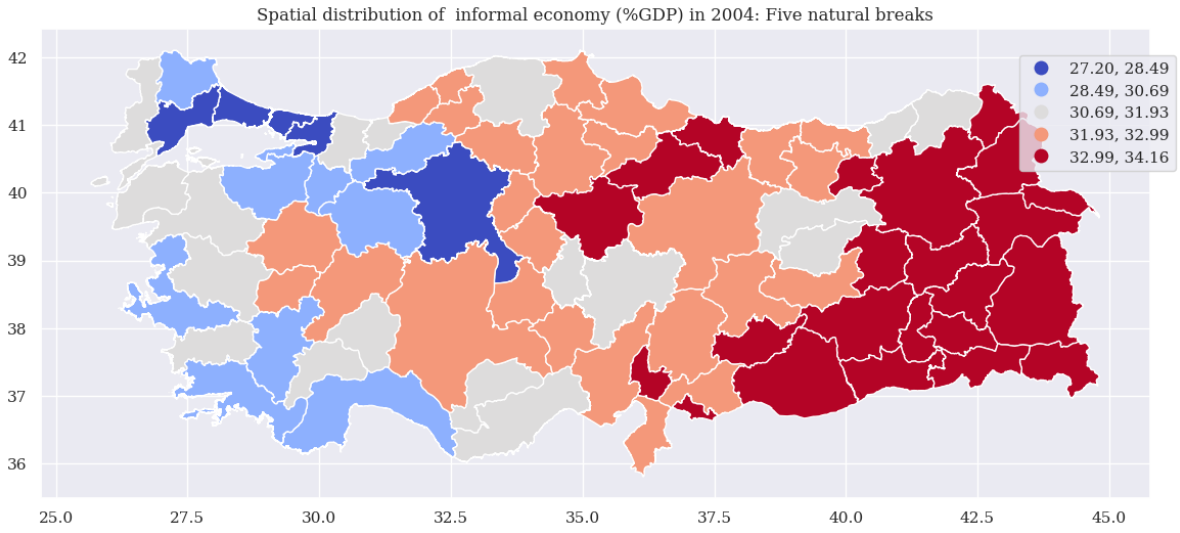
```
geojson_data['coords'] = geojson_data['geometry'].apply(lambda x: x.representative_point().co
geojson_data['coords'] = [coords[0] for coords in geojson_data['coords']]
```

```
fig, ax = plt.subplots(figsize=(14,8))
geojson_data.plot(column="2004", scheme='NaturalBreaks', k=5, cmap='coolwarm', legend=True, l
plt.title('Spatial distribution of  informal economy (%GDP) in 2004: Five natural breaks')
for idx, row in fig, ax = plt.subplots(figsize=(14,8))
merged_data.plot(column="2004", scheme='NaturalBreaks', k=5, cmap='coolwarm', legend=True, le
plt.title('Spatial distribution of  informal economy (%GDP) in 2004: Five natural breaks')
for idx, row in merged_data.iterrows():
    ax.annotate(text=row['province'], xy=row['coords'], fontsize=10,
                horizontalalignment='center', bbox={'facecolor': 'white', 'alpha':0.8, 'pad


plt.tight_layout()
ax.axis("off")

plt.show().iterrows():
    ax.annotate(text=row['province'], xy=row['coords'], fontsize=10,
                horizontalalignment='center', bbox={'facecolor': 'white', 'alpha':0.8, 'pad
```

```
plt.tight_layout()
ax.axis("off")

plt.show()
```

```
C:\Users\uursavas\AppData\Local\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1436:
  warnings.warn(
```

```
NameError: name 'merged_data' is not defined
```




```
# lattice stored in a geo-table
w = weights.contiguity.Queen.from_dataframe(geojson_data)
w.transform = "R"
w.neighbors
```

```
C:\Users\uursavas\AppData\Local\Temp\ipykernel_20972\3870122094.py:3: FutureWarning: `use_inc
  w = weights.contiguity.Queen.from_dataframe(geojson_data)
```

```
{0: [36, 41, 57, 61, 46, 63],
 1: [32, 80, 54, 25, 41],
 2: [19, 52, 53, 38, 24, 74, 31],
```

```
3: [17, 37, 59, 75, 44, 30],
4: [66, 77, 71, 23],
5: [19, 52, 38, 57, 58, 43],
6: [64, 11, 30],
7: [40, 24, 58, 55],
8: [20, 53, 21, 55, 40],
9: [48, 50, 18, 52, 22, 10, 31],
10: [50, 52, 9, 60, 61],
11: [44, 6, 30],
12: [42, 45, 78],
13: [17, 67, 56, 25, 59, 79],
14: [64, 34, 72, 29, 30],
15: [65, 18, 51, 20, 53, 31],
16: [30, 73, 59, 28, 29, 25],
17: [3, 67, 75, 59, 13],
18: [65, 9, 42, 78, 15, 22, 26, 31],
19: [2, 5, 38, 24, 58],
20: [51, 53, 8, 76, 15],
21: [8, 27, 70],
22: [48, 18, 23, 9, 42, 45],
23: [48, 66, 4, 68, 77, 22, 45],
24: [2, 19, 7, 74, 55, 58],
25: [1, 13, 80, 16, 54, 56, 59, 28],
26: [65, 18, 78],
27: [49, 21, 70],
28: [16, 54, 73, 29, 25],
29: [33, 34, 69, 73, 14, 16, 54, 28, 30],
30: [64, 3, 6, 11, 44, 14, 16, 59, 29],
31: [2, 18, 52, 53, 9, 15],
32: [80, 1, 36, 41, 63, 47],
33: [34, 69, 72, 29, 62],
34: [72, 33, 29, 14],
35: [75, 79],
36: [0, 32, 63],
37: [3, 44],
38: [2, 19, 52, 5],
39: [49, 51, 70],
40: [8, 55, 7],
41: [0, 1, 32, 69, 54, 46, 63],
42: [18, 22, 12, 45, 78],
43: [57, 52, 5],
44: [11, 3, 37, 30],
45: [68, 22, 23, 42, 12],
```

```
 46: [0, 61, 69, 41, 60, 77],
 47: [32],
 48: [50, 22, 23, 9, 77],
 49: [27, 70, 39],
 50: [48, 9, 10, 60, 77],
 51: [65, 20, 39, 76, 15],
 52: [2, 5, 38, 9, 10, 43, 57, 61, 31],
 53: [2, 20, 55, 8, 74, 31, 15],
 54: [1, 69, 41, 28, 29, 25],
 55: [53, 7, 40, 8, 74, 24],
 56: [80, 67, 25, 13, 79],
 57: [0, 52, 5, 43, 61],
 58: [24, 19, 5, 7],
 59: [16, 17, 3, 25, 13, 30],
 60: [50, 77, 10, 61, 46],
 61: [0, 52, 57, 10, 60, 46],
 62: [33, 66, 69, 71],
 63: [0, 41, 32, 36],
 64: [72, 30, 6, 14],
 65: [26, 18, 51, 15],
 66: [4, 68, 23, 71, 62],
 67: [17, 56, 75, 13, 79],
 68: [66, 45, 23],
 69: [33, 71, 41, 77, 46, 54, 29, 62],
 70: [49, 27, 21, 39],
 71: [66, 4, 69, 77, 62],
 72: [64, 33, 34, 14],
 73: [16, 28, 29],
 74: [24, 2, 53, 55],
 75: [17, 3, 67, 35, 79],
 76: [51, 20],
 77: [4, 69, 71, 46, 48, 50, 23, 60],
 78: [18, 26, 42, 12],
 79: [35, 67, 56, 75, 13],
 80: [32, 1, 56, 25]}
```

```
geojson_data["2004_lag"] = weights.spatial_lag.lag_spatial(
    w, geojson_data["2004"]
)
```

```
geojson_data["2021_lag"] = weights.spatial_lag.lag_spatial(
    w, geojson_data["2021"]
```

```
)
```

```
moran = esda.moran.Moran(geojson_data["2004"], w)
```

```
moran.I
```

