9.1 Maps

Using R and supporting packages we can create maps of the world, countries, counties, cities, or smaller areas, and we can portray characteristics or activities at selected locations.

9.1.1 Map the World

Projections

projection: Transformation to represent spherical coordinates on a flat surface.

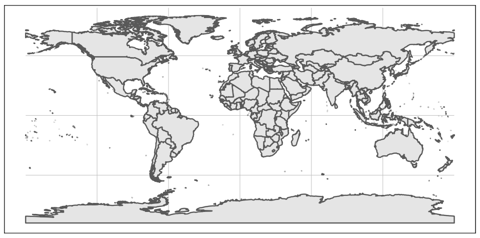
We typically view our maps on paper or a computer screen, both of which are flat. The Earth is almost a sphere, more accurately (but still not precisely) described as an ellipsoid derived from a rotated ellipse with two identical minor axes. A *projection* transforms spherical coordinates, which describe specific locations on the spherical surface, to the Cartesian coordinates of a flat surface. That is, a projection flattens the latitude and longitude of the Earth's ellipsoid surface to a two-dimensional representation. Every map of the Earth, or some part of the Earth's surface, is a projection.

The problem is that the surface of a flat surface cannot represent a spherical object such as an ellipsoid without distortion. This transformation cannot simultaneously retain accuracy of area, direction, distance, and shape, so each projection compromise at least one of these properties. There are many different projections, of which some of the most popular can be viewed online for interactive display. The `flexprojector.com` option is free, cross-platform downloadable Java software that allows for creating custom projections.

Interactive world maps from different projections

URL: `https://www.jasondavies.com/maps/transition/`
URL: `http://projections.mgis.psu.edu`
URL: `http://flexprojector.com`

Figure 9.1 presents two world maps, both easily created with `ggplot2`, derived from two different projections.


(a) Lat/long Projection

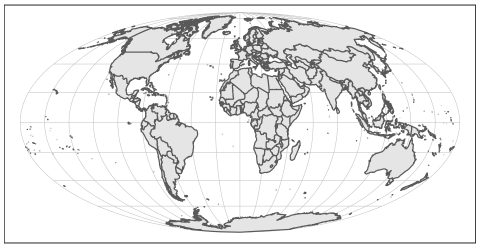

(b) Mollweide Projection

Figure 9.1: World maps created from the same data but two different projections.

The first projection, Figure 9.1a, is one of the simplest. This projection maps latitude and longitude directly into coordinates in the *x*-*y* plane, a version of what is called the Plate Carrée projection, which means "square plane" in French. The Plate Carrée projection renders the Earth as a flat rectangle, necessarily resulting in much distortion.

The projection in Figure 9.1b, Mollweide, presents a more sophisticated rendering of the Earth with much less distortion of the sizes of the continents, but with some loss of accuracy of angles between the continents and their shapes. The Mollweide projection follows from a non-linear transformation of the spherical coordinates of latitude and longitude into the coordinates of the map's x-y plane.

Spatial data

spatial data: Data that identifies the geographic location of features and boundaries on Earth.

To construct a map first download the corresponding *spatial data*, the data that describes geophysical features and locations of the area to be mapped, features such as the boundaries between countries, shore lines, and related details. Spatial data is stored in a file called a *shapefile* with a specific format. A comprehensive, free, public domain spatial dataset, provided in three different levels of detail and frequently updated, is Natural Earth, available on the web.

shapefile: Vector data storage format that stores the location, shape, and attributes of geographic features.

```
http://www.naturalearthdata.com
```

Files of spatial data can be downloaded directly from the website, or, more conveniently for R users, directly accessed with the R package `rnaturalearthdata` and formatted ready for use in R, as shown in the following example.

To read the spatial Natural Earth data into R, access the functions in the package `rnaturalearth`. The `ne_download()` function downloads spatial data directly from the Natural Earth website to an R data frame. Or, access data already downloaded with the package with `ne_country()`. For more detail use the `ne_states()` function. The default value of the parameter `scale` provides the least detail, `"small"`, but also the fastest computations. Figure 9.1 results from more detail by setting `scale` to `"medium"`, which also requires more processing to compute the map than does the default. Much more processing time is required for the most detail, set with `"large"`, which also requires installing the `rnaturalearthhires` package to access the much more comprehensive data.

