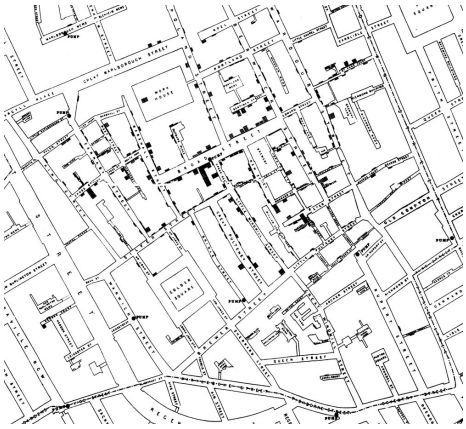
EPI 563: Spatial Epidemiology, Fall 2021

Michael Kramer

Last updated: 2021-08-25

Contents

How to use this eBook



Welcome to *Concepts & Applications in Spatial Epidemiology (EPI 563)*! This eBook is one of several sources of information and support for your progress through the semester. For an overview of the course, expectations, learning objectives, assignments, and grading, please review the full course syllabus on Canvas. This eBook serves to provide a *'jumping off point'* for the content to be covered each week. Specifically, the content herein will introduce key themes, new vocabulary, and provide some additional detail that is complementary to the *asynchronous* (pre-recorded) video lectures, and foundational to the *synchronous* (in class) work.

Strategy for using this eBook

There is a separate *module* or *chapter* for each week's content. In general, the content within each week's section is divided into two sections focusing on **spatial thinking** and **spatial analysis**. This dichotomy does not always hold, but in broad terms you can expect these sections to be more specific to content in class on *Tuesday* versus *Thursday* respectively.

- *Spatial thinking for epidemiology*: This section introduces vocabulary, concepts, and themes that are important to the incorporation of spatialized or geo-referenced data into epidemiologic work. At a minimum, plan to read this content *prior to class Tuesday*, although you will likely benefit from reading both sections before Tuesday.
- *Spatial analysis for epidemiology*: This section is more focused on data management, visualization, spatial statistics, and interpretation. This content is relevant for our work together on Tuesday's, but is essential for successful work in the Thursday lab activities.

Throughout the book some concepts or ideas may be highlighted with *call-out blocks*.

This block denotes a potential pitfall or area of caution.

This block denotes an additional bit of information or additional idea to *note* about the topic at hand.

This block denotes a *tip* or advice for best practices or efficiency.

Please note that I will be continually updating the eBook throughout the semester, so if you choose to download, please double-check the **Last updated** date (in colored bar at bottom of screen) to be sure you have the most recent version.

Part I

Getting ready...

Software installation

The information in this module follows on the pre-class video on setting up `R` and `RStudio` on your computer.

Installing `R` on your computer

As of August 2021, the most up-to-date version of `R` is 4.1.1. The *R Project for Statistical Computing* are continually working to update and improve `R`, and as a result there are new versions 1-2 times per year.

If you already have `R` installed, you can open the console and check your current version by doing this: `R.Version()$version.string`

If you do not have `R` or have an older version than that listed above, you can install `R` by going to the `R` repository: https://www.r-project.org/. Note that there are many 'mirrors' or servers where the software is stored. Generally it is wise to select one that is geographically close to you, although any should work in theory. One mirror that is relatively close to Atlanta is here: http://archive.linux.duke.edu/cran/

Installing *RStudio* on your computer

R-Studio is one of several *integrated development environments* (IDE) for working in `R`. That means it is a *wrapper* around the core `R` functionality that makes coding and project work in `R` much easier than it would be without. We *develop* projects or analyses using `R` within an IDE such as R-Studio. Using R-Studio lets us have robust code-editing and debugging, code syntax highlighting (e.g. coloring different words according to their use, and identifying possible errors), and some assistance with file management, working in larger projects, and outputting results.

The current version of RStudio 1.4.1717. If you do not have RStudio or have a version older than 1.2 please install/update.

TO INSTALL: go to https://www.rstudio.com/products/rstudio/download/

TO UPDATE: Open RStudio and go to Help Menu and choose 'Check for Updates'

- R-Studio Cheatsheat provides a quick reference guide for many of the ways that R-Studio makes your work with `R` easier.

