11

Spatial Data

In all the previous chapters, we were dealing with the most common data format in the social sciences: tables. These tables usually contain numbers and text. We discussed how you can store these tables in files, read and process them in R, or use relational databases to manipulate data distributed over several tables. For some applications, however, we need to go beyond this simple model. There are special types of data for which the tabular data model is insufficient. In this part of the book, we take a look at three of them. This chapter introduces *spatial* data, which are observations that come with geographic coordinates. In other words, with spatial data, each observation has a particular location on the globe assigned to it. In later chapters, we will cover text as data, followed by the final applied chapter on network data.

11.1 WHAT IS SPATIAL DATA?

Spatial data are closely linked to the world of Geographic Information Systems (GIS), which is the software to collect, process, and analyze data with spatial coordinates. There are two major types of spatial data: *vector* data and *raster* data. In this chapter, we discuss only the former. You can think of a vector dataset as a table similar to the ones we have used so far, but where each row has some geographic information attached to it. Take a look at the example in Figure 11.1: You can see a standard table on the right, which contains information about cities. This is the same type of data model we have used so far.

However, in a vector GIS dataset, this tabular information (called the "attribute table") is amended with spatial coordinates. As you can see

https://doi.org/10.1017/9781108990424.016 Published online by Cambridge University Press

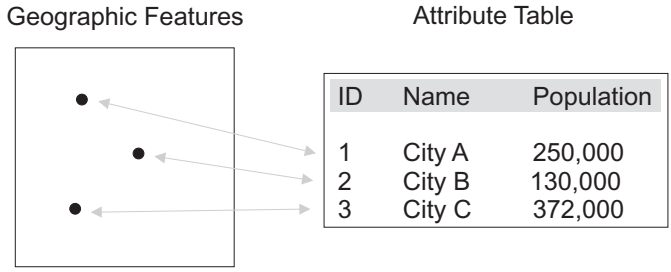

FIGURE 11.1. A table with cities (right), each of which is associated with spatial coordinates (left).

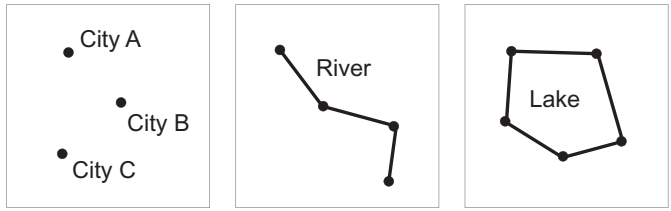

FIGURE 11.2. Different types of vector data.

in the example, each city in the table corresponds to a point on a map (left), which denotes the location of the respective city in the geographic space. Thus, a vector dataset is closely related to a standard table, with the only difference being that there is a new type of column with geographic information. This column contains what we typically call the *geometry* of a given entry. In our example above, this geometry is simply a two-dimensional *point*, in other words, a pair of (x, y) coordinates. However, GIS systems also allow more complex geometry types. Figure 11.2 shows a *line* geometry, which can be used to represent, for example, a river or a road. Finally, a *polygon* geometry is used to represent closed areas, such as a lake, or the borders of a country. A line is a sequence of (x, y) points, and a polygon is simply a closed line.

A question we cannot cover in depth is how we get from a three-dimensional surface (the earth) to a two-dimensional map, so I just convey some basic intuition here. There are two basic approaches to do this. The first one is to define a coordinate system for the (roughly spherical) surface of the earth (see Figure 11.3, left). This is what we do when using longitude and latitude coordinates: The equator has latitude 0, and locations north (south) of the equator have positive (negative) latitudes, each measured in radial coordinates. Longitude 0 is defined as going through

https://doi.org/10.1017/9781108990424.016 Published online by Cambridge University Press

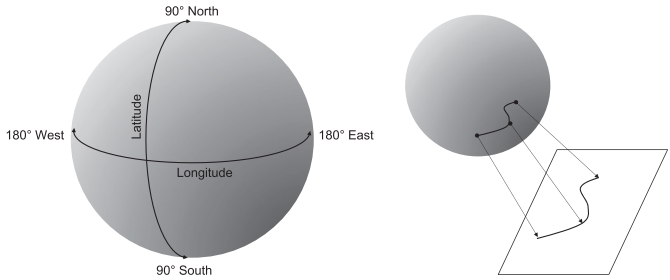

FIGURE 11.3. Coordinate systems for spatial data: A geographic (unprojected) coordinate system using radial coordinates on a spherical earth (left), and a projected coordinate system (right).

