**Geometries**

After understanding the coordinates systems, we can define how geometries can be described using these coordinate systems. Our goal now is to explain:

* **simple features**, a standard that describes point, line, and polygon geometries along with operations on them,
* operations on geometries,
* coverages, functions of space or space-time,
* tesselations, sub-divisions of larger regions into sub-regions, and
* networks.

1. Simple feature geometries:

They are a way to describe the geometries of features. features are defined previously as things that have a geometry, potentially implicitly some time properties, and further attributes that could include labels describing the thing and/or values quantitatively measuring it.

The main application of simple feature geometries is to describe geometries in two-dimensional space by points, lines, or polygons. The “simple” adjective refers to the fact that the line or polygon geometries are represented by sequences of points connected with straight lines that do not self-intersect.

The seven most common simple feature geometry types used to represent a single feature are:



Example:

library(sf) |> suppressPackageStartupMessages() #|> symbol is called the pipe operator in R. It is used to pass the result of an expression on its left-hand side as the first #argument to the function call on its right-hand side. It's a convenient way to chain function calls and enhance code #readability. IT is equivalent to >suppressPackageStartupMessages(library(sf))

**Z and M coordinates**

In addition to X and Y coordinates, single points (vertices) of simple feature geometries may have:

* a Z coordinate, denoting altitude, and/or
* an M value, denoting some “measure”

The M attribute shall be a property of the vertex. It sounds attractive to encode a time stamp in it for instance to pack movement data (trajectories) in LINESTRINGs. These become however invalid (or “non-simple”) once the trajectory self-intersects, which happens when only X and Y are considered for self-intersections. Using only X and Y coordinates (without considering Z or M) is not helpful to represent trajectories or movement paths. Specifically, it's discussing the concept of "self-intersections" in trajectories. In spatial analysis, a "self-intersection" occurs when a trajectory crosses or intersects itself. This can happen, for example, when a moving object changes direction and retraces its path. If we're only considering X and Y coordinates (ignoring Z and M), our analysis may fail to detect or handle these self-intersections properly. Both Z and M are not found often, and software support to do something useful with them is (still) rare.

**Empty geometries**

A very important concept is the empty geometry. Empty geometries arise when we do geometrical operations, for instance when we want to know the intersection of POINT (0 0) and POINT (1 1):

# GEOMETRYCOLLECTION EMPTY

(e <- st\_intersection(st\_point(c(0,0)), st\_point(c(1,1))))

and it represents essentially the empty set: when combining (unioning) an empty point with other non-empty geometries, it vanishes.

**Text and binary encodings**

Part of the simple feature standard are two encodings: a text and a binary encoding. The well-known text encoding (WKT), is human-readable. The well-known binary encoding is machine-readable (WKB). (WKB) encodings, are lossless and typically faster to work with than text encoding (and decoding), and they are used for instance in all communications between R package sf and the GDAL, GEOS, liblwgeom, and s2geometry libraries.

Example :

library(sf)

# Create a simple point geometry

point\_geometry <- st\_point(c(2.0, 3.0))

# Obtain WKT representation

wkt\_representation <- st\_as\_text(point\_geometry)

print("WKT Representation:")

print(wkt\_representation)

# Obtain WKB representation

wkb\_representation <- st\_as\_binary(point\_geometry)

print("WKB Representation:")

print(wkb\_representation)

**Manipulating Objects Common functions to manipulate sf objects include the following:**

* st\_read() reads a sf object,
* st\_write() writes a sf object,
* st\_crs() gets or sets a new coordinate reference system (CRS),
* st\_transform() transforms data to a new CRS,
* st\_coordinates() retrieves coordinates of a sf object,
* st\_as\_sf() converts a foreign object to a sf object.
* st\_intersection() intersects sf objects,
* st\_union() combines several sf objects into one,
* st\_simplify() simplifies a sf object

**Real-Life Application: Urban Planning and Traffic Management**

In urban planning and traffic management, consider that we want to analyze and optimize road intersections in a city.

1. st\_intersection() - Analyzing Intersections:

* We have spatial data representing roads as LineString geometries.
* We use st\_intersection() to find the points where these road geometries intersect, i.e., road intersections.
* This helps us analyze congested or critical intersections, plan traffic signals, and optimize traffic flow by making adjustments based on intersection data.

2. st\_union() - Combining Geometries:

* We have multiple spatial data layers representing different aspects of the transportation system, such as roads, bus stops, and traffic signals.
* We use st\_union() to combine these different sf objects into one, creating a comprehensive representation of the transportation network.
* This combined geometry allows us to perform holistic analysis, plan new routes, and design integrated transportation systems.

3. st\_simplify() - Simplifying Geometries:

* Over time, the spatial data representing road networks might become very detailed and complex.
* We use st\_simplify() to simplify the road geometries, reducing the complexity of the data while maintaining the essential shape and structure.
* Simplified geometries enable faster analysis and visualization without significantly compromising accuracy.

**Handling objects :**

Example :

library(sf)

library(ggplot2)

map <- read\_sf(system.file("shape/nc.shp", package = "sf"))

head(map)

#Delete some of the polygons by taking a subset of the rows of map.

map <- map[-which(map$FIPS %in% c("37125", "37051")), ]

ggplot(map) + geom\_sf(aes(fill = SID79))

#use st\_union(), argument by\_feature = FALSE to combine all geometries together.

ggplot(st\_union(map, by\_feature = FALSE) %>% st\_sf()) + geom\_sf()

# Simplify the boundaries of th map

ggplot(st\_simplify(map, dTolerance = 10000)) + geom\_sf()

Note : In ggplot2, there are various geoms and aesthetics you can use to represent spatial data other than geom\_sf(aes(col = v1)). The choice of geom and aesthetic will depend on the type of spatial data you have and how you want to visualize it. Here are a few alternatives:

1. Points:

* geom\_point(aes(x = x\_column, y = y\_column, col = v1)): Plots points, where x\_column and y\_column are the x and y coordinates.

2. Lines:

* geom\_line(aes(x = x\_column, y = y\_column, col = v1)): Plots lines connecting points, where x\_column and y\_column are the x and y coordinates.

3. Polygons:

* geom\_polygon(aes(x = x\_column, y = y\_column, fill = v1)): Plots filled polygons, where x\_column and y\_column are the x and y coordinates.

4. Paths:

* geom\_path(aes(x = x\_column, y = y\_column, col = v1)): Plots paths, which are lines without points.

Difference between Path and Line?

library(ggplot2)

# Sample data with different order for paths

data <- data.frame(

x = rep(1:10, 2),

y = c(runif(10), runif(10) + 1), # Two sets of y values for demonstration

v1 = rep(c('A', 'B'), each = 10) )

# Shuffle the data for paths

data\_path <- data[sample(nrow(data)),]

# Plot using geom\_line (lines connecting points)

ggplot(data, aes(x = x, y = y, col = v1)) + geom\_line() +

labs(title = "Lines - Connecting Points")

# Plot using geom\_path (paths without explicit points)

ggplot(data\_path, aes(x = x, y = y, col = v1)) + geom\_path() +

labs(title = "Paths - Connecting Observations (Shuffled)")

**How to handle point data?**

Transforming point data into Simple Feature (sf) objects ensures standardization, interoperability, and efficient storage for spatial data. It also enables advanced spatial analysis, visualization, and geospatial operations like : calculating distances, creating buffers (zones around points), conducting overlays (combining different map layers), and performing queries to extract specific information from maps.

In order to work with point data in spatial analysis, we transform it, in R, to a sf object. The st\_as\_sf() function allows to convert a foreign object to a sf object.

For example, we can have a data frame containing the coordinates of a number of locations and attributes that we wish to turn into a sf object. To do that, we use the st\_as\_sf() function passing the object we wish to convert and specifying in argument coords the name of the columns that contain the point coordinates.

In the following, we use st\_as\_sf() to turn a data frame containing coordinates long and lat in addition to two variables : place and value to a sf object. Then, we use st\_crs() to set the coordinate reference system given by the EPSG code 4326 to represent longitude and latitude coordinates.

**Counting the number of points within polygons**

Counting the number of points within polygons is a fundamental operation in spatial data analysis with a range of important applications. It allows for tasks like estimating population density in geographic areas, predicting service demands, identifying crime hotspots, analyzing retail store performance, modeling transportation demand, and managing natural resources effectively.

In R, we can use the st\_intersects() function of sf to count the number of points within the polygons of a sf object. The returned object is a list with feature ids intersected in each of the polygons, and we can use the lengths() function to calculate the number of points inside each feature

Example :

library(sf)

library(ggplot2)

map <- read\_sf(system.file("shape/nc.shp", package = "sf")) # Points over map (simple feature geometry list-column sfc)

points <- st\_sample(map, size = 100) #generate a random sample of 100 points within the boundaries of the map.