Installing packages for this course

While base `R` has a great deal of essential functionality, most of the power of `R` comes from the rapidly growing list of user-created and contributed 'packages'. A package is simply a bundle of functions and tools, sometimes also including example datasets, basic documentation, and even tutorial 'vignettes'. You can see all the official `R` packages by going here: https://cran.r-project.org/web/packages/.

The most common way to install package in `R` is with the `install.packages()` command. For instance to install the package `ggplot2` you do this:

```
install.packages("ggplot2")
```

Remember that you only need to install a package once (although you may have to update packages occasionally – see the green Update button in the Packages tab in R Studio). When you want to actually *use* a package (for example `ggplot2`) you call it like this:

```
library(ggplot2)
```

If your call to `library()` is working, nothing visible happens. However if you see errors, they might be because your package is out of date (and thus needs to be updated/reinstalled), or because some important *dependencies* are missing. Dependencies are other packages on which this package depends. Typically these are installed by default, but sometimes something is missing. If so, simply install the missing package and then try calling `library(ggplot2)` again.

Notice that for the function `install.packages('yourPackage')` you **must use quotes** around the package name. In contrast for the function `library(youPackage)` you **do not use quotes**.

As you submit each installation request, note the output in your R console. If you get a warning that says installation was not possible because you are missing a package *'namespace'*, that suggests you are missing a dependency (e.g. something the main package needs to work correctly). Try installing the package mentioned in the error. If you have trouble, reach out to the TA's!

Installing `Rtools40` (Windows Only)

While **most** packages can be installed as mentioned above (e.g. using `install.packages()`), there are instances where an installation requires a more complex '*unpacking*' of the course code or installation from github. Mac OS and Unix have the capability of doing this, but on a Windows machine you may require additional *tools*. Luckily there is a package for that! It is called `Rtools40`, and you should install that **before** you install the packages below.

If you are running Windows, navigate to this website: https://cran.r-project.org/bin/windows/Rtools/ and follow the instructions.

Installing packages used for general data science

For the rest of this page, copy and paste the provided code in order to install packages necessary for this course. Notice if you hover to the right of a code-chunk in the html version of the eBook, you will see a *copy* icon for quick copying and pasting.

Although you are copying and pasting the code, take a moment to look at the output. Did you get any error messages that a package did not install? If so, re-check the code or try again.

These packages will support some of our general work in `R`:

- `rmarkdown` allows the creation of mixed output documents that combine code, documentation and results in a single, readable format.
- The packages `tinytex` and `knitr` are necessary for creating the `R` documents including PDF output that will be required for submitting assignments.
- We will use many data manipulation tools from the `tidyverse`. You can learn more about the `tidyverse` here: https://tidyverse.tidyverse.org/, and you can see applications of `tidyverse` packages in the R for

Epidemiologists Handbook. The `tidyverse` is actually a collection of data science tools including the visualization/plotting package `ggplot2` and the data manipulation package `dplyr`. For that reason, when you `install.packages('tidyverse')` below, you are actually installing *multiple* packages!

- The packages `here` and `pacman` are utilities to help simplify file pathnames and package loading behavior.

```
install.packages('tidyverse')
install.packages(c('pacman', 'here'))
install.packages(c('tinytex', 'rmarkdown', 'knitr'))
tinytex::install_tinytex()
# this function installs the tinytex LaTex on your
#  computer which is necessary for rendering (creating) PDF's
```

Installing packages use for geographic data

There are many ways to get the data we want for spatial epidemiology into `R`. Because we often (but don't always) use census geographies as aggregating units, and census populations as denominators, the following packages will be useful. They are designed to quickly extract both geographic boundary files (e.g. *'shapefiles'*) as well as attribute data from the US Census website via an API. **NOTE**: For these to work you have to request a free Census API key. Notice the `help()` function below to get instructions on how to do this.

```
install.packages(c('tidycensus','tigris'))

help('census_api_key','tidycensus')
```

Installing packages used for spatial data manipulation & visualization

This section installs a set of tools specific to our goals of importing, exporting, manipulating, visualizing, and analyzing spatial data.