Greenwich, and longitudes increase as we move east, again measured in radial coordinates. Thus, each point on the globe can now be uniquely identified by its longitude (x) and latitude (y) values, which is called a *geographic coordinate system*. Importantly, we need to be careful what we can and cannot do with this system. Distance calculations, for example, can be tricky, since the distance on a sphere with radial coordinates cannot be computed similar to a planar surface.

The second approach to turn a three-dimensional surface of the earth into a two-dimensional map is to *project* it (see Figure 11.3, right). A projection is essentially an instruction for mapping points on the globe to corresponding ones on a map. There are many different ways for doing this, some of which are designed for particular purposes (e.g., they allow you to compute the distances between points correctly, thus avoiding problems such as the one described for geographic coordinate systems).

While the features of different projections are not important here, you need to keep in mind that the coordinate system and the projection of a geographic dataset are a important parameters you need to know when working with spatial data. Certain operations on spatial data (e.g., measuring the area of a polygon) only produce valid results if they are performed in a suitable projection. If you want to read up on this topic, I highly recommend the University of Boulder's *Earth Data Analytics* online course, which contains an entire chapter about spatial coordinate systems (Wasser, 2020).

GIS software is designed to handle spatial data – it allows you to read, modify, analyze, or visualize it. There are many different GIS systems available: *ArcGIS* is one of the most widely used commercial ones. If you want to try out an open-source GIS, I recommend *QGIS*, which is available free of charge at https://www.qgis.org/ for all major operating

https://doi.org/10.1017/9781108990424.016 Published online by Cambridge University Press

systems. In this chapter, we do not rely on specialized GIS software. Rather, we use extensions to the tools we introduced in the previous parts of the book: The R statistical software can in fact be turned into a powerful tool to process and analyze spatial data. Also, PostgreSQL has a spatial extension called PostGIS, which allows us to carry out spatial operations in SQL in combination with all the existing benefits of relational databases. However, before we discuss spatial data management with R and PostGIS, let me briefly introduce the applied example we use in this chapter.

11.2 APPLICATION: PATTERNS OF VIOLENCE IN THE BOSNIAN CIVIL WAR

In the practical exercises for this chapter, we examine a dataset about violent incidents in the civil war in Bosnia (1992–1995). The break-up of the Yugoslav Federation went along with an outbreak of violence between the three major population groups. Much of this violence happened in the (now independent) republic of Bosnia and Herzegovina, which is what we focus on in this chapter. Specifically, we will analyze the distribution of violence in Bosnia over space, to identify where it was most severe. Although we will not do a full explanatory analysis to study the drivers of this violence, the approach we present here is usually similar for any kind of statistical analysis that involves spatial data.

We use data on violent events from the *Geo-referenced Event Dataset* (GED), collected and maintained by the Uppsala Conflict Data Program (Sundberg and Melander, 2013; Högbladh, 2019). The GED is part of a family of datasets related to political violence, and the current version of the data as well as much additional documentation and information can be found on their website. The GED is an *event dataset*, which means that it provides information at the level of individual incidents. In the case of the GED, each of these incidents is a violent, lethal confrontation between two of the conflict parties. The dataset records the date of the incident, the actors involved, the number of casualties, as well as several additional variables. The following is a (shortened) single entry from the dataset:

id	side_a	side_b	source_article	date_start	longitude	latitude
200416	Gvt. of BH	Civilians	BBC Monit.	1993-10-26	17.28	44.55

Each incident has an `id`, and it identifies the participants in the event (`side_a` and `side_b`). In the above example, the dataset records

https://doi.org/10.1017/9781108990424.016 Published online by Cambridge University Press

an incident of violence against civilians, perpetrated by government forces. This information was obtained from an article published in BBC Monitoring (`source_article`). The incident took place on October 26, 1993, at the location with the given geographic coordinates (`longitude` and `latitude`). As we can see from the latter coordinates, each entry in the GED corresponds to a point on the map, so we can treat the entire collection as a GIS vector dataset in our application below.

To examine the spatial pattern of violence, we use Bosnia's pre-war administrative divisions. The smallest administrative unit was the municipality (opština), and Bosnia had 109 of them (with the capital Sarajevo divided into five municipalities). Our goal in this exercise is to compute the level of violence for each of the municipalities over the course of the war. Using administrative divisions as spatial units of observation is only one way to conduct a spatial analysis. Although we do not do this here, you could use this approach to relate an outcome we want to explain (in our case, violence) to particular socio-demographic variables measured at the level of administrative divisions, for example the ethnonational composition of a municipality (Weidmann, 2011). Alternative approaches for spatial analysis include the use of artificial grid cells as a unit of observation (Tollefsen et al., 2012), or no fixed spatial unit at all, as in point process models (Warren, 2015).