Two `rnaturalearth` package functions for accessing Natural Earth spatial data

rnaturalearth: `world <- ne_countries(scale="medium", returnclass="sf")`
rnaturalearth: `world <- ne_download(scale="medium", type="countries",`
`returnclass="sf")`

In this example obtain spatial data for the entire world because no specific countries or continents were specified. The spatial data table, named *world* in this example, typically organizes the data in one of two shapefile formats: `"sp"` or `"sf"`. The `"sp"` format, for SpatialPolygonsDataFrame, is the older standard, of which geographers are transitioning away from to the newer standard, `"sf"`, or "simple features". The features are the objects that have a geometry such as points, lines, or polygons, all described with vector (drawing) attributes. The structure of `sf` data is simpler than `"sp"` encoded data. Each row represents a single spatial object such as a line, any associated data such as length, and a variable that contains the coordinates of the object.

simple features data frame example, Section 9.4, p. 222

By default, both `ne_countries()` and `ne_states()` return a spatial data frame of type `sp`. To obtain the emerging standard, invoke the `returnclass` parameter with the argument of `"sf"`. Many plotting functions such as provided by `ggplot2` convert the spatial data to `"sf"` format automatically, but more straightforward and faster to begin with the data structure input into the analysis.

Create the world maps

To create the map, here use `ggplot2` with its newly developed geom for Version 3, `geom_sf()`, which visualizes simple feature objects. Given this geom and the *world* data set, creating the default world map requires little effort.

Create the default version of Figure 9.1a.

ggplot2: `ggplot() + geom_sf(data=world)`

The default projection in Figure 9.1a is not one that a cartographer would typically choose to represent the Earth. Most would agree that the Mollweide projection presented in Figure 9.1b more accurately depicts the Earth's surface, and there are many more alternative sophisticated projections from which to choose. How to realize these options? Via `ggplot2` the answer is the `coord_sf()` function, which includes the `crs` parameter for Coordinate Reference System (CRS). A `CRS` transforms geospatial coordinates from one coordinate reference system to another, which includes projections.

Running the following code generates both Figure 9.1a and b from the previously created *world* data frame. To provide for the flexibility of creating maps with different projections, in this example most of the `ggplot2` function calls are saved in the `ggplot2` object *p*, referenced later to add a specific projection for the creation of each map.

R Input *Create the world maps in Figure 9.1*

```
ggplot2: p <- ggplot() + geom_sf(data=world) +
        theme_set(theme_bw()) +
        theme(panel.grid.major = element_line(color="gray75", size=.5))
Lat/long: p + coord_sf(crs="+proj=longlat")
Mollweide: p + coord_sf(crs="+proj=moll")
```

To map just part of the world use parameters `xlim` and `ylim` for `coord_sf()` to specify starting and ending longitudes and latitudes, respectively.

The `CRS` transformations are from the public domain `PROJ` library (PROJ contributors, 2018), version 4, written as `PROJ.4`. The `sf` function `st_proj_info()` lists the available projections within R. The following web site reference, some of the official documentation of `PROJ`, visually displays each projection and describes its parameters.

Available projections in R

sf: `st_proj_info(type="proj")`
URL: `https://proj4.org/operations/projections/index.html`

proj-string: Description of a coordinate system used to render a map.

Within the `PROJ` system, describe coordinate transformations with *proj-strings*, which serve as values of the `crs` parameter for the `ggplot2` function `coord_sf()`. Precede each parameter in a proj-string with a `+`. Here we focus on just the projection parameter, `proj`. The default value is `longlat`, explicitly provided in the preceding `ggplot2` function call, but not necessary. The value of `proj` for the Mollweide projection is `moll`, which must be explicitly indicated to obtain that projection.

datum: Define the shape and size of the Earth and provide a reference point for describing locations via coordinates.

Also required is a model, called a *datum*, of the specific spherical surface that serves as a reference point for the geospherical coordinates such as longitude and latitude. The most common choice of cartographers is the *WGS84* datum, the 1984 World Geodetic System standard, which also serves as the reference standard for GPS, the Global Positioning System. Still, alternate Earth surface models are available, as well as models designed specifically for local regions.