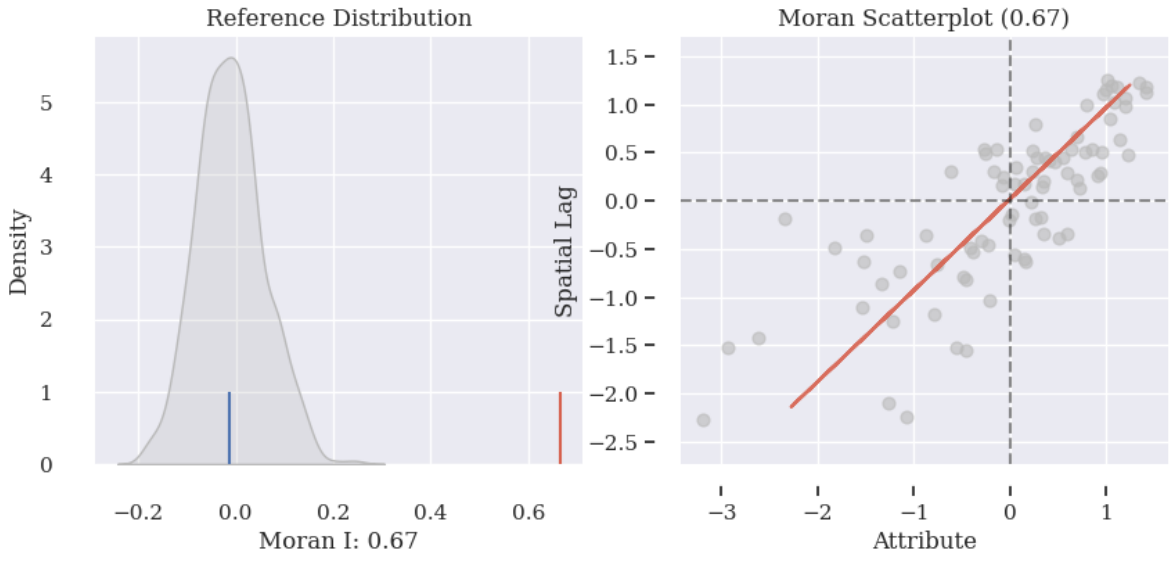
```
0.6673411138948363
```

```
moran1 = esda.moran.Moran(geojson_data["2021"], w)
moran.I
```

```
0.6673411138948363
```

```
plot_moran(moran);
```

```
C:\Users\uursavas\AppData\Local\anaconda3\Lib\site-packages\splot\_viz_esda_mpl.py:354: Futu

`shade` is now deprecated in favor of `fill`; setting `fill=True`.
This will become an error in seaborn v0.14.0; please update your code.

  sbn.kdeplot(moran.sim, shade=shade, color=color, ax=ax, **kwargs)
```



```
from splot import esda as esdaplot
```

```
print(moran2004.I, moran2021.I)
```

```
NameError: name 'moran2004' is not defined
```

```
moran.p_sim
```

```
0.001
```

```
lisa = esda.moran.Moran_Local(geojson_data["2004"], w)
```

```
esdaplot.lisa_cluster(lisa, geojson_data, p=0.01)


# Display the figure
plt.show()
```


```
lisa1 = esda.moran.Moran_Local(geojson_data["2021"], w)
```

```
esdaplot.lisa_cluster(lisa1, geojson_data, p=0.01)


# Display the figure
plt.show()
```