Our task in the analysis below is, therefore, to combine the vector dataset of violent events (points) with a dataset of Bosnia's municipalities, which are represented by polygons. We do so by identifying those violent events that took place within a municipality. In other words, we use the spatial coordinates of events and municipalities to link them to each other. In the GIS world, this is called an "overlay" operation, but we can also refer to it as a "spatial" join – the joining of data based on a spatial relationship (in our case, a point being located in a polygon). In the next section, we process spatial data using R and some extension packages, before introducing the PostGIS spatial database as an alternative workflow.

11.3 READING AND VISUALIZING SPATIAL DATA IN R

While most spatial data are usually processed in specialized GIS systems, R has grown into a powerful and versatile GIS itself, due to the development of new extension libraries. In this section, we use R's *Simple Features* (`sf`) library, a relatively new generic library for vector data. The term "feature" is often used in the GIS world to refer to the computational

https://doi.org/10.1017/9781108990424.016 Published online by Cambridge University Press

representation of real-world objects, such as houses or lakes. The `sf` is the latest addition to R's spatial libraries and will likely supersede older ones, such as the `sp` package. As always, to use the functionality of the package, we have to load it first:

```
library(sf)
```

As described above, a vector dataset is essentially a data table with a new type of column that contains the spatial representation of the respective entry. This column is referred to as its geometry, and geometries can be points, lines, or polygons. The `sf` package follows exactly this approach. It uses R's standard data frames, but adds a new column type for geometries. So in other words, a spatial dataset in `sf` is just a regular table with a specific column for spatial information. Let us take a look at how to create such a table in R. We first load the GED data on violent events. This dataset comes as a regular CSV file:

```
events <- read.csv(file.path("ch11", "ged.csv"))
```

I removed many columns from the dataset to make the exercises below easier to follow. Compared to the original version, the reduced dataset contains only events for Bosnia, and only those events for which the exact location is known. Also, the dataset has a limited set of columns, namely, those with the unique ID of each event, the date it occurred, the location of the event (stored in the `longitude` and `latitude` columns), and the number of casualties (the `best` estimate provided by the GED):

```
summary(events)
```

```
       id          date_start           latitude       longitude
 Min.   :199077   Length:1136        Min.   :42.71   Min.   :15.78
 1st Qu.:199690   Class :character   1st Qu.:43.85   1st Qu.:18.10
 Median :200136   Mode  :character   Median :43.85   Median :18.38
 Mean   :200106                      Mean   :44.11   Mean   :18.16
 3rd Qu.:200503                      3rd Qu.:44.54   3rd Qu.:18.38
 Max.   :200874                      Max.   :45.19   Max.   :19.54
      best
 Min.   :   0.00
 1st Qu.:   1.00
 Median :   3.00
 Mean   :  17.29
 3rd Qu.:   6.00
 Max.   :8106.00
```

For now, R treats the `events` table as a regular data frame and does not know that each entry has spatial point coordinates attached to it.

https://doi.org/10.1017/9781108990424.016 Published online by Cambridge University Press

Therefore, we need to "spatially enable" the dataset, which we can simply do by converting it to a spatial object of type `sf`. The `st_as_sf()` function takes a regular data frame, and requires you to specify the names of the columns where the spatial coordinates are stored. In addition to the names of coordinate columns, we need to specify what spatial reference system is used for the dataset.[1] Longitude/latitude coordinates indicate a geographic coordinate system (see Figure 11.3), which has the ID 4326. If your data uses a different reference system or if it is projected, it is important to correctly specify the coordinate system here:

```
events <- st_as_sf(events, coords = c("longitude", "latitude"), crs = 4326)
```

Now you will see that `events` is no longer just a data frame, but much more:

```
print(events, n=2)
```

```
Simple feature collection with 1136 features and 3 fields
Geometry type: POINT
Dimension:     XY
Bounding box:  xmin: 15.78194 ymin: 42.71194 xmax: 19.53556 ymax: 45.18944
Geodetic CRS:  WGS 84
First 2 features:
      id date_start best                   geometry
1 199885 1993-02-01    6 POINT (18.80833 44.87278)
2 199767 1994-03-03    1 POINT (15.91861 44.84444)
```

What do we see in this output? Our dataset contains a total of 1,136 features, each of which is a conflict event. The data uses *points* as geometries, in a two-dimensional space (hence the dimension `XY`). We also see the overall spatial extent of the dataset, referred to as its "bounding box" – this is the rectangle defined by the minimum and maximum coordinates along the x and y axes.