# Map of points within polygons

ggplot() + geom\_sf(data = map) + geom\_sf(data = points)

# Intersection (first argument map, then points)

#Uses st\_intersects() to determine which points intersect with the polygons #(features) in the map.It returns a list where each element corresponds to a polygon and contains the IDs of intersecting points

inter <- st\_intersects(map, points)

# Add point count to each polygon

map$count <- lengths(inter)

# Map of number of points within polygons

ggplot(map) + geom\_sf(aes(fill = count))

**Identifying polygons containing points**

Identifying the polygons that contain points is an important operation as it supports numerous spatial operations, analytics, and visualization, providing valuable insights into the geographic distribution and relationships within the dataset.

Given a sf object with points and a sf object with polygons, and we would like to identify the polygons which contain points. This can be done easily, and we can obtain the polygon which each point belongs to using, for example, the st\_intersects() function.

For example, the following code shows a map with the names of the polygons that contain three points over the map.

library(sf)

library(ggplot2)

map <- read\_sf(system.file("shape/nc.shp", package = "sf"))

points <- st\_sample(map, size = 3) %>% st\_as\_sf()

inter <- st\_intersects(points, map)

# Adding column areaname with the name of the areas containing the points

points$areaname <- map[unlist(inter), "NAME", drop = TRUE] # drop geometry (the spatial information) points

#multiple layers

ggplot(map) + geom\_sf() + geom\_sf(data = points) + geom\_sf\_label(data = map[unlist(inter), ], aes(label = NAME), nudge\_y = 0.2)

**Joining map and data**

Sometimes, a map and its corresponding data are available separately and we may wish to create a sf object representing the map with the added data that we can manipulate and plot.

Generally, it is possible to have a map and its corresponding data available separately (any additional information or attributes that are associated with the geographical features on the map). The map itself typically consists of geographical features like points, lines, or polygons that represent geographic locations, boundaries, or routes. The "added data" can include various attributes or properties related to these geographical features. For example, if the map represents regions like counties, the added data could include information such as population, area, demographic data, economic indicators, or any other relevant statistics associated with each county.

Creating an sf (simple features) object by combining the map's geometry (the geographical features) with this additional data allows to have a single object that contains both the spatial information (geometry) and the associated attributes (data). This combined sf object provides a comprehensive representation of the map, making it easier to manipulate, analyze, and visualize the geographic data along with its corresponding attributes.

To create a sf map with the data attributes by joining the map and the data, we use the left\_join() function of the package dplyr.

For example, we add air pollution data obtained with the wbstats package to a sf map obtained with the rnaturalearth package. First, use the ne\_countries() function of rnaturalearth package to download the world map with the country polygons of class sf. Then, use the wbstats package to download a data frame of air pollution data from the World Bank. Specifically, we search the pollution indicators with wb\_search(), and use wb\_data() to download PM2.5 in year 2016 by specifying the indicator corresponding to PM2.5, and the start and end dates.

install.packages("rnaturalearth")

library(rnaturalearth) map <- ne\_countries(returnclass = "sf") #return sf object

install.packages("wbstats")

library(wbstats) # wbstats package is a tool for working with World Bank data

indicators <- wb\_search(pattern = "pollution") #search for indicators that have "pollution" in their names or descriptions

d <- wb\_data(indicator = "EN.ATM.PM25.MC.M3", start\_date = 2016, end\_date = 2016)

install.packages("dplyr")

library(dplyr)

library(ggplot2)

install.packages("viridis")

#For colors

library(viridis)

Now, use the left\_join() from the package dplyr to join the map and the data specifying the argument by the variables we wish to join by. For example, use «ISO3» standard code of the countries rather than the country names, since names can be written differently in the map and the data frame.

map1 <- left\_join(map, d, by = c("iso\_a3" = "iso3c"))

#visualize the spatial features (polygons) from map1 based on the EN.ATM.PM25.MC.M3 values, with a color scale

#indicating levels of PM2.5 pollution (EN.ATM.PM25.MC.M3)

ggplot(map1) + geom\_sf(aes(fill = EN.ATM.PM25.MC.M3)) + scale\_fill\_viridis() + labs(fill = "PM2.5") + theme\_bw()

Note: when using left\_join(), the class of the resulting object is the same as the class of the first argument. So, left\_join(sf\_object, data.frame\_object) creates a sf object, whereas left\_join(data.frame\_object, sf\_object) is a data.frame as :

map1 <- left\_join(map, d, by = c("iso\_a3" = "iso3c"))

d1 <- left\_join(d, map, by = c("iso3c" = "iso\_a3"))

Additionally, given that we have two data frames x and y, we can join them by using the left\_join(), right\_join(), inner\_join() or full\_join() functions.

left\_join(x, y) includes all rows in x so the resulting data frame includes all observations in the left data frame x, whether or not there is a match in the right data frame y.

right\_join(x, y) includes all rows in y so the resulting data frame includes all observations in y, whether or not there is a match in x.

inner\_join(x, y) includes all rows in x and y so the resulting data frame includes only observations that are in both data frames.

full\_join(x, y) includes all rows in x or y so the resulting data frame includes all observations from both data frames.

Another example,

employees <- data.frame(

Employee\_ID = c(1, 2, 3, 4, 5),

Employee\_Name = c("Alice", "Bob", "Charlie", "David", "Eve")

)

salaries <- data.frame(

Employee\_ID = c(2, 4, 5),

Salary = c(60000, 75000, 55000)

)

# Left join employees and salaries based on Employee\_ID

employee\_salaries <- left\_join(employees, salaries, by = "Employee\_ID")

**Raster Data**

As clarified previously, it is commonly used to represent spatially continuous phenomena by dividing the study region into a grid of equally sized rectangles (referred to as cells or pixels) with the values of the variable of interest. In simple terms, raster data uses a grid of squares to measure and record information about things that change across space, like temperature or elevation, which allows us to analyze and visualize how these things vary from one place to another.

Using softwares, ‘terra’ package in R, multilayer raster data is represented with the SpatRaster class. The SpatVector class is used to represent vector data such as points, lines and polygons, and their attributes. In this part, we will handle raster data in order to read, create, and operate it.

The rast() function can be used to create and read raster data, and the writeRaster() function allows to write raster data.

Example : read raster data representing elevation in Luxembourg from the file ‘elev.tif’ from the terra package, and assign it to an object r of class SpatRaster.

install.packages("terra")

library(terra)

pathraster <- system.file("ex/elev.tif", package = "terra")

r <- rast(pathraster)

plot(r)

Additionally, we use rast() to create a SpatRaster object r by specifying the number of columns, the number of rows, as well as the minimum and maximum x and y values.

r <- rast(ncol = 10, nrow = 10, xmin = -150, xmax = -80, ymin = 20, ymax = 60)

r

class : SpatRaster

dimensions : 10, 10, 1 (nrow, ncol, nlyr)

resolution : 7, 4 (x, y)

extent : -150, -80, 20, 60

(xmin, xmax, ymin, ymax)

coord. ref. : lon/lat WGS 84

**Note:** resolution : 7, 4 (x, y): means that each cell (or pixel) in the raster dataset represents a specific area in the geographic space. The "7" in the x-direction and "4" in the y-direction indicate the width and height of each cell in the raster, measured in the units of the coordinate reference system being used (in this case, degrees of longitude and latitude since the coordinate reference is WGS 84). In other words, this raster dataset has cells where each cell covers 7 degrees in the x-direction (longitude) and 4 degrees in the y-direction (latitude). For example, if you have a cell centered at 40 degrees longitude, it might cover the range from 37 degrees to 43 degrees longitude. In the vertical direction (latitude), each cell spans 4 degrees of latitude. For example, if you have a cell centered at 30 degrees latitude, it might cover the range from 28 degrees to 32 degrees latitude.

This resolution defines the size of the cells and affects the level of detail in the data and the accuracy of spatial measurements and analyses.

A l**ower resolution (larger cell size)** may result in a **coarser representation** of the geographic area, while a **higher resolution (smaller cell size)** provides a **finer representation** with more detail.