HH
LL
ns

5

```
import pandas
import geopandas
from libpysal import graph
import os
import geopandas as gpd
```

```
os.chdir('F:/projects/2024/ursavas_alahmadi_chen/data/ntl_data')
```

```
panel = (
    pandas.read_csv(
        'spatial.csv',
        encoding = 'ISO-8859-9' # Turkish encoding
    )
    .set_index(['asdf_id', 'year'])

)
```

```
geo = gpd.read_file("F:/projects/2024/informal/TUR_ADM1.geojson")
```

```
w = (
    graph.Graph.build_contiguity(geo, rook=False)
    .transform('R')
)
```

```
w.neighbors
```

```
{0: (36, 41, 46, 57, 61, 63),
 1: (25, 32, 41, 54, 80),
 2: (19, 24, 31, 38, 52, 53, 74),
 3: (17, 30, 37, 44, 59, 75),
 4: (23, 66, 71, 77),
 5: (19, 38, 43, 52, 57, 58),
```

```
6: (11, 30, 64),
7: (24, 40, 55, 58),
8: (20, 21, 40, 53, 55),
9: (10, 18, 22, 31, 48, 50, 52),
10: (9, 50, 52, 60, 61),
11: (6, 30, 44),
12: (42, 45, 78),
13: (17, 25, 56, 59, 67, 79),
14: (29, 30, 34, 64, 72),
15: (18, 20, 31, 51, 53, 65),
16: (25, 28, 29, 30, 59, 73),
17: (3, 13, 59, 67, 75),
18: (9, 15, 22, 26, 31, 42, 65, 78),
19: (2, 5, 24, 38, 58),
20: (8, 15, 51, 53, 76),
21: (8, 27, 70),
22: (9, 18, 23, 42, 45, 48),
23: (4, 22, 45, 48, 66, 68, 77),
24: (2, 7, 19, 55, 58, 74),
25: (1, 13, 16, 28, 54, 56, 59, 80),
26: (18, 65, 78),
27: (21, 49, 70),
28: (16, 25, 29, 54, 73),
29: (14, 16, 28, 30, 33, 34, 54, 69, 73),
30: (3, 6, 11, 14, 16, 29, 44, 59, 64),
31: (2, 9, 15, 18, 52, 53),
32: (1, 36, 41, 47, 63, 80),
33: (29, 34, 62, 69, 72),
34: (14, 29, 33, 72),
35: (75, 79),
36: (0, 32, 63),
37: (3, 44),
38: (2, 5, 19, 52),
39: (49, 51, 70),
40: (7, 8, 55),
41: (0, 1, 32, 46, 54, 63, 69),
42: (12, 18, 22, 45, 78),
43: (5, 52, 57),
44: (3, 11, 30, 37),
45: (12, 22, 23, 42, 68),
46: (0, 41, 60, 61, 69, 77),
47: (32,),
48: (9, 22, 23, 50, 77),
```

```
 49: (27, 39, 70),
 50: (9, 10, 48, 60, 77),
 51: (15, 20, 39, 65, 76),
 52: (2, 5, 9, 10, 31, 38, 43, 57, 61),
 53: (2, 8, 15, 20, 31, 55, 74),
 54: (1, 25, 28, 29, 41, 69),
 55: (7, 8, 24, 40, 53, 74),
 56: (13, 25, 67, 79, 80),
 57: (0, 5, 43, 52, 61),
 58: (5, 7, 19, 24),
 59: (3, 13, 16, 17, 25, 30),
 60: (10, 46, 50, 61, 77),
 61: (0, 10, 46, 52, 57, 60),
 62: (33, 66, 69, 71),
 63: (0, 32, 36, 41),
 64: (6, 14, 30, 72),
 65: (15, 18, 26, 51),
 66: (4, 23, 62, 68, 71),
 67: (13, 17, 56, 75, 79),
 68: (23, 45, 66),
 69: (29, 33, 41, 46, 54, 62, 71, 77),
 70: (21, 27, 39, 49),
 71: (4, 62, 66, 69, 77),
 72: (14, 33, 34, 64),
 73: (16, 28, 29),
 74: (2, 24, 53, 55),
 75: (3, 17, 35, 67, 79),
 76: (20, 51),
 77: (4, 23, 46, 48, 50, 60, 69, 71),
 78: (12, 18, 26, 42),
 79: (13, 35, 56, 67, 75),
 80: (1, 25, 32, 56)}
```

```
lags = pandas.DataFrame(index=panel.index, columns=panel.columns)

for year in panel.index.get_level_values('year').unique():
    for var in lags.columns:
        vals = panel.loc[pandas.IndexSlice[:, year], var]
        lags.loc[vals.index, var] = w.lag(vals)
```

```
lags.to_csv('lagged_ntl_panel.csv')
```

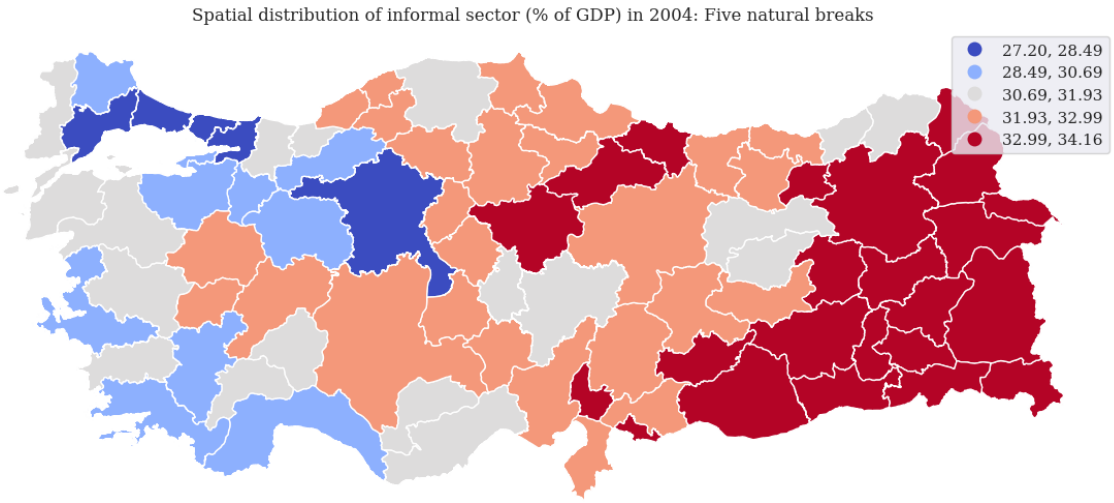
6 Spatial weights matrix and spatial lag

```
import numpy as np
import pandas as pd
import seaborn as sns
sns.set_style("darkgrid")
sns.set_context(context="paper", font_scale=1.5, rc=None)
sns.set(font="serif")
import seaborn

import geopandas as gpd
import matplotlib.pyplot as plt

import libpysal
from libpysal  import weights
from pysal.explore import esda
import esda
from esda.moran import Moran, Moran_Local

import splot
from splot.esda import moran_scatterplot, plot_moran, lisa_cluster
from splot.libpysal import plot_spatial_weights

from giddy.directional import Rose
import os

from numpy.random import seed
seed(12345)

os.chdir('F:/projects/2024/informal')
```

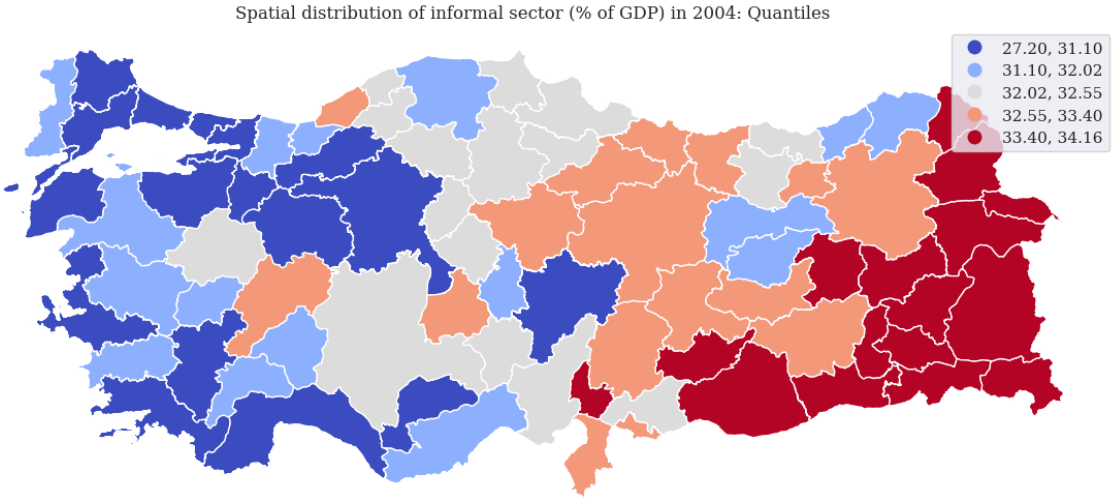
```
informal = pd.read_csv("F:/projects/2024/informal/results/informal_spatial.csv", encoding='u

geojson_data = gpd.read_file("F:/projects/2024/informal/TUR_ADM1.geojson")

merged_data = pd.merge(geojson_data, informal, how='left', left_on='asdf_id', right_on='asdf
```

```
fig, ax = plt.subplots(figsize=(12,8))
merged_data.plot(column="2004", scheme='NaturalBreaks', k=5, cmap='coolwarm', legend=True, ax
plt.title('Spatial distribution of informal sector (% of GDP) in 2004: Five natural breaks')
plt.tight_layout()
ax.axis("off")
plt.show()
```

```
C:\Users\uursavas\AppData\Local\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1436:
  warnings.warn(
```