`sf` provides plotting functions specifically designed for spatial data. The easiest approach is to plot only the events as points. This is done by extracting the geometry from the dataset using the `st_geometry()` function, and by sticking it into the plot function as follows:

```
plot(st_geometry(events))
```

You can see the result in Figure 11.4. However, often we may want to color/style the plotted features according to some quantity associated

[1] Spatial reference systems were defined by the *European Petroleum Survey Group* (EPSG); online catalogues can be accessed, for example, at https://spatialreference.org or https://epsg.io.

https://doi.org/10.1017/9781108990424.016 Published online by Cambridge University Press

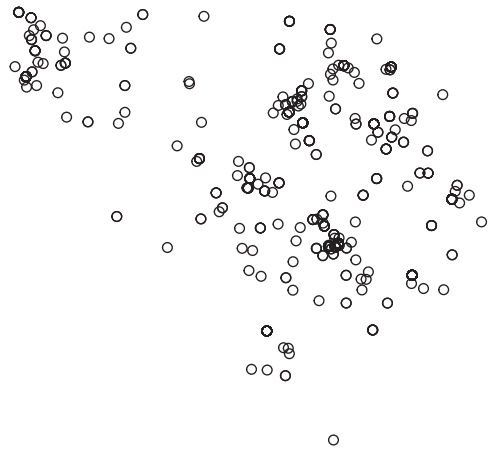

FIGURE 11.4. A simple plot of the conflict events from the GED.

with them. For example, we could color the dots according to the severity of the event, or according to the year in which they took place. In `sf`, this can be done by subsetting the data to the variable you want to use for coloring, and using again the `plot()` function, as for example in `plot(events["best"])`. There are many other options for tuning the plotting of spatial features, and you should consult the `sf` manual if you are interested in learning more.

We have now imported the events data from a CSV file, and converted it to a spatial dataset with point geometries. The second dataset we need contains the municipalities for our spatial analysis. The borders of each municipality are stored as a polygon, which is why we cannot simply store their coordinates in two columns of a CSV table. Instead, the dataset of municipal borders is provided in the *shapefile* format, an old legacy format for GIS vector datasets. As you can see in the data repository, a shapefile actually consists of at least three files with the same name, but different endings (`.shp`, `.shx`, and `.dbf`). Due to the fact that this format is still widely used, all GIS tools including `sf` are able to import it. When we do this, we have to manually set the coordinate reference system to the standard longitude/latitude system using the `crs` parameter, as above:

```
municipalities <- st_read(file.path("ch11", "bosnia.shp"), crs = 4326)

Reading layer `bosnia' from data source
  `/Users/nilsw/Books/DataManagement/dmbook/ch11/bosnia.shp'
  using driver `ESRI Shapefile'
Simple feature collection with 109 features and 2 fields
```

https://doi.org/10.1017/9781108990424.016 Published online by Cambridge University Press

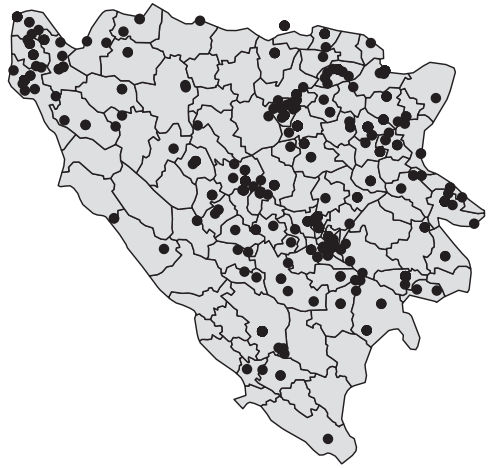

FIGURE 11.5. Municipality boundaries and conflict events.

```
Geometry type: POLYGON
Dimension:     XY
Bounding box:  xmin: 15.74059 ymin: 42.56583 xmax: 19.61979 ymax: 45.26595
Geodetic CRS:  WGS 84
```

With both the events and the administrative borders imported in R, let us take a quick look at a combined plot. For the map in Figure 11.5, we first plot the underlying municipalities, and then place the events on top of them with the `add=T` parameter:

```
plot(st_geometry(municipalities), col = "lightgrey")
plot(st_geometry(events), pch = 16, add = T)
```

We see that there are some areas with lots of events, while others experienced no violence. Still, this is difficult to tell exactly: Events occurring at the same location have the same coordinates and are therefore plotted on top of each other, which makes it impossible for us to keep them apart. Therefore, we proceed with our exercise and count the number of events per municipality, which helps us gauge the spatial distribution of violence over the entire country.