Moreover, to get the size of the raster :

nrow(r) # number of rows

ncol(r) # number of columns

dim(r) # dimension

ncell(r) # number of cells

To set and access the values of the raster ‘grid’:

values(r) <- 1:ncell(r)

The above raster consists of one layer. To create a multilayer object, we use c() as:

r2 <- r \* r

r2 <- r \* r

names(r) <- "layer1"

names(r2) <- "layer2"

s <- c(r, r2)

then the layers can be subsetted with [[ ]]

as: plot(s[[2]]) # layer 2

**Raster Data Processing "Manipulation": Cropping, masking, and aggregating raster data**

To refine and manipulate raster data in spatial analysis in order to suit the specific needs of a given analysis or visualization, we use the following operations:

**Crop:** the process of extracting a specific geographic region or area of interest from a larger raster dataset. It involves selecting and retaining only the portion of the raster that corresponds to the desired spatial extent, allowing you to focus on a particular region or study area.

**Mask**: the technique of using a mask, typically in the form of another raster(a shape or pattern), to specify which areas of a raster dataset should be retained for further analysis. It involves preserving or removing specific cells or pixels based on the values or criteria in the mask raster.

**Aggregate:** involves reducing the spatial resolution of a raster dataset by combining or summarizing multiple cells or pixels into larger cells or pixels. This process can involve various mathematical operations like averaging or summing, and it helps to simplify the data while preserving key information.

**When to use each technique:**

**Crop:** it’s required to zoom in on a particular area of interest within a larger dataset. For example, if you're studying weather patterns in a specific city, you'd crop the data in the country to cover just that city's boundaries.

**Mask:** is useful when you have a specific criterion or condition to apply to your data. For instance, if you're interested in areas with temperatures above a certain threshold, you'd use a mask to select those areas and exclude the rest.

**Aggregate:** it’s needed to reduce the spatial resolution of the raster data. This technique is beneficial when you have high-resolution data that is too detailed for your analysis or when you want to create a more generalized representation of the data. Aggregating is useful for data smoothing, improving performance, data compression, generalization, visualization, and creating multi-scale representations of your data.

Example: Use the worldclim\_country() function of the ‘geodata’ package to download global temperature dataset from the WorldClim database. Specifically, download the monthly average temperature in degree Celsius by specifying the country (country = "Spain"), the variable mean temperature (var = "tavg"), the resolution (res = 10), and the path where to download the data to as a temporary file (path = tempdir()).

install.packages("geodata")

library(geodata)

library(terra)

r <- geodata::worldclim\_country(country = "Spain", var = "tavg", res = 10, path = tempdir())

plot(r)

Then, it is possible to average the temperature raster data over months with the mean() function #reduces the temporal complexity by **summarizing multiple layers into one layer**

r <- mean(r)

plot(r)

Following, we download the map of Spain with the rnaturalearth package, and delete the Canary Islands region

library(ggplot2)

#install.packages("rnaturalearthhires")

map <- rnaturalearth::ne\_states("Spain", returnclass = "sf")

ggplot(map) + geom\_sf()

map <- map[-which(map$region == "Canary Is."), ] # delete region

ggplot(map) + geom\_sf()

Now, we need to obtain the spatial extent of the map after removing part of it, then we use that spatial extent to crop the temperature raster. So, we keep only the temperature data that corresponds to the spatial extent of mainland Spain, and remove the part of the raster that is outside that spatial extent.

# Cropping

r <- geodata::worldclim\_country(country = "Spain", var = "tavg", res = 10, path = tempdir())

r <- mean(r)

# ext() to obtain the spatial extent of the map

sextent <- terra::ext(map)

r <- terra::crop(r, sextent)

plot(r)

Additionally, we can use the mask() function to convert all values outside the map to NA.

# Masking

r <- geodata::worldclim\_country(country = "Spain", var = "tavg", res = 10, path = tempdir())

r <- mean(r)

# **vect(map)** is used to convert the map object from (sf) format into a format that the **terra::mask** function can work with, as the **mask function expects a vector, data frame or matrix, not an sf object directly.**

r <- terra::mask(r, vect(map))

plot(r)

Finally, to aggregate groups of cells of a raster in order to create a new raster with a lower resolution (i.e., larger cells):

# Aggregating

r <- geodata::worldclim\_country(country = "Spain", var = "tavg", res = 10, path = tempdir())

r <- terra::aggregate(r, fact = 20, fun = "mean", na.rm = TRUE)

plot(r)

The argument fact of aggregate() denotes the aggregation factor expressed as number of cells in each direction (horizontally and vertically), or two integers denoting the horizontal and vertical aggregation factor, like fact = c(10, 15), to control the horizontal and vertical aggregation factors separately. In this case, the first integer (10) determines how many cells will be combined horizontally, and the second integer (15) determines how many cells will be combined vertically.

If we have 90 cell in the horizontal scale and the factor is 10 then the aggregation output will be 9 cells.

Argument fun specifies the function used to aggregate values (e.g., mean).

The last resulted plot above shows a low resolution raster of average annual temperatures in Spain obtained using the aggregate() function.

**Extracting raster values at points**

This technique is used in GIS and spatial analysis to retrieve information from a raster dataset (e.g., satellite imagery, elevation data, land cover) at specific point locations on the Earth's surface. It is essential for various applications and is important for several reasons:

* Data Relevance: Raster datasets often cover large areas and contain a wealth of information. However, when you have specific points of interest, such as weather stations, it's important to extract data relevant to those points. This allows you to focus on the data that directly pertains to your study locations.
* Spatial Analysis: For many spatial analyses, it's essential to have data at specific point locations. For example, you might want to perform statistical analyses, geostatistical modeling, or create spatial patterns within a specific set of locations. Extracting values at points provides you with the necessary data for such analyses.
* Integration: Point data extracted from raster values can be integrated with other datasets, allowing you to combine information from different sources. This integration is valuable for various applications, including environmental monitoring, site suitability analysis, and geospatial modeling.
* Visual Representation: Point data extracted from raster values can be used for creating maps or visualizations that highlight specific locations of interest. This can help in presenting your findings effectively.

Indeed, the technique typically involves matching the coordinates of point locations with the corresponding cells in the raster dataset and retrieving the values at those locations.

Given a raster of class SpatRaster, we can extract the raster values at a set of points with the extract() function of terra. Here, is an example of the use of extract() using a raster representing the elevation of Luxembourg, and a vector file with the divisions of Luxembourg from files in the terra package.

The R package terra allows to extract the raster values at a set of points with the extract() function from a raster of class SpatRaster.

Example : Use extract() using a raster representing the elevation of Luxembourg, and a vector file with the divisions of Luxembourg from files in the terra package.

library(terra)

# Raster (SpatRaster)

r <- rast(system.file("ex/elev.tif", package = "terra"))

# Polygons (SpatVector)

v <- vect(system.file("ex/lux.shp", package = "terra"))

points <- crds(centroids(v)) #centroids() to obtain the centroids of polygons, and crds() to obtain their coordinates.

plot(r)

plot(v, add = TRUE)

points(points)

The resulted plot shows the elevation raster, the polygons, and the points of the centroid locations.

Following, to obtain the values of the raster at «points» using extract(), fill as first argument the SpatVector object with the raster, and as second argument a data frame with the points.

# data frame with the coordinates

points <- as.data.frame(points)

valuesatpoints <- extract(r, points)

#r: from which you want to extract information

#points: the specific locations at which you want to retrieve information from the raster ‘r’ cbind(points, valuesatpoints)

**Extracting and averaging raster values**

As presented previously, extracting raster values at points gives the values at specific locations. On the other hand, extracting and averaging raster values, provides the average values over larger areas or groups of cells within the raster.

This process **combines** the values from **multiple cells within the raster.** Instead of focusing on individual points, you take a group of neighboring cells and compute their average value. This process effectively reduces the resolution or level of detail in the raster.

The result of this process is a new raster layer, where each cell represents an average value based on the original raster's values within a certain neighborhood.

The function extract() also used to obtain the values of raster objects of class SpatRaster within polygons of class SpatVector. Cells extracted within each polygon are cells that have centers covered by the polygon.

For the function extract(), set the argument **weights = TRUE** to get, apart from the cell values, the percentage of each cell covered by the polygon, and this can be used to compute **area-weighted averages**. In other words, when you want to compute averages in a way that considers the area. For example, if the average of temperatures within a polygon is calculated, it might be required to give more importance to larger portions of cells within the polygon. Using the percentage of coverage helps you do this. So, by setting weights = TRUE, you can get not only the values but also information about how much each cell contributes to the area of your polygon.

The argument ‘fun’ can be used to specify a function (e.g., mean) that summarizes the extracted values by polygon.

For last example, the process of extracting and averaging raster values within polygons using the extract() function:

# Extracted raster cells within each polygon

head(extract(r, v, na.rm = TRUE))

#na.rm = TRUE if there NA values in the raster cells within a polygon, they will be excluded from the calculation.