The reliance upon *WGS84* can be made explicit with `st_crs()` from the `sf` package, which retrieves the coordinate reference system upon which an `sf` data frame is based, illustrated for the *world* data frame in Listing 9.1.

```
> st_crs(world)
Coordinate Reference System:
  EPSG: 4326
  proj4string: "+proj=longlat +datum=WGS84 +no_defs"
```

Listing 9.1: Apply the `st_crs()` function to the *world* `sf` data frame.

The EPSG number is an alternate designation of the obtained prog-string. Each spatial data set is only interpreted within the context of a datum, and the projection that defines the map used to display the data. Different datums may be choose to enhance accuracy in a specific location. Using the correct datum on which the spatial data is based is essential because distances between locations can substantially differ according to different datums.

9.1.2 Raster Images

raster image: Set of tiny pixels or dots, each of which conveys a color.

vector image: Set of mathematical descriptions of geometric objects and positioning.

At the most general level, store an image in one of two primary storage formats: vector or raster. A *raster image* is composed of many pixels (on a computer screen) or dots (on a printed image). In contrast, a *vector image* consists of a mathematical description of geometric objects such as points, lines and polygons, and their relative positioning. A photograph is a raster image, typically composed of a large but fixed number of tiny pixels or dots. The format can require much storage space and does not scale well to larger images but can provide much detail and gradation of colors. Vector images can scale perfectly but without the fine level of detail and gradation provided by a raster image.

The previous examples of spatial data files in `sf` format are vector files. The `ne_download()` function also provides for highly detailed raster images available as part of the Natural Earth data set, as shown in Figure 9.2.

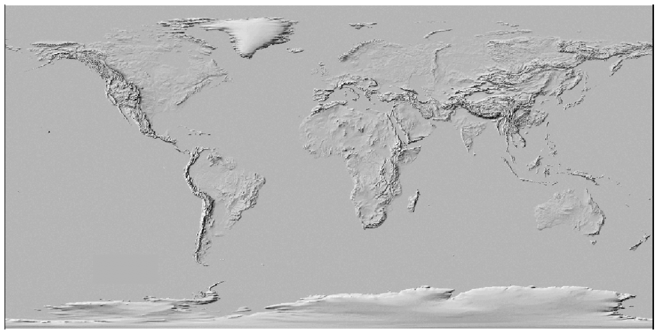


Figure 9.2: Grayscale raster image scale relief map of the world from Natural Earth.

To create this raster map, first download the data with the previously introduced `ne_download()`, but with the `category` parameter set to `"raster"`. Set the `type` parameter to `"MSR_50M"` to indicate a medium scale raster Manual Scale Relief map. Create the map with the base R function `plot()`, applied to objects of the class `raster`. As such, the full name of the plotting function is `plot.raster()`, invoked with just `plot()`.

R Input

naturalearth: `r.world <- ne_download(scale="medium", type="MSR_50M",`

```
                              category="raster")
```
base R: `plot(r.world, col=getColors("black", "white", n=48)`

Create the grayscale image with the parameter `col`. In this example, the `lessR` function `getColors()` generates a sequential scale of 48 shades of gray from `"black"` to `"white"`.

getColors() `lessR`, Section 3.2.1, p. 55

The default color palette for `plot.raster` generates the base R terrain colors. The following call to `getColors()` displays the terrain palette.

Base R terrain palette

lessR: `getColors("terrain", n=100, border="off")`

Of course, other color sequential scales could be employed to generate the raster image map.

9.1.3 Online geocode databases

geocoding: Obtain the geographical coordinates for a given location.

geocode: The geographical coordinates of a location.

Locate every place on the Earth's surface according to its geographical coordinates, usually latitude and longitude. Plot the locations on the map, such as cities or country boundaries, according to their latitude and longitude. *Geocoding* provides the geographical coordinates of a specific location, its *geocode*.

US Census Bureau geocodes

Multiple commercial options for geocoding exist, but so do some quality free alternatives. For locations within the USA, the US Census Bureau offers a free, authoritative source of geocodes, which provides access to an extensive database of addresses and their corresponding longitudes and latitudes.