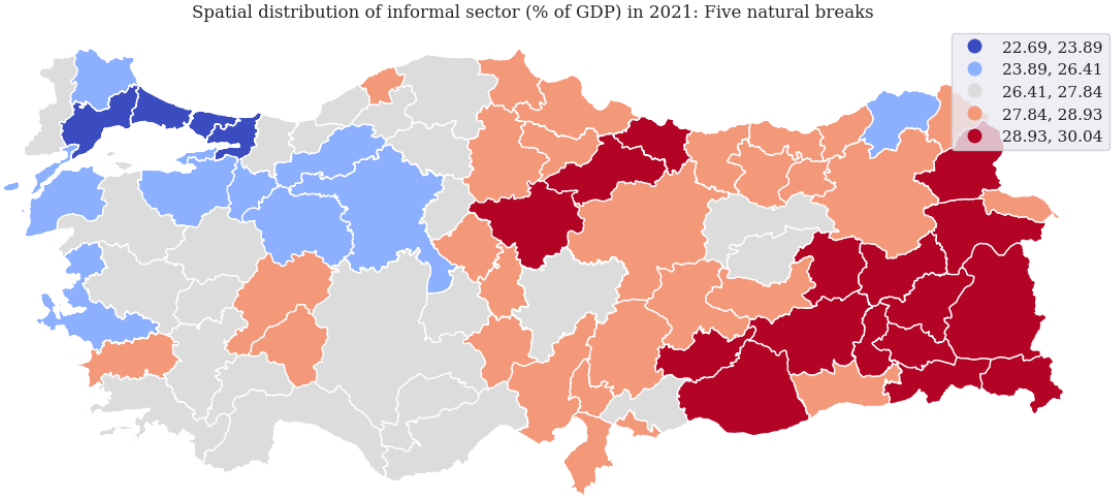



```
fig, ax = plt.subplots(figsize=(12,8))
merged_data.plot(column="2004", scheme='Quantiles',  cmap='coolwarm', legend=True, ax=ax)
plt.title('Spatial distribution of informal sector (% of GDP) in 2004: Quantiles')
plt.tight_layout()
ax.axis("off")
plt.show()
```

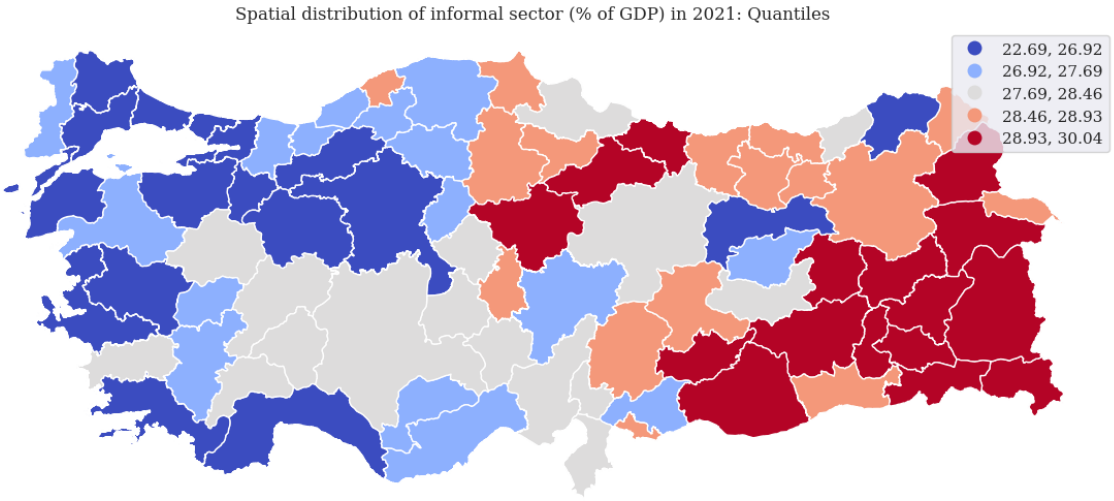



```
fig, ax = plt.subplots(figsize=(12,8))
merged_data.plot(column="2021", scheme='NaturalBreaks', k=5, cmap='coolwarm', legend=True, ax
plt.title('Spatial distribution of informal sector (% of GDP) in 2021: Five natural breaks')
plt.tight_layout()
ax.axis("off")
plt.show()
```

```
C:\Users\uursavas\AppData\Local\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1436:
  warnings.warn(
```