11.3.1 Overlaying Different Spatial Datasets in `sf`

Rather than just plotting our two datasets – the municipalities and the events – on top of each other, we now need to link events to their respective

https://doi.org/10.1017/9781108990424.016 Published online by Cambridge University Press

municipalities, such that we can count them. Before we do so, a short comment on terminology is in order. In the GIS world, a spatial dataset used in a project is typically called a *layer*. In our example, therefore, we have two layers – a layer of municipality boundaries, and a layer of events. Linking the events to the municipalities based on their location is an example of what is called an *overlay* operation in GIS terminology. However, from a relational database perspective, we can also think of these layers as *tables with spatial coordinates*. This way, an overlay operation is equivalent to *joining* tables based on location: Rather than linking records based on a common attribute (which is what a regular join does), we link them by location. In our example, we want to combine each event with the municipality it occurred in. In other words, an overlay of this kind is simply a *spatial join*, and I now demonstrate how this is done in R and `sf`.

The `st_join()` function is used to carry out a spatial join, so let us take a look at what it does:

```
joined <- st_join(events, municipalities)
print(joined, n = 2)

Simple feature collection with 1136 features and 5 fields
Geometry type: POINT
Dimension:     XY
Bounding box:  xmin: 15.78194 ymin: 42.71194 xmax: 19.53556 ymax: 45.18944
Geodetic CRS:  WGS 84
First 2 features:
    id.x date_start best id.y  name                  geometry
1 199885 1993-02-01    6  115 Breko POINT (18.80833 44.87278)
2 199767 1994-03-03    1   26 Bihac POINT (15.91861 44.84444)
```

A spatial join is very similar to a regular, non-spatial join: It links the records of one table to the corresponding records from another table. So for each conflict event, the function appends the attributes of the corresponding municipality the event is located in. In the above example, the event with ID 199767 occurred on March 3, 1994 in municipality 26, which is Bihać. The joined dataset is again a spatial one, as you can see from the geometry column, since it retains the spatial coordinates of the first dataset (the events).

Using the `st_join()` function in this way hides much of the power and complexity of spatial joins. Without an additional specification, layers are joined based on intersecting geometries; that is, an event is linked with

https://doi.org/10.1017/9781108990424.016 Published online by Cambridge University Press

a municipality if its geometry *intersects* with the municipalities geometry (the polygon). This does exactly what we want. However, things can become more complicated with different types of geometries or different relationships between them. For example, we can spatially join tables if geometries touch (but not intersect) each other, or if they are located within a certain distance to each other. To illustrate the basic idea of a spatial join, however, we do not go into more detail here.

We have now joined the two layers, such that each event is linked to the corresponding municipality. To create a map of the intensity of violence across Bosnia, there are two steps left to do: First, we need to aggregate the joined datasets by municipality and count the number of events for each of them, and, second, we need to append this information to our original dataset of municipalities, so that we can plot it as a map. Let us start with the first step. For convenience, we use the `tidyverse` approach for aggregation and merging (see Chapter 7), but it is of course also possible to do this in base R:

```
library(tidyverse)
eventcounts <- joined %>%
  as.data.frame() %>%
  group_by(id.y) %>%
  count(name = "num_events")
print(eventcounts, n = 3)
```

```
# A tibble: 79 x 2
# Groups:   id.y [79]
   id.y num_events
  <int>      <int>
1     1          1
2     2          7
3     4          1
# ... with 76 more rows
```

The above code first converts the `joined` spatial dataset to a regular, non-spatial data frame with `as.data.frame()`, since we no longer need the geometries of the municipalities. Using the standard `tidyverse` approach, it then groups the data using the `id.y` variable (which is the municipality identifier), and counts the records for each of them. This way, we get a list of municipality IDs (`id.y`) with the number of events that occurred in these municipalities. It is important to notice that the municipalities that do not contain any events do not show up in this list. `eventcounts` only contains event counts for 79 municipalities, which means that

https://doi.org/10.1017/9781108990424.016 Published online by Cambridge University Press

30 out of 109 municipalities did not experience any conflict events according to the GED.

What is left for us to do is to merge this information with the existing municipalities dataset. Here, we need to be careful, since the data frame with the event counts is much shorter than the complete list of municipalities. Therefore, we use a left join, which preserves the entire list of municipalities:

```
municipalities_sf <- municipalities %>%
  left_join(eventcounts, by=c("id" = "id.y"))
```

At the end of the chapter, we will use this new dataset to plot the distribution of violence across the different municipalities in Bosnia.

11.4 Spatial Data in a Relational Database

In the first part of this chapter, we relied on spatial data stored in files, which we then imported in R for processing. This file-based approach is only one way to work with spatial data. Similar to relational databases for non-spatial data, we can also use these databases to store and process data with spatial coordinates. There are several advantages of the latter, for example, a more efficient processing of large datasets, but also concurrent access to the data by multiple users.