US Census Bureau geocodes website

URL: `https://geocoding.geo.census.gov`

To obtain an individual geocode enter an address at the prompt, either as a single line with comma delimiters for the One Line button, or multiple lines for the Address button. Or, click on the Address Batch button to submit a batch file of up to 10,000 lines of addresses in either `csv` or Excel format to have the geocodes returned as a downloaded file in the same format as submitted. The first column of the input file is the row number, and the `csv` file cannot end with a blank line. The output file is named GecodeResults with a file type of `csv` of `xlsx`.

Google Map Services

A Google mapping technology called Google Map Services provides one source to obtain geocodes for addresses across the world. As of this writing, limited use of this service is free in the sense that although the lookup of geocodes costs .005¢ per geocode, or $5.00 per 1000, a $200 USD credit automatically applies each month. The result is that if no other mapping services are used, the first 40,000 geocodes per month are functionally free. The downside is that, unlike the previously discussed geocoding source, obtaining a geocode costs money, even if refundable, and so can only be accessed by providing the map service a credit card number.

To access this service within R requires registration with Google map services, and then proof of such registration offered to R. Register at the `cloud.google.com` website to obtain an Application Programming Interface (API) key to access specific types of information, here the geocode API as well as five other mapping API's by default. The second listed website is the source for maintaining the account.

> *Google map service geocodes registration*
>
> *URL*: `https://cloud.google.com/maps-platform/`
>
> *URL*: `https://console.cloud.google.com`

To register at the website, click a Get Started button, then choose Places. Next, select an existing project or enter a new project name. Now time to enter credit card information for billing. The system then provides a rather long character string that is a personal API Key. It is this key that allows you to access the mapping service from within R.

To inform R of your personal key, invoke `library()` to load the **ggmap** package, and then `register_google()` as follows. The `has_google_key()` function indicates if a key has been successfully registered and is available for the current R session.

library() base R, Section 1.1.4, p. 9

> *Inform R of your personal key*
>
> *ggmap*: `register_google(key="personal_key")`
>
> *ggmap*: `has_google_key()`

The `register_google()` function has a parameter `write`, `FALSE` by default. If set to `TRUE`, the function writes the entered personal key into the R system to be automatically available for future use. The potential difficulty with this approach is that R is not designed to store confidential information, so that it may be possible for a rogue package to locate this key and transmit the information somewhere else. Also obtain the personal key at any time from the google console, of which the `URL` is provided above.

Once the API key has been registered with R, Listing 9.2 illustrates the use of `geocode()` from **ggmap** to build a data frame of locations and longitudes and latitudes. First, create a character vector of the specified locations. Then submit this vector to `geocode()`, which constructs a `URL` for each location and then calls the mapping service with the `URL`'s, returning only each respective latitude and longitude. To construct the data frame of locations matched with their corresponding coordinates, here use base R `cbind()` for "column bind" to merge the two sources of information.

vector character, Section 1.3, p. 6

```
location <- c(
"615 SW Harrison St, Portland, OR",
"Disneyland",
"Rio de Janerio, Brazil"
)
d <- geocode(location)
d <- cbind(location, d)
d
                          location       lon       lat
1 615 SW Harrison St, Portland, OR -122.6832  45.51156
2                       Disneyland -117.9190  33.81209
3           Rio de Janerio, Brazil  -43.1729 -22.90685
```

Listing 9.2: Input, function call, and output for package **ggmap** function `geocode()`.

The example in Listing 9.2 demonstrates `geocode`'s impressive flexibility to identify locations according to a variety of references. The first location is a standard address, the second the name of a well-known landmark, and the third location only a city name name.

Geonames geocodes

A free, public domain database provides geocodes that each represents an entire city, not just in the USA but the entire world. GeoNames offers, among other databases, city geocodes in files that include cities across the world of just 500 or more inhabitants, as well as 1000 or more, 5000 or more, and 15000 or more inhabitants, as described in the *readme* file.

Readme file for the Geonames databases

URL: `http://download.geonames.org/export/dump/readme.txt`

Licensed under the generous Creative Commons Attribution 4.0 License, the data and related information can be copied and redistributed in any medium or format, and transformed and modified as needed for any purpose, even commercially, with the requirement to cite the source of the data.