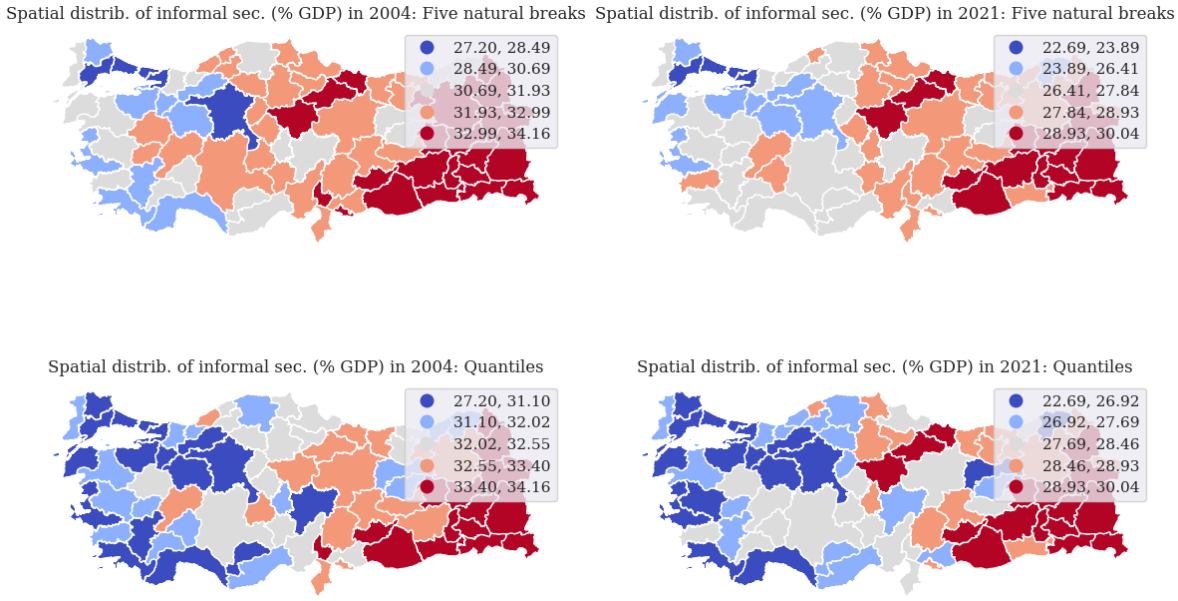
```
fig, ax = plt.subplots(figsize=(12,8))
merged_data.plot(column="2021", scheme='Quantiles',  cmap='coolwarm', legend=True, ax=ax)
plt.title('Spatial distribution of informal sector (% of GDP) in 2021: Quantiles')
plt.tight_layout()
ax.axis("off")
plt.show()
```




```
fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(12,8))
merged_data.plot(column="2004", scheme='NaturalBreaks', k=5, cmap='coolwarm', legend=True, a
merged_data.plot(column="2021", scheme='NaturalBreaks', k=5, cmap='coolwarm', legend=True, a
merged_data.plot(column="2004", scheme='Quantiles',  cmap='coolwarm', legend=True, ax=axes[1
merged_data.plot(column="2021", scheme='Quantiles',  cmap='coolwarm', legend=True, ax=axes[1
plt.tight_layout()
axes[0,0].axis("off")
axes[0,1].axis("off")
axes[1,0].axis("off")
axes[1,1].axis("off")

axes[0,0].title.set_text('Spatial distrib. of informal sec. (% GDP) in 2004: Five natural bre
axes[0,1].title.set_text('Spatial distrib. of informal sec. (% GDP) in 2021: Five natural bre
axes[1,0].title.set_text('Spatial distrib. of informal sec. (% GDP) in 2004: Quantiles')
axes[1,1].title.set_text('Spatial distrib. of informal sec. (% GDP) in 2021: Quantiles')
plt.show()
```

```
C:\Users\uursavas\AppData\Local\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1436:
```

```
  warnings.warn(
C:\Users\uursavas\AppData\Local\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1436:
  warnings.warn(
```




```
# Load weights matrix
w_queen = weights.contiguity.Queen.from_dataframe(merged_data)


# Row-standardize weights matrix
w_queen.transform = "R"
```

```
C:\Users\uursavas\AppData\Local\Temp\ipykernel_19752\3796823966.py:2: FutureWarning: `use_inc
  w_queen = weights.contiguity.Queen.from_dataframe(merged_data)
```

```
# Create spatial lag variables
merged_data["w_2004"] = weights.lag_spatial(w_queen, merged_data["2004"])
merged_data["w_2021"] = weights.lag_spatial(w_queen, merged_data["2021"])
```

```
merged_data[["2004", "w_2004", "2021", "w_2021"]]
```

	2004	w_2004	2021	w_2021
0	32.246507	32.285411	27.963184	27.976770
1	33.744531	32.976798	29.467049	28.881355
2	32.554150	31.500792	28.388501	27.414615
3	34.154317	33.724340	30.042331	29.336063
4	32.385330	32.815203	28.537311	28.871858
...	...	...	...	...
76	30.397265	28.646569	25.600179	24.339981
77	33.064559	32.355686	29.061970	28.332046
78	32.795890	31.436861	27.468281	27.367597
79	33.526188	33.762620	28.986277	29.301213
80	33.605297	33.306058	30.030058	28.821534

```
## Spatial autocorrelation
```

```
x2004 = merged_data["2004"]
x2021 = merged_data["2021"]
```

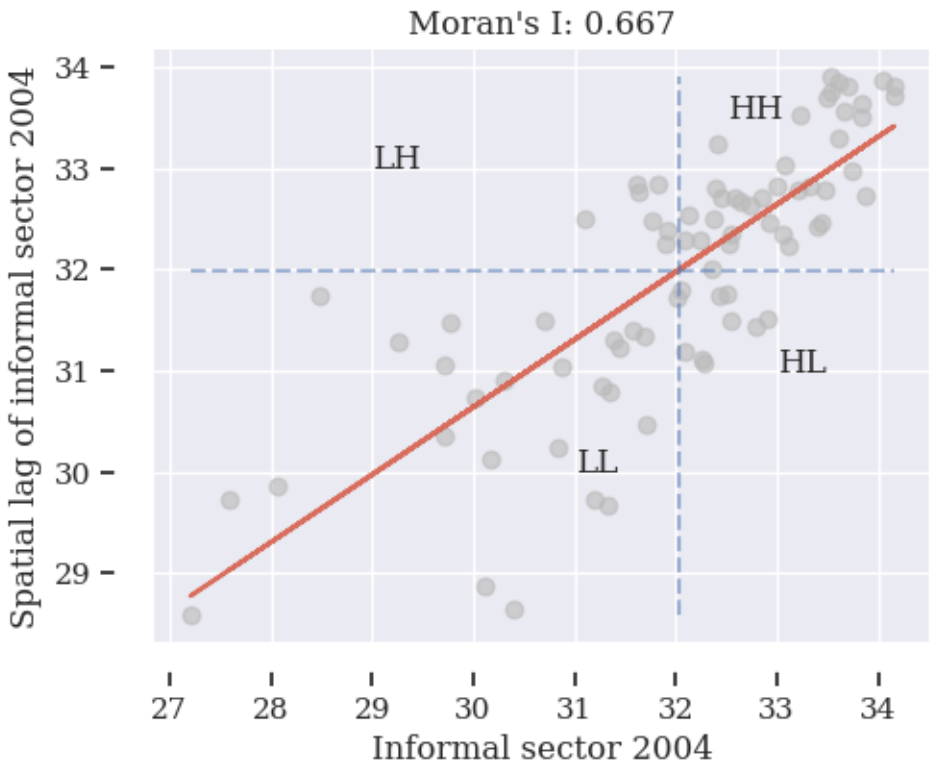
```
moran2004 = Moran(x2004, w_queen)
moran2021 = Moran(x2021, w_queen)
```

```
print(moran2004.I, moran2021.I)

print(moran2004.p_sim, moran2021.p_sim)
```