# Extracted raster cells and percentage of area # covered within each polygon

head(extract(r, v, na.rm = TRUE, weights = TRUE))

# Average raster values by polygon are obtained as

v$avg <- extract(r, v, mean, na.rm = TRUE)$elevation

#To obtain the area-weighted average raster values by polygon, we use extract() with weights = TRUE

# Area-weighted average raster values by polygon

v$weightedavg <- extract(r, v, mean, na.rm = TRUE, weights = TRUE)$elevation

# Plot average raster values within polygons

library(ggplot2)

library(tidyterra)

ggplot(data = v) + geom\_spatvector(aes(fill = avg)) + scale\_fill\_terrain\_c()

# Plot area-weighted average raster values within polygons

ggplot(data = v) + geom\_spatvector(aes(fill = weightedavg)) + scale\_fill\_terrain\_c()

The resulted plot shows maps of the elevation averages and area-weighted averages in each of the Luxembourg divisions.

**Areal Data**

Areal «or lattice» data arise when a study region is partitioned into a finite number of areas at which outcomes **are aggregated**. Examples of areal data are the number of individuals with a certain disease in municipalities of a country, the number of road accidents in provinces, or the average housing prices in districts of a city.

**Spatial neighborhood matrices**

The concept of spatial neighborhood is useful for the exploration of areal data to **assess spatial autocorrelation** and find out whether close areas have similar or dissimilar values. Spatial neighbors can be defined in several ways **depending** on the **variable of interest**. The **simplest** neighborhood definition assumes that neighbors are **areas** that **share a common border**, perhaps a vertex. We can also expand the idea of neighborhood to include areas that are close, but not necessarily adjacent, by assuming neighbors are areas that are within some distance apart.

Given a spatial neighborhood definition, we can construct a spatial neighborhood matrix which will allow us later to assess the spatial autocorrelation.

The elements of the spatial neighborhood matrix can be viewed as weights that spatially connect areas. In this matrix, entries corresponding to close areas will have more weight than entries corresponding to areas that are farther apart.

Using a software, the **spdep** package contains a number of functions to deal with spatial dependence structures. It has functions that can be used to construct **spatial neighborhood matrices** and perform spatial autocorrelation analyses. For example, the functions poly2nb() and dnearneigh() can be used to create **neighbor lists** based on **contiguity and distance criteria, respectively.** Following, s**patial neighborhood matrices** can be built from the **neighbor lists** using the nb2listw() function.

Example :

Use the spdep package to learn how to construct several types of spatial neighborhood structures and matrices. From the package spData, read the map of the 49 neighborhoods of Columbus, Ohio, which is in the columbus shapefile.

install.packages("spData")

install.packages("spdep")

library(spData)

library(sf)

library(spdep)

library(ggplot2)

map <- st\_read(system.file("shapes/columbus.shp", package = "spData"), quiet = TRUE) # quiet = TRUE to prevent displaying additional information or warning messages

**1. Neighbors based on contiguity**

\* In summary: Zone i and j are neighbors if zone i is contiguity or adjacent to zone j

Neighbors based on contiguity are constructed by assuming that neighbors of a given area are other areas that share a common boundary. The following figure shows two types of contiguity neighbors.

a. Neighbors can be of type Queen if a single shared boundary point meets the contiguity condition.

b. Rook if more than one shared point is required to meet the contiguity condition.





# construct a list of neighbors based on areas with contiguous boundaries using the function poly2nb()

# poly2nb() accepts a list of polygons and returns a list of class nb with the neighbors of each area

# the default type in poly2nb() is queen = TRUE

nb <- spdep::poly2nb(map, queen = TRUE)

nb

#Neighbour list object:

#Number of regions: 49

#Number of nonzero links: 236

#Each link represents a connection between two neighboring regions. A nonzero link means that there's a connection

#between these regions. In other words, it tells us how many regions are adjacent to each other.

#Percentage nonzero weights: 9.829238 #indicates that around 9.83% of the possible pairs of regions are actually neighbors and share a boundary or edge = 236/(49^2 = 2,401)

#Average number of links: 4.816327 #This tells you how many neighboring regions, on average, each region is connected to.

head(nb)

To plot a map with the neighbors obtained: First plot the map, and then overlap the neighborhood structure using the function plot.nb() by passing the neighbor list and the coordinates of the map.

plot(st\_geometry(map), border = "lightgray") # st\_geometry() is used to extract the geometry column

plot.nb(nb, st\_geometry(map), add = TRUE) #plot.nb() is used to visualize a neighbor list object (nb) in spatial analysis.

Additionally, it is possible to plot the neighbors of a given area by adding a new column in map representing the neighbors of the area. For example, to show the neighbors of area 20 :

id <- 20 # area id

map$neighbors <- "other"

map$neighbors[id] <- "area"

map$neighbors[nb[[id]]] <- "neighbors"

ggplot(map) + geom\_sf(aes(fill = neighbors)) + theme\_bw() + scale\_fill\_manual(values = c("gray30", "gray", "white"))

Finally, we can count the number of neighbors of each area and build a table with the areas’ ids in rows, and the number of neighbors in columns.

Given a neighbor list, the cardinality function spdep::card() counts the number of neighbors of each area. We can obtain the number of neighbors of each area with lengths(nb). Then, we can use table(1:nrow(map), card(nb)) to build the table.

**2. Neighbors of order k based on contiguity**

It might be interesting to find the kth order (1st, 2nd) neighbors based on contiguity.



How to know if the polygon is hexagon:

geometry\_type <- st\_geometry\_type(map)

# Print the geometry type

print(geometry\_type)

In R, the nblag() function of the spdep package is used to create higher order neighbor lists.

Higher-order neighbors are neighbors of neighbors, essentially extending the concept of spatial relationships beyond immediate neighbors.

The arguments of this function are a neighbors list of class nb, and the **maximum lag considered**. The returned object is a list of lagged neighbors lists each with class nb.

For example, use nblag() with the maximum lag equal to maxlag = 2 to create a list containing a list of neighbors of order 1, and a list of neighbors of order 2.

library(spdep)

nb <- poly2nb(map, queen = TRUE)

nblags <- spdep::nblag(neighbours = nb, maxlag = 2)

# Neighbors of first order

plot(st\_geometry(map), border = "lightgray")

plot.nb(nblags[[1]], st\_geometry(map), add = TRUE)

# Neighbors of second order

plot(st\_geometry(map), border = "lightgray")

plot.nb(nblags[[2]], st\_geometry(map), add = TRUE)

Additionally, the function nblag\_cumul() of spdep cumulates a list of lagged neighbors as the output of nblag() and returns a single neighbor list of class nb containing the neighbors of order 1 until the maximum lag considered.

# Neighbors of order 1 until order 2

nb <- spdep::poly2nb(map, queen = TRUE)

nblagsc <- spdep::nblag\_cumul(nblags)

plot(st\_geometry(map), border = "lightgray")

plot.nb(nblagsc, st\_geometry(map), add = TRUE)

**3. Neighbors based on k nearest neighbors**

It is also possible to consider as neighbors of an area its k nearest neighbors based on the distance separating them.



In R, the function knearneigh() of spdep allows to obtain a matrix with the indices of points belonging to the set of the k nearest neighbors of each area. The arguments of this function include a matrix of point coordinates and the number k of nearest neighbors to be returned.

Then, it is possible to use knn2nb() to convert this list into a neighbor list of class nb with the integer vectors containing the ids of the neighbors.

# Neighbors based on 3 nearest neighbors

coo <- st\_centroid(map)

nb <- knn2nb(knearneigh(coo, k = 3))

# k number nearest neighbors

#Euclidean distance by default

# if data is in geographical coordinates (latitude and longitude), use great-circle distance (haversine distance) instead of Euclidean distance, so set **longlat = TRUE** to use great-circle distance.

plot(st\_geometry(map), border = "lightgray")

plot.nb(nb, st\_geometry(map), add = TRUE)

**4. Neighbors based on distance**

In summary: zone i and j are neighbors if the distance between them are less than the threshold distance. It is also possible to define neigborhood structures by considering neighbors areas that are within a given distance.