Download the zipped data file from *geonames.org* from within R with base R `download.file()`. When calling this function, first specify the URL, and then the destination file. Because no path name precedes the file name in this example, R writes the file to the current working directory, obtained from `getwd()`. Use base R `unzip()` to unzip the file to a `.txt` file to read into R. The downloaded data file, however, does not include the variable names. Include the variable names with the call to `Read()` as the value of the base R parameter `col.names`.

Download and read Geonames databases such as cities15000.zip

base R: `download.file("http://download.geonames.org/export/dump/", cities15000.zip"))`

base R: `unzip("cities1500.zip")`

lessR:
```
d <- Read("cities15000.txt", col.names = c("id","name","ascii_name",
    "alt_names","latitude","longitude","feature_class","feature",
    "country.code","cc2","admin1","admin2","admin3","admin4",
    "population","elevation","dem","timezone","mod_date"))
```

Once read into R, to specify the data used in a subsequent map, query the data frame for specific city geocodes. For example, apply the base R subsetting function to limit the *d* data frame to all Italian cities larger than 250,000 inhabitants, with just the specified variables.

R Input *Base R query of geocities database for specified cities, variables*

data: `d <- Read("http://lessRstats.com/data/employee.csv")`

```
cols <- c("name", "longitude", "latitude", "population", "elevation")
rows <- d$country.code=="IT" & d$population > 250000
d <- d[rows, cols]
d
```

Listing 9.3 shows the query results.

```
          name longitude latitude population elevation
11990  Palermo  13.33561 38.13205     648260        14
12020  Catania  15.07041 37.49223     290927         7
12084    Turin   7.68682 45.07049     870456       239
12162     Rome  12.51133 41.89193    2318895        20
12231   Naples  14.26811 40.85216     959470        17
12258    Milan   9.18951 45.46427    1236837       122
12330    Genoa   8.94439 44.40478     580223        19
12354 Florence  11.24626 43.77925     349296        50
12458  Bologna  11.33875 44.49381     366133        54
12470     Bari  16.86982 41.12066     277387         5
```

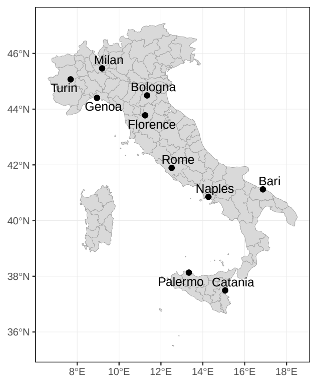
Listing 9.3: Extracted geocode information from free, downloadable Geonames database.

Define one variable, *cols*, to specify the variables to retain, and another variable, *rows*, to specify the rows of the data table to retain. Then do the subsetting with the `[ ]` operator. Replace the *country.code* in the definition of *rows* to select cities from any other country.

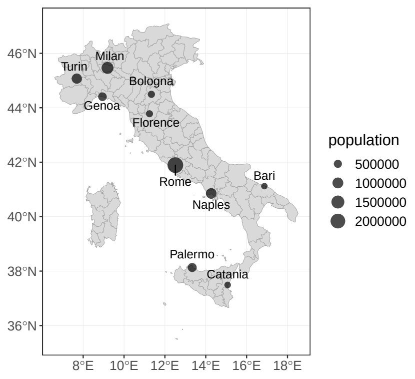
subsetting (extract) function, Section 1.2.3, p. 18

9.1.4 Create a Country Map with Cities

The example here is a map of one country, Italy, that includes its ten most populous cities. The `ggplot2` code to construct this map references different data sets in different layers. One data set is for the polygons that define the map of Italy, and the second data set contains the information for the ten cities, their geographical coordinates, and their populations. Apply both data sets in their respective layers to obtain Figure 9.3.



(a) cities as points



(b) cities as bubbles

Figure 9.3: Italy and its ten most populous cities.