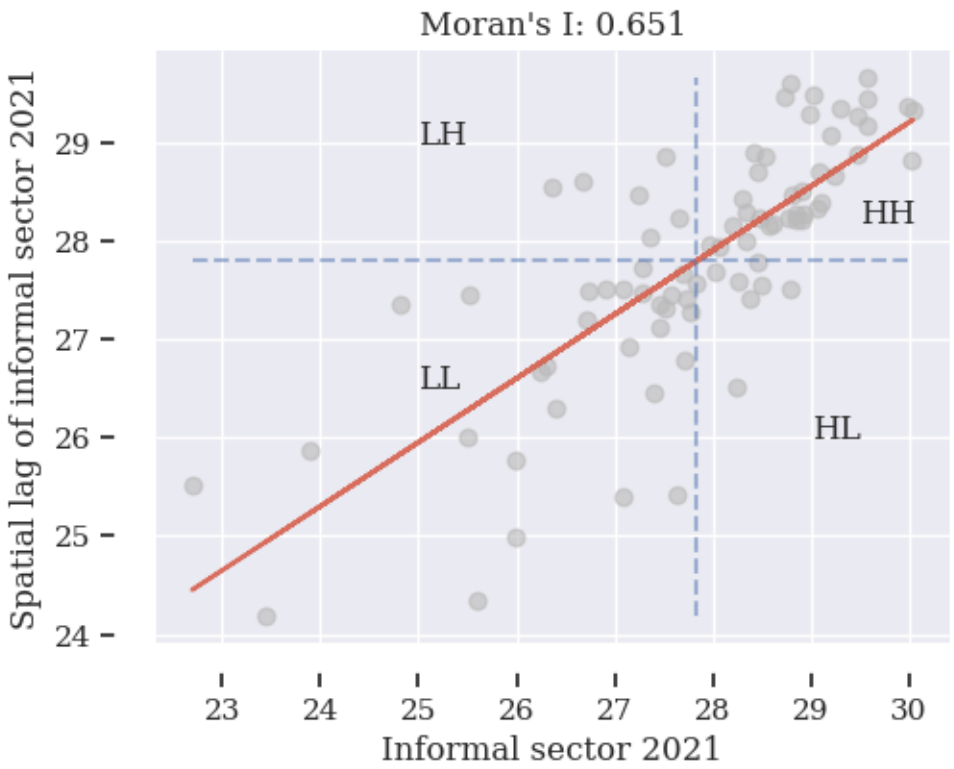
```
0.6673411138948363 0.6516556145477524
0.001 0.001
```

```
fig, ax = plt.subplots(figsize=(6,4))
moran_scatterplot(moran2004, zstandard=False, ax = ax)
ax.set_xlabel("Informal sector 2004")
ax.set_ylabel("Spatial lag of informal sector 2004")
ax.set_title("Moran's I: 0.667")

ax.text(31,30, 'LL')
ax.text(32.5,33.5, 'HH')
ax.text(33,31, 'HL')
ax.text(29,33, 'LH')
```

```
plt.show()
```




```
fig, ax = plt.subplots(figsize=(6,4))
moran_scatterplot(moran2021, zstandard=False, ax = ax)
ax.set_xlabel("Informal sector 2021")
ax.set_ylabel("Spatial lag of informal sector 2021")
ax.set_title("Moran's I: 0.651")

ax.text(25,26.5, 'LL')
ax.text(29.5,28.2, 'HH')
ax.text(29,26, 'HL')
ax.text(25,29, 'LH')


plt.show()
```




```
#Evolution of spatial dependence
```

```
# Create multidimentional array
dfa = merged_data.loc[:,'2004':'2021'].values
dfa
```

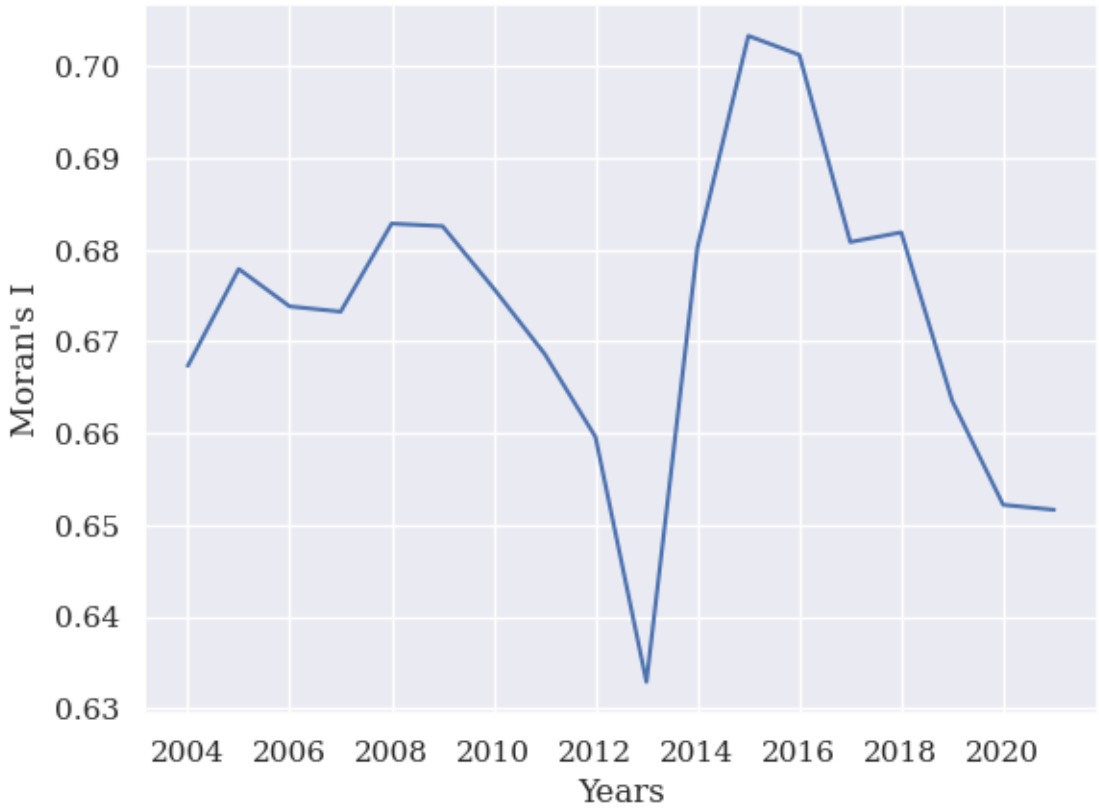
```
array([[32.24650724, 31.68800442, 31.1363712 , ..., 28.03218968,
        27.96208341, 27.96318404],
       [33.74453059, 33.17080523, 32.58001702, ..., 29.37008434,
        29.40589497, 29.46704928],
       [32.55414979, 32.06300928, 31.46635918, ..., 28.10038133,
        28.12354415, 28.3885011 ],
       ...,
       [32.7958902 , 32.16115138, 31.45265665, ..., 28.05074082,
        27.83310122, 27.46828148],
       [33.52618756, 33.04539777, 32.4438166 , ..., 28.94672984,
        28.86962828, 28.98627677],
       [33.60529683, 33.15259018, 32.63714094, ..., 29.88145291,
        29.92967381, 30.03005795]])
```

```
# Create array of years
years = np.arange(2004,2022)
years
```

```
array([2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014,
       2015, 2016, 2017, 2018, 2019, 2020, 2021])
```