In this chapter, we rely again on the PostgreSQL relational database system. As you know, PostgreSQL is a great tool to work with tabular data – this is something we discussed at length in the previous chapters. As it turns out, however, PostgreSQL can also process spatial data, once we enable the PostGIS spatial extension. The combination of PostgreSQL/PostGIS (which from now on, we simply refer to as PostGIS) then becomes a powerful spatial database that is an ideal tool for more complex spatial data operations. You interact with PostGIS in the same way as we did with PostgreSQL alone. This means that you need to have the database server running and set up a new database (which we call `spatialdata`) for this chapter, as described in Chapter 2. We then connect to our database exactly as we did in the previous chapters:

```
library(RPostgres)
db <- dbConnect(Postgres(),
   dbname = "spatialdata",
   user = "postgres",
   password = "pgpasswd")
```

https://doi.org/10.1017/9781108990424.016 Published online by Cambridge University Press

We now have a blank PostgreSQL relational database that keeps data in tables. For us to be able to work with spatially referenced data, however, we need to spatially enable our database by switching on the PostGIS extension:

```
dbExecute(db, "CREATE EXTENSION postgis")
```

Let us check if PostGIS is working correctly. The following code should output a single line similar to what you see below, which indicates the PostGIS version installed on your system:

```
dbGetQuery(db,
   "SELECT name, installed_version
   FROM pg_available_extensions
   WHERE name = 'postgis'")

     name installed_version
1 postgis             3.1.5
```

We are now ready to import the spatial datasets into PostGIS. Again, we use the conflict events and the municipalities data for Bosnia that you are familiar with from the exercises above. Let us start with the events. We first read the CSV file and write it as a simple table in the database:

```
events <- read.csv(file.path("ch11", "ged.csv"))
dbWriteTable(db, "events", events)
```

This step is exactly the same as for a non-spatial table. So far, our table is not yet "spatially enabled," which means that PostGIS does not know yet that each event is actually associated with a point with x and y (or rather, longitude and latitude) coordinates. This is why, similar to the R-based workflow above, we need to explicitly create a column in the events table for the spatial location associated with each row. As with the `sf` package for R, this column is called a *geometry* column, and we create it using an `ALTER TABLE` statement. Recall that we have used different column types in PostgreSQL before, such as `integer` for numbers or `varchar` for text (see Chapter 8). With PostGIS enabled, we can now define columns of type `geometry`. For a geometry column, you need to specify the type of the geometry (a `point`, a `line` or a `polygon`), as well as the coordinate reference system (the EPSG ID we used above):

```
dbExecute(db,
   "ALTER TABLE events ADD COLUMN geom geometry(point, 4326)")
```

https://doi.org/10.1017/9781108990424.016 Published online by Cambridge University Press

We can now use the data in the `longitude` and `latitude` fields of our table to update the geometry column with newly created point geometries. Here, the `st_point()` function creates the point, and the `st_setSRID()` function defines the spatial coordinate system (which you already know from the previous section):

```
dbExecute(db,
   "UPDATE events
   SET geom = st_setSRID(st_point(longitude, latitude), 4326)")
```

Done! Our `events` table now has the spatial coordinates of all events stored in a new column, which we can later use for spatial computations. Next, we need to import the municipalities to our database. Here, we follow a slightly different approach. We first import the shapefile using the `st_read()` function from the `sf` package (see above), rename the geometry column to `geom` for consistency, and then send the (spatial) table to PostGIS. The latter is done using the `st_write()` function from `sf`, which is essentially the spatial equivalent to the `dbWriteTable()` we used in previous chapters. The function requires you to specify the database connection (this is the `db` object), as well as a name for the layer you would like to create in the database.

```
municipalities <- st_read(file.path("ch11", "bosnia.shp"), crs = 4326) %>%
  rename(geom = geometry)
st_write(municipalities, dsn = db, layer = "municipalities")
```

Let us count the records in our two spatial tables to make sure that the import was successful:

```
dbGetQuery(db, "SELECT count(*) FROM municipalities")

  count
1   109

dbGetQuery(db, "SELECT count(*) FROM events")

  count
1  1136
```

11.4.1 A Spatial Join with PostGIS

The two tables were correctly imported: We have 109 municipalities and 1,136 events in our database. We can now proceed to spatially join them, using the geometry columns from the two tables. Let us recall first what a join of two tables does: It links the records of one table to those of another, based on a defined relationship between attributes of the two tables. In a conventional join, we usually require that an attribute from one table

https://doi.org/10.1017/9781108990424.016 Published online by Cambridge University Press

be equal to the attribute from another table. For example, in the previous chapter, we joined parties to elections based on the party ID attribute.