The next task needed to generate the map is to create the data table of polygons for Italy and its provinces. To obtain the polygons that define the Italian provinces and boundaries with `rnaturalearth` function `ne_states()`, set `country` to `"italy"`. Also needed is a second data data frame of the city coordinates, based on the already obtained *d* data frame of the ten most populous Italian cities and their geographical coordinates.

create data frame of Italian cities, Section 9.3, p. 219

R Input *Two data sets for the map of Italy given data frame d from Listing 9.3*

rnaturalearth: `italy <- ne_states(country="italy", returnclass="sf")`

sf:
```
cities <- st_as_sf(d, coords = c("longitude", "latitude"),
                   crs=st_crs(italy), remove=FALSE)
```

Multiple spatial data sets plotted on the same panel should all share the same coordinate reference system and projection. If not explicitly specified with `coord_sf()`, `ggplot2`'s `geom_sf()` assumes the coordinate reference system of the first set, and performs any needed transformations on subsequent layers with functions from the `sf` package to ensure a common reference system. `geom_sf()` also adds by default the lines of longitude and latitude, as well as the axis value labels of longitude and latitude.

However, instead of relying on this default, use `st_as_sf()` from the `sf` package to assign the *italy* `CRS` to the cities data frame *d*. For parameter `crs`, invoke `st_crs()` to retrieve this `CRS`. Set the `st_as_sf()` `remove` parameter to `FALSE` to retain the city coordinates, latitude and longitude, as separate variables in the resulting `sf` data frame.

world maps code, Section 9.1.1, p. 214

As with the world maps, use `geom_sf()` to do the plotting. Each row of a `sf` data frame after the preliminary header information contains the specifications for an object to plot, including the type of object. So `geom_sf()` plots polygons for the map of Italy and points for the cities placed on the map.

Constrained to grayscale, for the map of Italy the `fill` for the polygon interiors and `color` for the borders are set to shades of gray. Reduce the size of the border lines from their default by setting `size` to 0.2. For the plotting of the cities, increase the default `size` to 2.

library() base R, Section **??**, p. **??**

The `geom_text_repel()` function, from the `ggrepel` package, extends the concept of `geom_text()` to provide convenient features for maps. By default, the text, here the city names, is written above or below the corresponding coordinate, to not write over the corresponding point. As always, the package must first be retrieved from the R package library with `library()`.

R Input *Map of Italy with its ten most populous cities plotted as points*

```
ggplot2: ggplot() +
      geom_sf(data=italy, fill="gray85", color="gray65", size=0.2) +
      geom_sf(data=cities, size=2) +
      theme_set(theme_bw()) + labs(x=NULL, y=NULL) +
ggrepel: geom_text_repel(data=cities, aes(longitude,latitude, label=name),
                         size=3.25, col="black")
```

scale_size_area() `ggplot2`, Section 6.2.3, p. 149

Figure 9.3b plots the coordinate for each city as a bubble, the size of which depends on the population of the corresponding city.[1] The `scale_size_area()` function scales the population as the area of each bubble, instead of the default radius. Activate plotting the bubble plot by mapping the variable *population* to the parameter `size`.

The `geom_text_repel()` parameters `nudge_y` and `nudge_x` move the corresponding text element the specified amount, and then, if the text element becomes too far from the corresponding coordinate, it automatically draws a line segment from the text to the coordinate. To use the `nudge` parameters, apply them to an array that is of the same length as the number of text elements to plot. Each element corresponds to the corresponding coordinate in the defining data frame. Here also invoke the `size` parameter to increase the text from the default size.

[1]The `geom_sf()` function properly plotted the bubbles, but due to an apparent bug, did not properly display the legend. As such, `geom_point()` did the plot in Figure 9.3b according to: `aes(longitude, latitude, size=population)`.

R Input *Map of Italy with its ten most populous cities plotted as bubbles*

ggplot2:
```
ggplot() +
      geom_sf(data=italy, fill="gray85", color="gray65", size=.2) +
      geom_sf(aes(size=population), data=cities, alpha=.7) +
      scale_size_area() +
      theme_set(theme_bw()) + labs(x=NULL, y=NULL) +
```
ggrepel:
```
geom_text_repel(data=cities, aes(longitude,latitude, label=name),
                size=3.75, col="black",
                nudge_y=c(.5,.4,.5,-.6,-.4,.4,-.2,0,.5,.4))
```

Visualize any other country on Earth, or multiple countries by specifying a vector of countries, with the `rnaturalearth` function `ne_states`.