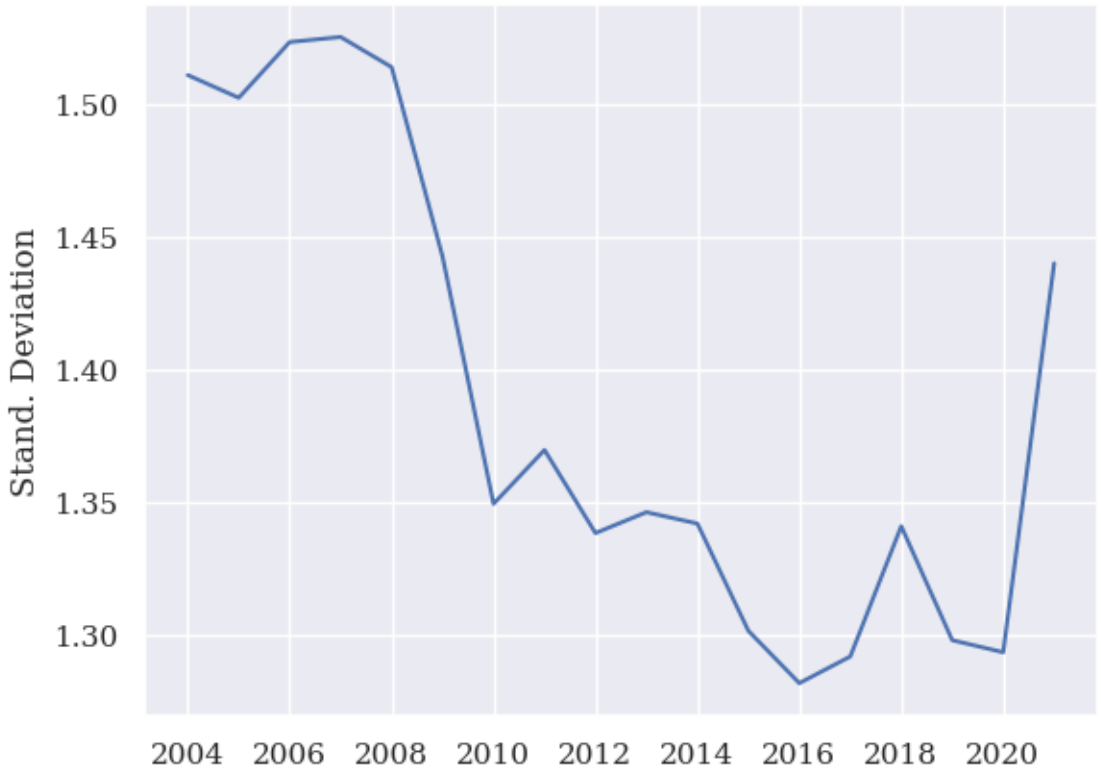
```
mits = [Moran(cs, w_queen) for cs in dfa.T]
res = np.array([(m.I, m.EI, m.p_sim, m.z_sim) for m in mits])
```

```
plt.plot(years, res[:, 0])

# Set integer ticks on the x-axis
plt.xticks(years)
plt.gca().xaxis.set_major_locator(plt.MaxNLocator(integer=True))

# Add labels
plt.xlabel("Years")
plt.ylabel("Moran's I")

# Show the plot
plt.show()
```

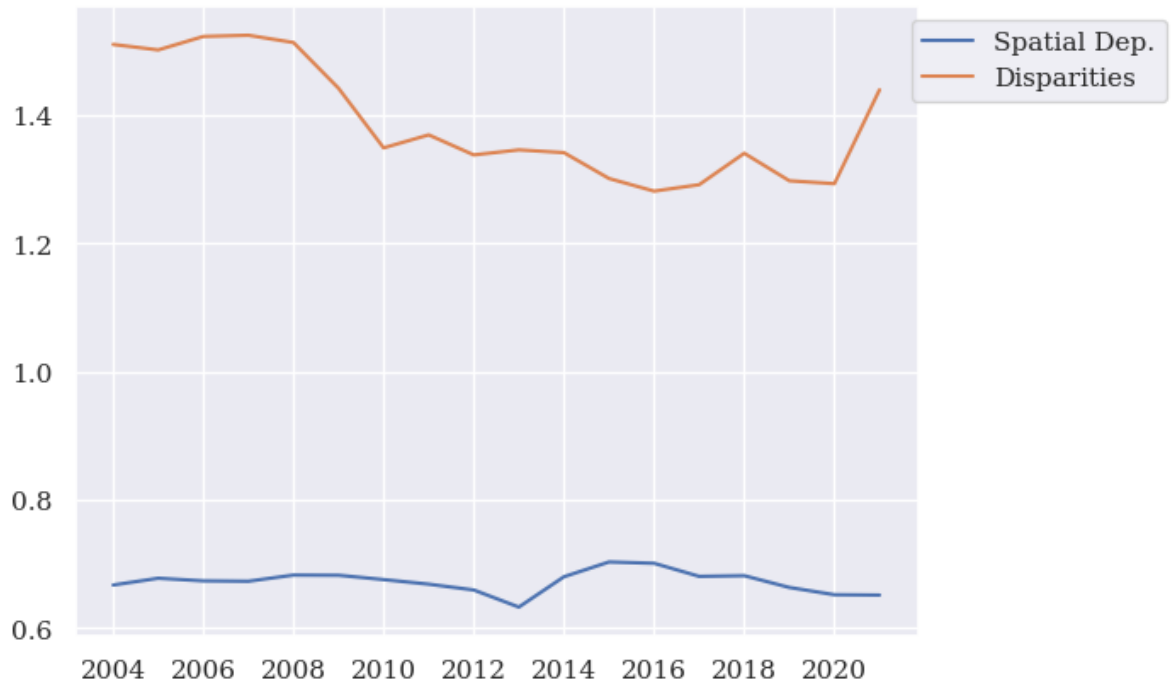



```
### Evolution of regional disparities
```