In a spatial join, we no longer match records based on common attributes, but based on their spatial coordinates. Specifically, we want to join an event with a municipality if the former *is contained in* a given unit. Hence, we simply replace the join condition in a conventional join (which usually requires two given attributes to be equal) with a spatial condition (namely, that one geometry is contained in another). Let us take a look at how this is done in the context of a `SELECT` statement:

```
dbGetQuery(db,
   "SELECT municipalities.id, events.id
   FROM municipalities JOIN events
      ON st_contains(municipalities.geom, events.geom)
   LIMIT 3")
```

```
   id  id..2
1 115 199885
2  26 199767
3  70 200575
```

This query demonstrates how we join the two tables based on their geometries. The basic structure of this query should be familiar: We specify what fields we want to see (in our case, the IDs of the municipalities and the corresponding events), and which tables we want to select from. Here, the `JOIN` keyword is used to link `municipalities` and `events`. The part that is new is the join condition. Here, we use the PostGIS function `st_contains()`, to require that we only want to retain those pairs of records where the municipality geometry (which is a polygon) contains the event geometry (which is a point).

So each entry we see in the output above is a pair of municipality ID and event ID that satisfies the join condition, that is, where the municipality contains the event. Since the municipalities are non-overlapping, each point can be linked with at most one municipality, so the maximum number of records this table can have is 1,136 (the number of events). Joining municipalities and events is only the first step. Again, as in the R-based example above, we need to aggregate this table such that it counts the number of events per municipality. You should be familiar with SQL's aggregation – all we need to do is specify the aggregation function (`count(*)`) as well as the grouping level (`GROUP BY`):

```
dbGetQuery(db,
   "SELECT municipalities.id, count(*) as num_events
   FROM municipalities JOIN events
      ON st_contains(municipalities.geom, events.geom)
```

https://doi.org/10.1017/9781108990424.016 Published online by Cambridge University Press

```
   GROUP BY municipalities.id
   LIMIT 3")
  id num_events
1  1          1
2  2          7
3  4          1
```

We now have the information that we want – for each municipality, we have the number of conflict events that occurred there. The final step is to add this information to our `municipalities` table, such that we can access it along with the existing data we have on municipalities. For this, we first add a new column to this table, which we later use to store the event counts:

```
dbExecute(db, "ALTER TABLE municipalities ADD COLUMN num_events integer")
```

Recall that in our above example, we first stored the event counts in a separate table, which we later merged with the main municipalities data frame. In SQL, we can do something similar. However, rather than creating a new table for the event counts that we later have to delete again, we use a "temporary" table within our statement. This table is created on the fly as we run the query, but exists solely for the purpose of this query and is later deleted. You can define such a temporary table using the `WITH` keyword. In the statement below, we define a temporary table called `eventcounts` using exactly the same SQL code as in the previous example. We then use this table in an `UPDATE` statement, where we link it to the main municipalities table using the municipality ID:

```
dbExecute(db,
   "WITH eventcounts AS (
      SELECT municipalities.id, count(*) as num_events
      FROM municipalities JOIN events
         ON st_contains(municipalities.geom, events.geom)
      GROUP BY municipalities.id)
   UPDATE municipalities
   SET num_events = eventcounts.num_events FROM eventcounts
   WHERE municipalities.id = eventcounts.id")
```

Only 78 municipalities are updated, since the others do not contain any events. Finally, we export the municipalities table as a spatial dataset to R, such that we later draw a map of the violence in Bosnia. The `st_read()` function can not just read data from files (as above), but also from a database connection. Once we have done that, we can close the database connection.

https://doi.org/10.1017/9781108990424.016 Published online by Cambridge University Press

```
municipalities_pg <- st_read(dsn = db, layer = "municipalities")
dbDisconnect(db)
```

11.5 RESULTS: PATTERNS OF VIOLENCE IN THE BOSNIAN CIVIL WAR

We computed event counts per municipality in two ways: First, using the `sf` package for R and, second, using the PostGIS spatial database. This gives us two spatial datasets: `municipalities_sf` is the result of the former approach, while `municipalities_pg` is the result of the latter. Both have the same structure: Each entry corresponds to a municipality, and the `num_events` column contains the event count for the respective municipality, with NA values for those municipalities without events. We can now use either of these datasets to draw a map of the distribution of violence in the Bosnian war as in Figure 11.6.