- The first line of packages have functions for defining, importing, exporting, and manipulating spatial data.
- The second line has some tools we will use for visualizing spatial data (e.g. making maps!).

```
install.packages(c('sp', 'sf', 'rgdal', 'raster', 'RColorBrewer', 'rgeos', 'maptools',
install.packages(c('tmap', 'tmaptools', 'ggmap', 'shinyjs', 'shiny', 'micromap'))
```

Installing packages used for spatial analysis

Finally these are packages specifically to spatial analysis tasks we'll carry out.

```
install.packages(c('spdep', 'CARBayes', 'sparr', 'spatialreg',  'DCluster', 'SpatialEpi
install.packages(c('GWmodel', 'spgwr') )
```

Part II

Weekly Modules

Chapter 1

Locating Spatial Epidemiology

1.1 Getting Ready

1.1.1 Learning objectives

Table 1.1: Learning objectives by weekly module

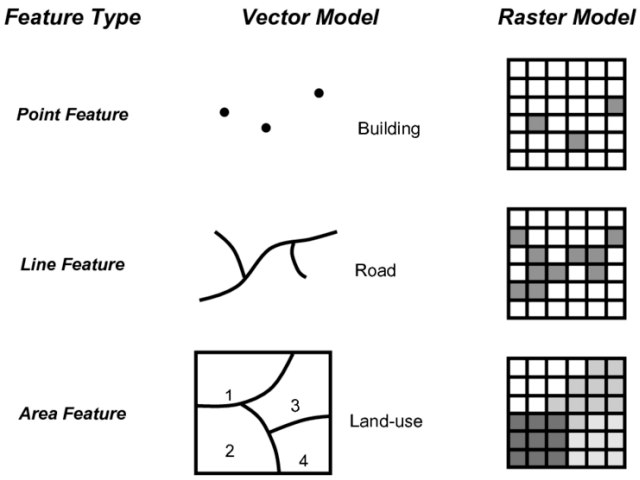
After this module you should be able to...
Explain the potential role of spatial analysis for epidemiologic thinking and practice.
Produce simple thematic maps of epidemiologic data in R.

1.1.2 Additional Resources

- Geocompution with R by Robin Lovelace. This will be a recurring 'additional resource' as it provides lots of useful insight and strategy for working with spatial data in `R`. I encourage you to browse it quickly now, but return often when you have qusetiona about how to handle geogrpahic data (especially of class `sf`) in `R`.
- A basic introduction to the `ggplot2` package. This is just one of dozens of great online resources introducing the *grammar of graphics* approach to plotting in `R`.
- A basic introduction to the `tmap` package This is also only one of many introductions to the `tmap` mapping package. `tmap` builds on the *grammar*

of graphics philosophy of `ggplot2`, but brings a lot of tools useful for thematic mapping!
- R for SAS users cheat sheet

1.1.3 Important Vocabulary



1.2 Spatial Thinking in Epidemiology

When first learning epidemiology, it can be difficult to distinguish between the concepts, theories, and purpose of epidemiology versus the skills, tools, and methods that we use to implement epidemiology. But these distinctions are foundational to our collective professional identity, and to the way we go about doing our work. For instance do you think of epidemiologists as data analysts, scientists, data scientists, technicians or something else? These questions are bigger than we can address in this class, but their importance becomes especially apparent when learning an area such as *spatial epidemiology*. This is because there is a tendency for discourse in spatial epidemiology to focus primarily on the *data* and the *methods* without understanding how each of those relate to the *scientific questions* and *health of population* for which we are ultimately responsible. Distinguishing these threads is an overarching goal of this course, even as we learn the data science and spatial analytic tools.

One quite simplistic but important example of how our questions and methods are inter-related is apparent when we think of **data**. Data is central to quantitative analysis, including epidemiologic analysis. So how is *data* different in *spatial* epidemiology?

The first thing that might come to mind is that we have explicitly geographic or spatial measures contained within our data. The content of the spatial data is distinct: the addition of geographic or spatial location may illuminate otherwise *aspatial* attributes. But even more fundamental than the content is thinking about the *unit of analysis*.

It is likely that in many other examples in your epidemiology coursework, the explicit (or sometimes implicit) unit of analysis has been the individual person. Spatial epidemiology can definitely align with individual-level analysis. But as we'll see, common units we observe and measure in spatial epidemiology – and therefore the units that compose much of our **data** – are not individuals but instead are geographic units (e.g. census tract, county, state, etc) and by extension the *collection* or *aggregation* of all the individuals therein. This distinction in unit of analysis has important implications for other epidemiologic concerns including precision, bias, and ultimately for inference (e.g. the meaning we can make from our analysis), as we'll discuss throughout the semester.