The dnearneigh() function of spdep builds a list of neighbors based on a distance between specific lower and upper bounds. The arguments of dnearneigh() include the object with the point coordinates, and the lower and upper distance bounds (d1 and d2).

library(spData)

library(sf)

library(spdep)

library(ggplot2)

map <- st\_read(system.file("shapes/columbus.shp", package = "spData"), quiet = TRUE)

# quiet = TRUE to prevent displaying additional information or warning messages

nb <- dnearneigh(x = st\_centroid(map), d1 = 0, d2 = 0.4) #0.4 is a distance threshold in the same units as the crs of data

#Euclidean distance by default, use use\_s2=FALSE as recommended in the R documentation

plot(st\_geometry(map), border = "lightgray")

plot.nb(nb, st\_geometry(map), add = TRUE)

Further, it is possible to determine an appropriate upper distance to ensure that each area has at least k neighbors. To determine an appropriate upper distance bound for a given number of neighbors k:

First, use the spdep::knearneigh() function to obtain the k nearest neighbors of each of the areas. We use this function passing a matrix of point coordinates and the number k of chosen nearest neighbors to obtain a matrix with the k nearest neighbors of each area.

Then, use spdep::knn2nb() to convert this list into a neighbor list of class nb with the ids of the neighbors. In the following example, we determine the upper distance bound to ensure all areas have at least one neighbor.

coo <- st\_centroid(map) # k is the number nearest neighbors

nb1 <- knn2nb(knearneigh(coo, k = 1))

Then, use spdep::nbdists() passing the nb list with the neighbors and the matrix of point coordinates to obtain the distances along the links. Finally, compute summaries to determine the upper distance bound.

dist1 <- nbdists(nb1, coo) summary(unlist(dist1))

 Min. 1st Qu. Median Mean   3rd Qu. Max.

0.128 0.254 0.316 0.329      0.404           0.619

So, in the above example, the maximum distance is 0.62 and we can consider this value as an upper bound of the distance to ensure each area has at least one neighbor.

**Neighborhood matrices**

A spatial neighborhood matrix W defines a neighborhood structure over the entire study region, and its elements can be viewed as weights. The (i,j)th element of W, denoted by wij, spatially connects areas i and j in some fashion. More weight is associated with areas closer to i than those farther away from i.



As shown above, if neighbors are based on contiguity, we can construct a binary spatial matrix with wij=1 if regions i and j share a common boundary, and wij=0 otherwise. Indeed, wii is set to 0 for i=1,…,n. Note that this choice of proximity measure results in a symmetric spatial matrix.



Other spatial weight definitions could be to use wij=1 for all i and j within a specified distance, or to use wij=1 if j is one of the k nearest neighbors of i. Weights wij can also be defined as the inverse distance between areas. Note that in most situations, this matrix is not symmetric when the areas are irregularly shaped.

In addition, we may want to adjust for the total number of neighbors in each area and use a standardized matrix with entries wstd,i,j= wij / ∑j=1 wij.

**Great Circle Formula:**

****

**General Spatial Weights Based on Distance:**

****

**Spatial weights matrix based on a binary neighbor list**

Binary Weight Matrix:

In a binary weight matrix, the weights are binary values : 0 or 1.

It is used to represent whether a connection or relationship exists between areas:1 for "connected" and 0 for "not connected".

This style is commonly used in network analysis to represent binary relationships in networks, such as social networks where connections between individuals are either present (1) or absent (0).

Using R packages, the function nb2listw() of the spdep package can be used to construct a spatial neighborhood matrix containing the spatial weights corresponding to a neighbors list. The neighbors can be binary or based on inverse distance values. To compute a spatial weights matrix based on a binary neighbor list, we use the nb2listw() function with the following arguments:

* nb: list with neighbors,
* style: indicates the coding scheme chosen. For example, style = B is the basic binary coding, and W is row standardized (1/number of neighbors),
* zero.policy is used to take into account regions with 0 neighbors. Specifically, zero.policy = TRUE permits the weight list to contain zero-length weights vectors, and zero.policy = FALSE stops the function with an error if there are empty neighbor sets.

nb <- poly2nb(map, queen = TRUE)

nbw <- spdep::nb2listw(nb, style = "W")

nbw$weights[1:3]

We can visualize the spatial weight matrix by creating a matrix with the weights with listw2mat(), and using lattice::levelplot() to create the plot.

m1 <- listw2mat(nbw)

lattice::levelplot(t(m1), scales = list(y = list(at = c(10, 20, 30, 40), labels = c(10, 20, 30, 40))))

**Spatial weights matrix based on inverse distance values**

Weighted Weight Matrix: In a weighted weight matrix, the weights are real-valued numbers, often representing the strength, intensity, or some measure of the relationship between entities.

It is used to quantify the strength or degree of connection between entities, allowing for another representation.

This style is common in applications like data analysis, where relationships between entities can vary in strength, such as representing distances, similarities, or correlation coefficients.

For a list of neighbors in R packages, it is possible to use **nbdists()** to compute the distances along the links. Then, we can construct the list with spatial weights based on inverse distance values using nb2listw() where the argument glist is equal to a list of general weights corresponding to neighbors.

coo <- st\_centroid(map)

nb <- poly2nb(map, queen = TRUE)

dists <- nbdists(nb, coo) # stores in a matrix the distances between neighbors based on the coordinates of centroids.

ids <- lapply(dists, function(x){1/x}) # to find the inverse of a distance

nbw <- nb2listw(nb, glist = ids, style = "B") #construct the list of weights as binary style nbw$weights[1:3] # see the list of neighbors for the 1st three polygons

m2 <- listw2mat(nbw) #levelplot to to visualize spatial patterns and the strength of connections between locations

#levelplot expects the matrix to be in the form of columns representing different variables or categories, so the rows of #the matrix to represent locations (polygons or points) and the columns to represent neighbors or connected

lattice::levelplot(t(m2), scales = list(y = list(at = c(10, 20, 30, 40),labels = c(10, 20, 30, 40))))

**Spatial Autocorrelation**

**Tobler's First Law** of Geography: is a fundamental principle in the field of spatial analysis. It **states that: Everything is related to everything else, but near things are more related than distant things.**

From a **statistical point of view**, **spatial autocorrelation** is used to describe the **extent** to which a variable is correlated with itself through space.

Positive spatial autocorrelation occurs when observations with similar values are closer together (clustered). Negative spatial autocorrelation occurs when observations with dissimilar values are closer together (dispersed). The following plot shows three configurations of areas showing different types of spatial autocorrelation :



Spatial autocorrelation can be assessed using indices that summarize the degree to which **similar observations tend to occur near each other over the study area**.

Two common indices that are used to assess spatial autocorrelation in areal data are Moran’s I and Geary’s C.

**Global Moran’s**

I It is defined as:



where **n** is the **number of regions**, **Yi** is the **observed value** of the variable of interest in region i, and **Y\_bar** is the **mean** of all **values**. **wij** are **spatial weights** that denote the **spatial proximity between regions i and j**, with wii=0 and i,j=1,…,n.

The definition of the spatial weights depends on the variable of study and the specific setting.

We can **test the presence of spatial autocorrelation** using the **Moran’s I**, which **quantifies how similar each region is with its neighbors and averages all these assessments.**

Under the **null hypothesis** of **no spatial autocorrelation**, observations Yi are **independent identically distributed**, and **I is asymptotically normally distributed** with mean and variance equal to:



S0: quantifies the overall variability in the variable across all locations **without** considering their **spatial relationships**. It is the **global variance of the variable**, treating all locations equally

S1: quantifies how the **values** of **a location are related to the values of its neighbors.** So it **indicates** the degree of positive or negative **spatial autocorrelation**.

S2: quantifies the **sum** of the **spatial weights** for all pairs of locations. It essentially **quantifies** the **total strength of spatial relationships in the data**

Moran’s I values usually range from –1 to 1. Moran’s I values significantly **above E[I]=−1/(n−1)** indicate **positive spatial autocorrelation** or **clustering**. This occurs when neighboring regions tend to have similar values. Moran’s I values significantly **below E[I]** indicate **negative spatial autocorrelation or dispersion**. This happens when regions that are close to one another tend to have different values. Finally, Moran’s I values **around E[I]** indicate **randomness**, that is, **absence of spatial pattern.** When the **number** of **regions is sufficiently large**, **I** has a **normal distribution** and we can assess whether any given pattern deviates significantly from a random pattern by comparing the z-score



to the standard normal distribution.

An alternative approach to **judge significance** is **Monte Carlo randomization.** This method creates random patterns by reassigning the observed values among the areas and calculates the Moran’s I for each of the patterns, providing a randomization distribution for the Moran’s I. If the observed value of **Moran’s I** lies in the **tails** of this distribution, the **assumption** of **independence among observations is rejected**. Thus, we can test spatial autocorrelation by following these steps:

1. State the null and alternative hypotheses:

H0:I=E[I] (no spatial autocorrelation),

H1:I≠E[I] (spatial autocorrelation).