9.1.5 Choropleth Map

A *choropleth map* shades areas in proportion to the values of a variable displayed on the map, such as population density or incidence of a disease. The example in Figure 9.4 is the Gini coefficient (Gini, 1921), an index of income inequality, of the 48 contiguous states of the USA from 2017 data U.S. Census Bureau (2017). New York state has the largest income equality, followed by a band of states throughout the south-east and California. The north-west and mid-west states have the lowest income inequality.

choropleth map: Specific areas are colored or shaded in proportion to the value of a corresponding variable.

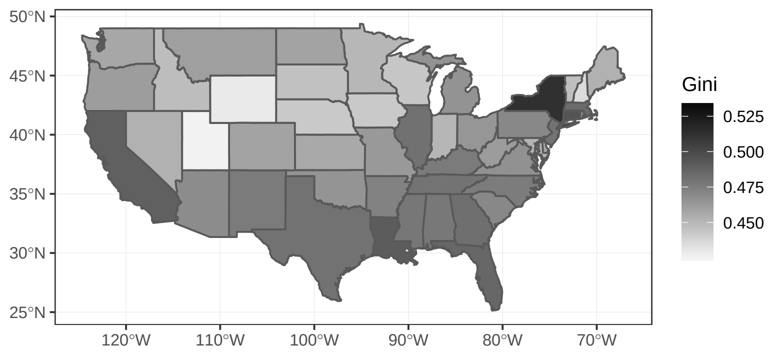


Figure 9.4: Income inequality by State of USA, 2017 Gini coefficients.

This example relies upon another widely used source of mapping data, the `maps` package. The map data files produced by the `map` function from `maps` is not in simple features format, so convert with `st_as_sf()` and store in the `sf` data frame *states*. Listing 9.4 presents the first several lines of *states*. After the header information, there are two data fields, *geometry*, the information for the polygon that defines the boundary of each state, and *ID*, which is the variable of state names, all in lower case.

The *gini* data are downloaded from the American Community Survey data section of the US Census web site, available at `factfinder.census.gov`. The downloaded file was edited by removing the first row and a few columns not relevant to the current analysis. To match the information in the *states* data frame, covert the state names to lower case and rename the column of State names as *ID*. Rename the column of Gini scores as *Gini*. This revised data frame is saved on the web for general access.

The data wrangling issue here is that to create the map in Figure 9.4 merge the Gini data,

```
> head(states, n=3)
Simple feature collection with 3 features and 1 field
geometry type:  MULTIPOLYGON
dimension:      XY
bbox:           xmin: -114.8093 ymin: 30.24071 xmax: -84.90089 ymax: 37.00161
epsg (SRID):    4326
proj4string:    +proj=longlat +datum=WGS84 +no_defs
                        geometry       ID
1 MULTIPOLYGON (((-87.46201 3...  alabama
2 MULTIPOLYGON (((-114.6374 3...  arizona
3 MULTIPOLYGON (((-94.05103 3... arkansas
```

Listing 9.4: Header information and first three rows of data for the simple features *states* data frame.

which exists in a regular data frame, into the special features *states* data frame. The *states* data frame consists of 49 rows of data, the contiguous 48 states and the District of Columbia. The *gini* data frame consists of all 50 states plus the District of Columbia and Puerto Rico. The merged data file should have all the rows of data (records) for which a state exists in both data frames.

Accomplish the merging with an inner join, that is, a joining of the two data frames that contains only the shared data that exists based on the values of the joined variable in each data table, here *ID*. Join the data frames by *ID* with the `sf` package function `inner_join.sf()`, which follows the form of the tidyverse `dplyr` function, `inner_join`, referenced without the `.sf`. The result is a simple features data frame with the Gini coefficient for each state.

R Input *Data wrangling to create the data frame for analysis yielding Figure 9.4*

sf, maps: `states <- st_as_sf(map("state", plot=FALSE, fill=TRUE))`
lessR: `gini <- Read("http://lessRstats.com/data/Gini2017.xlsx")`
base R: `gini$ID <- tolower(gini$ID)`
sf: `states <- inner_join(states, gini, by="ID")`

With `ggplot2` create the map with only a single line of code as `geom_sf()` accomplishes the work. Specify the *states* data frame to visualize the states of the USA, and then specify the variable *Gini* as the `fill` variable to create the choropleth map. The optional second line of code, the call to `scale_fill_gradient()`, displays the map in grayscale. The default is a sequential blue palette.