This maps shows us the municipalities that were hit hardest by violence in the civil war, as measured by the number of events. Part of the city of Sarajevo (displayed in black) experienced most of the attacks, but there are other areas in the north and the northwest of the country for which the GED recorded many conflict events. Of course, we can debate whether

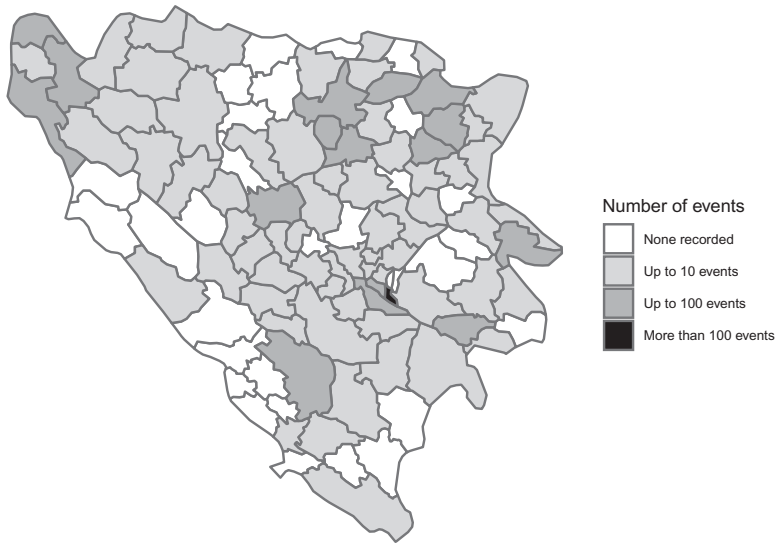

FIGURE 11.6. Civil war violence in Bosnia, as measured by the number of conflict events in the GED.

https://doi.org/10.1017/9781108990424.016 Published online by Cambridge University Press

counting the number of events is a valid approach to approximate the patterns of violence. Alternative ones are possible, for example, by using the GED's casualty statistics.

11.6 SUMMARY AND OUTLOOK

Spatial data are empirical observations tagged with geographic coordinates, such that they can be assigned to particular places on Earth. In this chapter, we have focused on vector data, which can be of different types: *points* to represent single locations, *lines* for rivers or roads, and *polygons* for political units. A GIS vector dataset usually consists of a set of geographic features, each of which is linked to additional data contained in an attribute table. When working with vector data in R or GIS, we use an amended version of the standard tabular data structure with special column types, namely, those that store the corresponding spatial features.

We discussed how to create these spatially enabled tables both in R (using the `sf` package), but also in PostgreSQL's PostGIS extension. For the exercises, we used two spatial datasets: a point dataset of violent events in the civil war in Bosnia, and a polygon dataset of municipality boundaries. With these two datasets, we performed a spatial overlay operation, where the points are superimposed on the boundaries to find out which events occurred in each municipality. In database terminology, this is a *spatial join*, where we link entries from two tables based on a spatial relationship they have (in our case, whether a point is located within a polygon).

In this and the following chapters, we use both file-based and database-driven workflows to process our data. You are now experienced enough to decide whether one or the other is more suitable for your project: Using R and its extension packages to process data stored in files is easier as regards the technical infrastructure you need, but may not be ideal for projects involving several collaborators and/or large datasets. Spatial operations can be time-consuming, which is why it is often useful to perform them in a spatial database such as PostGIS. In particular, while not necessary in our above example, you can use indexing as described in the previous chapter also for spatial columns, which, in many cases, will give you significant performance improvements. Beyond the topics we covered in the chapter, here are some suggestions that can be helpful to you:

https://doi.org/10.1017/9781108990424.016 Published online by Cambridge University Press

- *Explore the world of spatial data further:* If this chapter sparked your interest in spatial data, there is much more to explore. In our examples, we only used a limited set of operations with one type of spatial data – vector data. In particular in combination with raster data, there are many other useful applications for spatial analysis in the social sciences, and other, more focused introductions can help you make progress (see, for example, Lovelace et al., 2019).
- *Visualize your spatial data whenever possible:* When working with spatial data, it is often useful to plot your data. Looking at a map helps you understand your data better and lets you identify errors or artifacts that would otherwise be difficult to spot. This is why I recommend that you visually explore your data, using R's plotting features or an interactive GIS program such as QGIS (see next point).
- *Use QGIS to easily browse GIS datasets:* Oftentimes, it is useful to take a look at a spatial dataset to explore its structure or browse its contents. For this purpose, a graphical GIS can be useful. QGIS is a powerful, open-source GIS tool, which is available free of charge for all major operating systems from https://www.qgis.org/. With QGIS, you can even connect to a PostGIS spatial database and explore the different spatial tables visually.
- *Do not despair, GIS data formats can be confusing:* For GIS, there is a wealth of different file formats. Many of them are legacy formats (such as shapefiles), which continue to be used in the field. Also, the mix of two very different approaches (vector and raster data) adds to the complexity. As you make progress in spatial analysis, you will encounter many more file formats, and it is useful to consult the various online references for more information about their specifications.

https://doi.org/10.1017/9781108990424.016 Published online by Cambridge University Press