2. Choose the significance level α we are willing to tolerate, which represents the maximum value for the probability of incorrectly rejecting the null hypothesis when it is true (usually α=0.05).

3. Calculate the test statistic:



4. Find the p-value for the observed data by comparing the z-score to the standard normal distribution or via Monte Carlo randomization.

5. Make one of these two decisions and state a conclusion:

If **p-value < α**, we **reject** the **null hypothesis**. We conclude data provide evidence for the alternative hypothesis.

If **p-value** **≥ α**, we **fail** to **reject** the **null hypothesis**. **The data do not provide evidence for the alternative hypothesis.**

Using R packages, the function **moran.test()** of the **spdep** package can be used to test spatial autocorrelation using Moran’s I. The arguments of moran.test() are a numeric vector with the data, a list with the spatial weights, and the type of hypothesis.

The argument that denotes the hypothesis is called alternative and can be set equal to greater (default), less or two.sided to represent a different alternative hypothesis.

Example: Use the Moran’s I to test the spatial autocorrelation of housing prices in 506 census tracts in Boston, USA, in 1978 obtained from the spData package. The data has a variable called MEDV with the median prices of owneroccupied housing in $1000 USD.

First, create the variable vble with the values of MEDV that will be used in the analysis.

Following, plot the map which shows with the housing prices using mapview package. library(spData)

library(sf)

library(mapview)

map <- st\_read(system.file("shapes/boston\_tracts.shp", package = "spData"), quiet = TRUE)

map$vble <- map$MEDV

mapview(map, zcol = "vble")

Then, specify the null and alternative hypothesis as follows:

H0: I ≤ E[I] (negative spatial autocorrelation or no spatial autocorrelation),

H1: I > E[I] (positive spatial autocorrelation).

Then we use moran.test() to test this hypothesis by setting alternative = "greater". The list with the spatial weights is calculated by first obtaining the neighbors of each area with the poly2nb() function, and then creating a list containing the neighbors with the nb2listw() function of spdep as clarified last two lectures.

# Neighbors

library(spdep)

nb <- poly2nb(map, queen = TRUE)

# queen shares point or border

nbw <- nb2listw(nb, style = "W")

# Global Moran's I

gmoran <- moran.test(map$vble, nbw, alternative = "greater")

gmoran

Moran I test under randomisation

data: map$vble

weights: nbw

Moran I statistic

standard deviate = 23, p-value < 2e-16

alternative hypothesis: greater

sample estimates:

Moran I statistic Expectation Variance

0.6266754 -0.0019802 0.0007249

Moran I Statistic Standard Deviate: this value helps to assess whether there is spatial autocorrelation in data. As 23.35 is quite large, suggesting a strong indication of spatial autocorrelation.

P-value: (p-value < 2.2e-16). A low p-value indicates strong evidence against the null hypothesis, so we would reject Ho and conclude that there is indeed significant spatial autocorrelation in data.

Alternative Hypothesis: H1 specified is "greater," suggesting that the spatial autocorrelation is in a positive direction.

The object returned by moran.test() provides the Moran’s I statistic, the z-score and the p-value. We observe the p-value obtained is lower than the significance level 0.05. Then, we reject the null hypothesis and conclude there is evidence for positive spatial autocorrelation.

The same conclusion is obtained if Monte Carlo approach is used to assess significance. This approach creates random patterns by reassigning the values among the fixed areas and calculates the Moran’s I for each of these patterns and calculates the p-value.

For our example, conduct a Monte Carlo randomization approach using the moran.mc() function setting the number of simulations to nsim = 999.

gmoranMC <- moran.mc(map$vble, nbw, nsim = 999)

gmoranMC

Monte-Carlo simulation of Moran I

data: map$vble

weights: nbw

number of simulations + 1: 1000

statistic = 0.63, observed rank = 1000, p-value= 0.001

alternative hypothesis: greater It is observed that p-value is lower than 0.05 indicating that the data presents positive spatial autocorrelation.

Additionally, we can plot the histogram of the Moran’s I values for each of the simulated patterns, as well as the Moran’s I obtained with the real data.

hist(gmoranMC$res)

abline(v = gmoranMC$statistic, col = "red")

**Moran’s I scatterplot**

Moran’s I scatterplot is used to visualize the spatial **autocorrelation** in the data. This plot displays the **observations** of each area **against its spatially lagged values.** The spatially **lagged value** for a given area is calculated as a **weighted average of the neighboring values for that area**.

The moran.plot() function can be used to construct the scatterplot, and the spatially lagged value can be computed using the lag.listw() function of spdep by passing the values and the spatial weights corresponding to the areas.

Example: Construct the Moran’s I scatterplot for the housing prices data used moran.plot() function by passing the housing prices and the spatial weights corresponding to the Boston tracts.

moran.plot(map$vble, nbw)

You can observe a positive linear relationship between the observations and their spatially lagged values. Using this plot, we can also identify data points that have a high influence on the linear relationship between the data and the lagged values.

**Local Moran’s I**

We have used the global Moran’s I to provide an index to assess the spatial autocorrelation for the whole study region. There is often interest in providing a local measure of similarity between each area’s value and those of nearby areas.

**Local Indicators of Spatial Association (LISA)** are designed to provide an indication of the extent of significant spatial clustering of similar values around each observation. A desirable property is that the sum of the LISA’s values across all regions is equal to a multiple of the global indicator of spatial association. As a result, global statistics may be decomposed into a set of local statistics and most LISAs are defined as local versions of well-known global indexes.

Types of LISA:

* High-High (HH): Locations with high values surrounded by other locations with high values.
* Low-Low (LL): Locations with low values surrounded by other locations with low values.
* High-Low (HL): Locations with high values surrounded by other locations with low values.
* Low-High (LH): Locations with low values surrounded by other locations with high values.

Interpretation: **HH and LL** clusters indicate areas of **positive spatial autocorrelation**, while **HL and LH** clusters suggest areas of **negative spatial autocorrelation** One of the most popular LISAs is the local version of Moran’s I. For the ith region, the local Moran’s I defined as



It is important to know that the global Moran’s I is proportional to the sum of the local Moran’s I obtained for all regions:



The values of the LISAs are mapped to indicate the location of areas with comparatively high or low local association with neighboring areas. A **high** value for **Ii** suggests that the **area** is **surrounded** by areas with **similar values**. Such an area is part of a cluster of high observations, low observations, or moderate observations. A **low** value for **Ii** indicates that the area is **surrounded** by areas with **dissimilar** values. Such an area is an outlier indicating that the observation of area i is different from most or all of the observations of its neighbors.

To interpret the significance of local Moran’s I for each of the areas, it is necessary to compute a map of p-values which is calculated assuming the null hypothesis(states that there is no spatial autocorrelation) is true to know if our observations are likely due to random chance or if they're statistically meaningful.

These p-values, regardless of the presence or absence of global spatial association, may be obtained by a simulation process with a **conditional randomization approach.** In this approach, the observed value Yi at region i is fixed, and the remaining values are randomly reassigned over the other regions.

To assess the local autocorrelation in R, the localmoran() function of the spdep package can be used to compute the Local Moran’s I for the given dataset. The arguments of localmoran() include a numeric vector with the values of the variable, a list with the neighbor weights, and the name of an alternative hypothesis that can be set equal to greater (default), less or two.sided. The returned object of the localmoran() function contains the following information:

* Ii: Local Moran’s I statistic for each area,
* E.Ii: Expectation Local Moran’s I statistic,
* Var.Ii: Variance Local Moran’s I statistic,
* Z.Ii: z-score,
* Pr(z > E(Ii)), Pr(z < E(Ii)) or Pr(z != E(Ii)): p-value for an alternative hypothesis greater, less or two.sided, respectively.