R Input *Mapping code to create Figure 9.4*

ggplot2:
```
ggplot() + geom_sf(data=states, aes(fill=Gini)) +
      scale_fill_gradient(low="gray95", high="gray5")
```

This example demonstrates the ease of creating a droplet map with `ggplot2`'s `geom_sf()`. Map the variable analyzed, the Gini coefficient, to the visual aesthetic of choice, here the `fill` parameter.

Creating a map does not preclude other data visualizations. The distribution of a continuous variable, such as the Gini coefficient, can provide information complementary to a map. In this situation, the scale of the map in Figure 9.4 does not reveal information regarding the District of Columbia. Moreover, regardless of scale, how is the Gini coefficient distributed across the states?

The VBS plot (Gerbing, 2019) from `lessR Plot()` provides one answer: the integrated violin plot, box plot, and scatterplot. Figure 9.5 reveals that the distribution of Gini is approximately normal, with one outlier, the District of Columbia. A second Gini coefficient, for New York state, approaches outlier status as it is near but not beyond the fence, the boundary for labeling a point as an outlier.

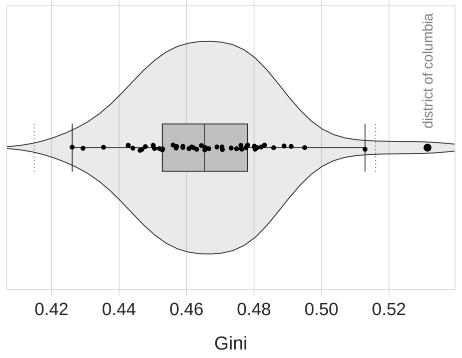


Figure 9.5: VBS plot of the Gini coefficient of income inequality across the 48 contiguous states of the USA plus the District of Columbia.

9.2 Network Visualizations

A *network* consists of nodes and edges that join the nodes. For a social network the nodes are people, and the edges are relationships between the people. An information network shows how information flows between people or organizations, such as a network of email communication among a company's employees. A biological network of feeding relationships from plants to predators that shows which organisms feed off of others is a food web. Transportation networks indicate routes that connect locations.

network: Set of objects called
joined by edges that represent r

Network visualization shows the connections between the nodes, revealing the structure of the network in terms of the interconnected nodes. What nodes are central? What sub-groups exist? Additional analysis tools answer questions such as, for transportation networks, finding the shortest path through the network.

R provides several traditional quality network visualization tools that include the `igraph` (Csardi & Nepusz, 2006) and `network` (Butts, 2008) packages. Although different network software applications represent data in different ways, one traditional method is an *adjacency matrix*, used by both `igraph` and `network`, which is a square matrix with the column and row names the nodes of the network. Each data value of 1 within the matrix indicates that there is a connection between the nodes, and a 0 indicates no connection.

adjacency matrix: Representation of network connections with a square matrix of nodes with data values of 1 or 0 or a connection or not.

A recent development, the approach pursued here, represents network data in terms of traditional data frames or tibbles, so that traditional data manipulation tools such as provided by base R and tidyverse packages such as `dplyr` and `tidyr` can be applied to the analysis of network data. Thomas Lin Pedersen's `tidygraph` (2019) package defines network data in terms of data frames or tibbles. Pedersen's `ggraph` (2018) package works with syntax similar to `ggplot2` to create the network graphs from the `tidygraph` data structure. The graphics software is built on `igraph`, and so easily accesses the power of `igraph` with a `tidyverse/ggplot2` style interface.

9.2.1 Data

The data are for a small demonstration network of only four nodes that generalizes to large networks with thousands of nodes. Consider four cities, generically named City_A through City_D. City_C is central, and directly connects with the other three cities. The only direct connection between Cities A, B and D is between Cities A and D. The network represents commuter trains that travel into the city for a morning commute, in which some of the routes are *assymetric* or directed. Bi-directional service at that time only for Cities

asymmetric relation: The direction of the relation from one node to another is not the same as the relation in the return