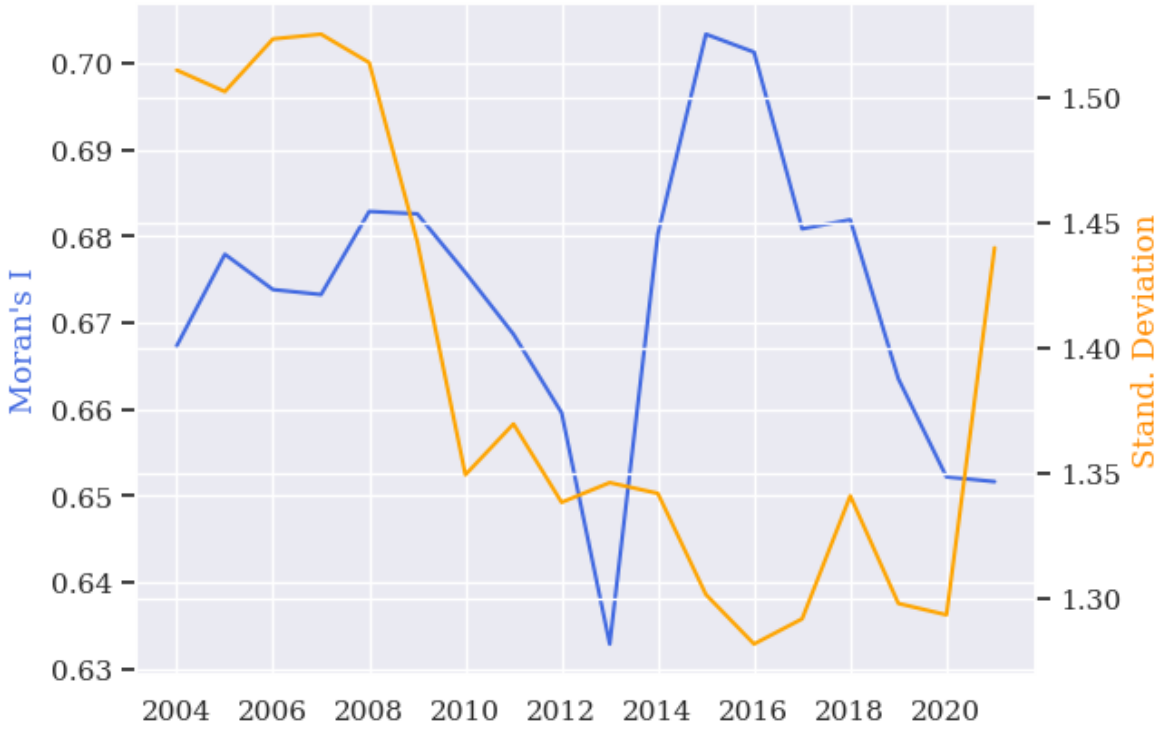
```
sigma = dfa.std(axis=0)
plt.plot(years, sigma)
plt.xticks(years)
plt.gca().xaxis.set_major_locator(plt.MaxNLocator(integer=True))
#plt.title("Sigma Convergence")
plt.ylabel('Stand. Deviation');
```




```
plt.plot(years, res[:,0], label="Spatial Dep.")
plt.plot(years, sigma,  label="Disparities")
plt.xticks(years)
plt.gca().xaxis.set_major_locator(plt.MaxNLocator(integer=True))
plt.legend(bbox_to_anchor=(1.31,1), loc="upper right");
```



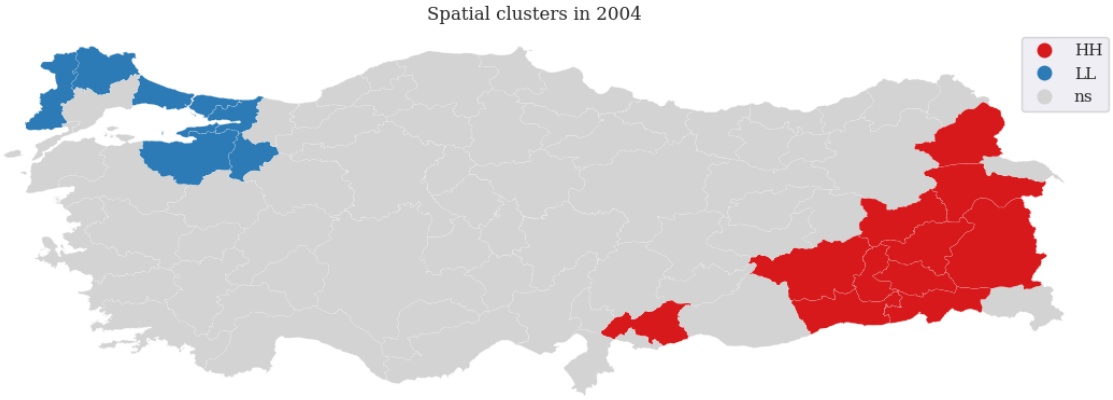

```
# create figure and axis objects with subplots()
fig,ax = plt.subplots()
# make a plot
ax.plot(years, res[:,0], color="royalblue")
# set y-axis label
ax.set_ylabel("Moran's I", color="royalblue")
# twin object for two different y-axis on the sample plot
ax2=ax.twinx()
# make a plot with different y-axis using second axis object
ax2.plot(years, sigma, color="orange")
ax2.set_ylabel("Stand. Deviation", color="darkorange")
plt.xticks(years)
plt.gca().xaxis.set_major_locator(plt.MaxNLocator(integer=True))
plt.show()
```




```
## LISA scatterplot and map
```

```
Lmoran2004 = Moran_Local(x2004, w_queen)
Lmoran2021 = Moran_Local(x2021, w_queen)
```

```
# Plot LISA map
fig, ax = plt.subplots(figsize=(14,12))
lisa_cluster(Lmoran2004, merged_data, p=0.01, figsize = (16,12),ax=ax)
plt.title('Spatial clusters in 2004')
plt.show()
```




```
# Plot LISA map
fig, ax = plt.subplots(figsize=(14,12))
lisa_cluster(Lmoran2021, merged_data, p=0.01, figsize = (16,12),ax=ax)
plt.title('Spatial clusters in 2021')
plt.show()
```