Example:

For the housing prices data, use the localmoran() function to compute the Local Moran’s I. Set alternative = "greater" which corresponds to testing H0: no or negative spatial autocorrelation vs. H1: positive spatial autocorrelation.

lmoran <- localmoran(map$vble, nbw, alternative = "greater")

head(lmoran)

Ii E.Ii Var.Ii Z.Ii Pr(z > E(Ii))

1 -0.3457508 -5.254e-04 3.275e-02 -1.9075             9.718e-01

2 0.0175875 -1.627e-05 2.046e-03 0.3892               3.486e-01

3 0.0123380 -6.557e-07 4.090e-05 1.9294               2.684e-02

4 -0.0001654 -1.059e-07 1.332e-05 -0.0453            5.181e-01

5 0.3591629 -1.428e-04 7.899e-03 4.0428              2.641e-05

6 0.0545611 -1.626e-04 1.357e-02 0.4697              3.193e-01

* Ii (Local Moran's I): the calculated Local Moran's I statistic for each region which measures the degree of spatial autocorrelation for each observation. Positive values indicate clustering of similar values, negative values indicate dispersion or dissimilarity, and values near zero suggest randomness.
* E.Ii (Expected Local Moran's I): the expected value of Local Moran's I under H0. It represents what you would expect if there were no spatial autocorrelation.
* Var.Ii (Variance of Local Moran's I): the variance of the Local Moran's I statistic, providing a measure of how much the observed values deviate from what would be expected under H0.
* Z.Ii (Z-score): indicates how many standard deviations an observed Local Moran's I value is from the mean of the expected values. **Positive Z-scores suggest significant positive spatial autocorrelation, while negative Zscores suggest significant negative spatial autocorrelation.**
* Pr(z > E(Ii)): the p-value associated with each Z-score. **Smaller** p-values (typically < 0.05) **suggest rejection of the null hypothesis.**

**Generally, significant positive Z-scores (with low p-values) indicate regions with significant positive spatial autocorrelation, while significant negative Z-scores indicate regions with significant negative spatial autocorrelation. Non-significant Z-scores suggest spatial randomness.**

According to the example:

**Observation 1:** There is a negative (as Z score is negtive) spatial autocorrelation in this area, suggesting that the region is surrounded by dissimilar values. the p-value is not significant (0.9718), indicating that the observed negative spatial autocorrelation is not statistically meaningful. The lack of significance suggests that the observed dissimilarity could reasonably occur by random chance, and it's not a reliable indication of a true spatial pattern.

**Observation 2:** positive spatial autocorrelation but not significant as P-Value (0.3486) not significant. So no significant spatial autocorrelation detected in this area.

**Observation 3:** Positively significant Z-score : Z-Score (1.9294) and P-Value (0.0268): Significant. Accordingly, there is a positive spatial autocorrelation in this area, indicating that the region is part of a cluster with similar values.

**Observation 4:** Z-Score (-0.0453): negative spatial autocorrelation and P-Value (0.5181): Not significant. So no significant spatial autocorrelation detected in this area.

**Observation 5:** Z-Score (4.0428): Positively spatial autocorrelation and P-Value (2.641e-05) : Highly significant, so there is a strong positive spatial autocorrelation in this area, suggesting a cluster of similar values.

**Observation 6:** Z-Score (0.4697): positive spatial autocorrelation and P-Value (0.3193): Not significant, so no significant spatial autocorrelation detected in this area.

For our example, it is possible to create maps with tmap package to show the housing prices, the local Moran’s I, z-scores, and p-values.

Areas with p-value less than the significance level 0.05 correspond to areas for which we would reject the null hypothesis H0 and conclude they present positive spatial autocorrelation.

#install.packages("tmap")

#install.packages("devtools")

# Load the devtools package

library(devtools)

library(spdep)

map <- st\_read(system.file("shapes/boston\_tracts.shp", package = "spData"), quiet = TRUE)

map$vble <- map$MEDV

nb <- poly2nb(map, queen = TRUE) # queen shares point or border

nbw <- nb2listw(nb, style = "W")

lmoran <- localmoran(map$vble, nbw, alternative = "greater")

# Install the tmap package from GitHub #devtools::install\_github("r-tmap/tmap")

library(tmap) #tmap\_mode("view") #allows you to create an interactive map

map$lmI <- lmoran[, "Ii"] # local Moran's I

map$lmZ <- lmoran[, "Z.Ii"] # z-scores # p-values corresponding to alternative greater

map$lmp <- lmoran[, "Pr(z > E(Ii))"]

p1 <- tm\_shape(map) + tm\_polygons("vble", title = "vble", style = "quantile", legend.show = TRUE) + tm\_layout(legend.position = c("right", "top"))

p2 <- tm\_shape(map) + tm\_polygons("lmI", title = "Local Moran's I", style = "quantile") + tm\_layout(legend.outside = TRUE)

p3 <- tm\_shape(map) + tm\_polygons("lmZ", title = "Z-score", breaks = c(-Inf, 1.65, Inf)) + tm\_layout(legend.outside = TRUE)

p4 <- tm\_shape(map) + tm\_polygons("lmp", title = "p-value", breaks = c(-Inf, 0.05, Inf)) + tm\_layout(legend.outside = TRUE)

tmap\_arrange(p1, p2, p3, p4)

Finally for the example, you can use alternative = "two.sided" instead of alternative = "greater", to test H0: no spatial autocorrelation vs. H1: positive or negative spatial autocorrelation. In this two-sided test, z-score values **lower** than **– 1.96 indicate negative spatial autocorrelation**, and z-score values **greater** than **1.96 indicate positive spatial autocorrelation.**

To plot the map which shows the areas with negative, no, and positive spatial autocorrelation, obtained by breaking the legend according to these z-score values.

**Clusters**

As previously shown, Local Moran’s I allows to identify 4 types of clusters: High-High, High-Low, Low-High and Low-Low.

To detect clusters, we first use the localmoran() function to calculate the local Moran’s I where column 5 of the returned object contains the the p-values for the alternative hypothesis "two.sided".

Following, identify the clusters of each type by using the information provided by the Moran’s I scatterplot obtained with the moran.plot(). Then, identify the cluster types by using the quadrants of the scaled values **(mp$x)** and their spatially lagged values **(mp$wx)**, and the p-values obtained with the local Moran’s I for each of the areas **(map$lmp).** The classification of the clusters is as presented in the code below.

Areas with significant local Moran’s I are classified as high-high if both the value and its corresponding spatially lagged value are positive, low-low if both the value and its spatially lagged value are negative, high-low if the the value is positive and the spatially lagged value negative, and low-high if the value is negative and the spatially lagged value is positive.

To plot, create the variable quadrant denoting the type of cluster for each of the areas using the quadrant corresponding to its value and its spatially lagged value, and the p-value. Specifically, areas with quadrant equal to 1, 2, 3, and 4 correspond to clusters of type high-high, low-low, high-low, and low-high, respectively. Areas with quadrant equal to 5 are non-significant.

lmoran <- localmoran(map$vble, nbw, alternative = "two.sided")

head(lmoran)

map$lmp <- lmoran[, 5] # p-values are in column 5

mp <- moran.plot(as.vector(scale(map$vble)), nbw)

head(mp) map$quadrant <- NA

# high-high

map[(mp$x >= 0 & mp$wx >= 0) & (map$lmp <= 0.05), "quadrant"]<- 1

# low-low

map[(mp$x <= 0 & mp$wx <= 0) & (map$lmp <= 0.05), "quadrant"]<- 2

# high-low

map[(mp$x >= 0 & mp$wx <= 0) & (map$lmp <= 0.05), "quadrant"]<- 3

# low-high

map[(mp$x <= 0 & mp$wx >= 0) & (map$lmp <= 0.05), "quadrant"]<- 4

# non-significant

map[(map$lmp > 0.05), "quadrant"] <- 5

tm\_shape(map) + tm\_fill(col = "quadrant", title = "",

breaks = c(1, 2, 3, 4, 5, 6),

palette = c("red", "blue", "lightpink", "skyblue2", "white"),

labels = c("High-High", "Low-Low", "High-Low", "Low-High", "Non-significant")) + tm\_legend(text.size = 1) + tm\_borders(alpha = 0.5) + tm\_layout(frame = FALSE, title = "Clusters") + tm\_layout(legend.outside = TRUE)

**Bayesian spatial models**

Bayesian hierarchical models can be used to analyze areal data that arise when a variable is aggregated into areas that form a partition of the study region.

Models can be specified to describe the variability in the response variable as a function of a number of covariates known to affect the outcome, as well as random effects to model variation not explained by the covariates.

This provides a flexible and robust approach that allows us to assess the effects of explanatory variables, accommodate spatial autocorrelation, and quantify the uncertainty in the estimates obtained.

A commonly used spatial model is the Besag-York-Mollié (BYM) model, which considers the fixed effects and also includes a spatial random effect. This latter accounts for the spatial dependence between outcomes indicating that areas that are close to each other may have similar values.

**Bayesian hierarchical models** can be **fitted** using a number of approaches such as integrated nested **Laplace approximation (INLA) and Markov chain Monte Carlo (MCMC) methods.**

**INLA is a computational approach to perform approximate Bayesian inference.** This includes a wide range of models such as **generalized linear mixed** models **and spatial and spatio-temporal models**. INLA uses a combination of **analytical approximations** and numerical integration to obtain **approximated** **posterior** distributions of the parameters and is very fast compared to MCMC methods.

The R-INLA package can be used to fit models using INLA. To install R-INLA, use the install.packages() function specifying the R-INLA repository since the package is not on CRAN.

install.packages("INLA", repos = "https://inla.r-inla-download.org/R/stable", dep = TRUE)

library(INLA)

**Spatial modeling of housing prices**

In this part, we work on an example to learn how to specify and fit a Bayesian hierarchical model to estimate housing prices in Boston, USA, using the R-INLA package.

library(sf)

library(spData)

map <- st\_read(system.file("shapes/boston\_tracts.shp",package = "spData"),quiet = TRUE)

As previously clarified, this dataset contains housing data of 506 Boston census tracts including median prices of owner-occupied housing in $1000 USD (MEDV), per capita crime (CRIM), and average number of rooms per dwelling (RM).

We create the variable called vble with the logarithm of the median prices, and map this variable using mapview. The map suggests that the housing prices are greater in the west, and prices are related to those in neighboring areas.

library(mapview)

map$vble <- log(map$MEDV)

mapview(map, zcol = "vble")

A screenshot of a map

Description automatically generated

Note : Why considering the log of median price instead of the median price ?

**For Normalization** as housing prices have a wide range with some **extremely high values.** Taking the logarithm helps to **compress this range**, making the **distribution** more **symmetrical** and **manageable** for **modeling**. Additionally, the log **reduces the variability** in **price differences**, which can help meet the assumptions of linear regression models.

Further, the logarithm allows for more straightforward interpretation in terms of percentage changes. For example, a unit increase in the logarithm of housing price might represent a certain percentage change in the actual price.

To plot the relationships **between pairs of variables**, use the ggpairs() function of the GGally package.

install.packages("GGally")

library(GGally)

ggpairs(data = map, columns = c("vble", "CRIM", "RM"))

A screenshot of a graph

Description automatically generated

We observe a negative relationship between the logarithm of housing price and crime, and a positive relationship between the logarithm of housing price and the average number of rooms.

Following, we model the logarithm of the median prices using, as covariates, the per capita crime (CRIM) and the average number of rooms per dwelling (RM). Let Yi be the logarithm of housing price of area i, i=1,…,n. We fit a BYM model that considers Yi as the response variable, and crime and number of rooms as covariates:

A math equations on a white background

Description automatically generated

The spatial **random effect ui** captures **spatial autocorrelation** or the **similarity between neighboring areas.** This effect assumes that areas that are geographically close to each other tend to have similar response values. It is modeled with **conditional autoregressive model (CAR)** that **smooths** the data **according** to a **certain neighborhood structure**.

However, the **unstructured effect vi** accounts for the **unexplained** or **residual variatio**n in the data that isn't **captured by the structured spatial effect.** It **represents** the **random** or **uncorrelated variability among areas that isn't related to their spatial proximity.** This effect is akin to the **'noise'** or **random fluctuations present** in the data that **cannot be attributed** to **spatial relationships**.

This spatial **random effect** **ui** needs to be **specified** using a neighborhood structure to **capture spatial dependencies among neighboring regions.** For this example, assume that two areas are neighbors if they share a common boundary. So you can create a neighborhood structure using functions of the spdep package. First, use the poly2nb() function to create a neighbors list based on areas with contiguous boundaries. Explained previously, each element of the list nb represents one area and contains the indices of its neighbors. For example, nb[[1]] contains the neighbors of area 1.

install.packages("INLA")

library(spdep)

library(INLA)

nb <- poly2nb(map)

head(nb)

Then, we use the nb2INLA() function to convert the nb list into a file called map.adj with the representation of the neighborhood matrix as required by R-INLA. The map.adj file is saved in the working directory which can be obtained with getwd(). Then, we read the map.adj file using the inla.read.graph() function of R-INLA, and store it in the object g which we later use to specify the spatial model using R-INLA.

nb2INLA("map.adj", nb)

g <- inla.read.graph(filename = "map.adj")

We specify the model formula by including the response variable, the ~ symbol, and the fixed and random effects.

map$re\_u <- 1:nrow(map) # index to specify the elements of the structured spatial random effect

map$re\_v <- 1:nrow(map) # index to specify the elements of the unstructured effect

formula <- vble ~ CRIM + RM +

f(re\_u, model = "besag", graph = g, scale.model = TRUE) +

f(re\_v, model = "iid")

By default, there is an intercept so we do not need to include it in the formula. In the formula, random effects are specified with the f() function. This function includes as first argument an index vector that specifies the element of the random effect that applies to each observation, and as second argument the model name. For the spatial random effect ui, we use model = "besag" with neighborhood matrix given by g. The option scale.model = TRUE is used to make the precision parameter of models with different CAR priors comparable. For the unstructured effect vi, we choose model = "iid". The index vectors of the random effects are given by re\_u and re\_v created for ui and vi, respectively. These vectors are equal to 1,…,n, where n is the number of areas. Following, fit the model by calling the inla() function specifying the formula, the family, the data, and using the default priors in R-INLA. We also set control.predictor = list(compute = TRUE) and control.compute = list(return.marginals.predictor = TRUE) to compute and return the posterior means of the predictors.

res <- inla(formula, family = "gaussian",

data = map, control.predictor = list(compute = TRUE), #to compute posterior means of the predictors

control.compute = list(return.marginals.predictor = TRUE)) #to return the posterior marginals of the predictors

The resulting object res contains the fit of the model. To obtain a summary of the fitted model use summary(res). res$summary.fixed contains a summary of the fixed effects.

res$summary.fixed

A screenshot of a computer

Description automatically generated

We observe the intercept estimate ^β0= 1.427 with a 95% credible interval equal to (1.253, 1.6). We observe the crime estimate ^β1= –0.008 with a 95% credible interval equal to (–0.01, –0.005). This indicates crime is significantly negatively related to housing price. The coefficient of rooms is ^β2= 0.26 with a 95% credible interval equal to (0.233, 0.288) indicating number of rooms is significantly positively related to housing price. Thus, the results suggest both crime and number of rooms are important in explaining the spatial pattern of housing prices.

* When both crime rate (CRIM) and number of rooms (RM) are zero, the estimated average housing price is approximately 1.427 units.
* For every unit increase in crime rate, the estimated average housing price decreases by approximately 0.008 units.
* For every additional room, the estimated average housing price increases by approximately 0.26 units.

Further, it is possible to type res$summary.fitted.values to obtain a summary of the posterior distributions of the response μi for each of the areas. Column mean indicates the posterior mean (Provides the estimated average response (housing price) for each area based on the model.), and columns 0.025quant and 0.975quant are, respectively, the lower and upper limits of 95% credible intervals representing the uncertainty of the estimates obtained (Wider intervals indicate higher uncertainty about the predicted housing prices for that area.).

summary(res$summary.fitted.values)

A screenshot of a document

Description automatically generated

So, we can create variables with the posterior mean (PM) and lower (LL) and upper (UL) limits of 95% credible intervals.

PM represents the estimated average housing price for each area. It's a point estimate showing the model's best guess for housing prices in various locations.

LL an UL of intervals show the range within which the true housing prices for each area are likely to fall with 95% confidence.

# Posterior mean and 95% CI

map$PM <- res$summary.fitted.values[, "mean"]

map$LL <- res$summary.fitted.values[, "0.025quant"]

map$UL <- res$summary.fitted.values[, "0.975quant"]

and we could create maps of these variables with mapview by specifying a common legend for the three maps, and using a popup table with the name, the logarithm of the housing prices, the covariates, and the posterior mean and 95% credible intervals. We use the sync() function of leafsync to plot synchronized maps.

# common legend

at <- seq(min(c(map$PM, map$LL, map$UL)), max(c(map$PM, map$LL, map$UL)), length.out = 8)

# popup table

popuptable <- leafpop::popupTable(dplyr::mutate\_if(map, is.numeric, round, digits = 2), zcol = c("TOWN", "vble", "CRIM", "RM", "PM", "LL", "UL"), row.numbers = FALSE, feature.id = FALSE)

m1 <- mapview(map, zcol = "PM", map.types = "CartoDB.Positron", at = at, popup = popuptable)

m2 <- mapview(map, zcol = "LL", map.types = "CartoDB.Positron", at = at, popup = popuptable)

m3 <- mapview(map, zcol = "UL", map.types = "CartoDB.Positron", at = at, popup = popuptable) #install.packages('leafsync') l

ibrary(leafsync)

m <- leafsync::sync(m1, m2, m3, ncol = 3)

m

A screenshot of a map

Description automatically generated