One concrete implication of the above discussion is that you should always be able to answer a basic question about any dataset you wish to analyze: "*what does one row of data represent?*" A row of data is one way to think of the *unit of analysis*, and often (but not always) in spatial epidemiology a row of data is a summary of the population contained by a geographic unit or boundary. Said another way it is an *ecologic summary* of the population. As stated above, this is only the most simplistic example of how and why it is important to not only learn the spatial statistics and methods, but to also maintain the perspective of epidemiology as a population health science. To advance public health we need good methods but we also need critical understanding of the populations we support, the data we analyze, and the conclusions we can reliably draw from our work.

As we move through the semester, I encourage you to dig deep into how methods work, but also to step back and ask questions like *"Why would I choose this method?"* or *"What question in epidemiology is this useful for?"*

1.3 Spatial Analysis in Epidemiology

1.3.1 Spatial data storage formats

If you have worked with spatial or GIS data using ESRI's ArcMap, you will be familiar with what are called *shapefiles*. This is one very common format for storing geographic data on computers. ESRI shapefiles are not actually a single file, but are anywhere from four to eight different files all with the same file name but different extensions (e.g. *.shp*, *.prj*, *.shx*, etc). Each different file (corresponding to an extension) contains a different portion of the data ranging from the geometry data, the attribute data, the projection data, an index connecting it all together, etc.

What you may not know is that shapefiles are not the only (and in my opinion **definitely not the best**) way to store geographic data. In this class I recommend storing data in a format called *geopackages* indicated by the `.gpkg` extension. *Geopackages* are an open source format that were developed to be functional on mobile devices. They are useful when we are storing individual files in an efficient and compact way. To be clear, there are many other formats and I make no claim that *geopackages* are the ultimate format; they just happen to meet the needs for this course, and for much of the work of spatial epidemiologists. It is worth noting that many GIS programs including ArcMap and QGIS can both read and write the *geopackage* format; so there is no constraint or limitation in terms of software when data are stored in `.gpkg` format.

1.3.2 Representing spatial data in `R`

The work in this course assumes that you are a *basic `R` user*; you do not need to be expert, but I assume that you understand data objects (e.g. `data.frame`, `list`, `vector`), and basic operations including subsetting by index (e.g. using square brackets to extract or modify information: `[]`), base-R plotting, and simple modeling. If you **are not familiar with `R`**, you will need to do some quick self-directed learning.

Here are some good online resources for R skills, and the instructor and TA's can point you to additional resources as needed:

- The Epidemiologist R Handbook
- R for Data Science, particularly the introductory chapters
- R Tutorial

Just as our conceptualization of, or thinking about *data* in spatial epidemiology requires some reflection, the actual structure and representation of that data with a computer tool such as `R` also requires some attention. Specifically, spatial data in `R` is not exactly like the conventional *aspatial* epidemiologic data that is often arranged as a rectangular `data.frame` (e.g. like a spreadsheet where *rows are observations* and *columns are variables*). While spatial data are more complex than just a spreadsheet, it *does not* need to be as complex as spatial data in software platforms like ESRI's ArcMap.

To be *spatial*, a dataset must have a representation of geography, spatial location, or spatial relatedness, and that is most commonly done with either a *vector* or *raster* data model (see description above in vocabulary). Those spatial or geographic representations must be stored on your computer and/or held in memory, hopefully with a means for relating or associating the individual locations with their corresponding attributes. For example we want to know the

attribute (e.g. the count of deaths for a given place), and the location of that place, and ideally we want the two connected together.

Over the past 10+ years, `R` has increasingly been used to analyze and visualize spatial data. Early on, investigators tackling the complexities of spatial data analysis in `R` developed a number of ad hoc, one-off approaches to these data. This worked in the short term for specific applications, but it created new problems as users needed to generalize a method to a new situation, or chain together steps. In those settings it was not uncommon to convert a dataset to multiple different formats to accomplish a task sequence; this resulted in convoluted and error-prone coding, and lack of transparency in analysis.

An eventual response to this early tumult was a thoughtful and systematic approach to defining a *class of data* that tackled the unique challenges of spatial data in `R`. Roger Bivand, Edzer Pebesma and others developed the `sp` package which defined spatial data classes, and provided functional tools to interact with them. The `sp` package defined specific data classes to hold points, lines, and polygons, as well as raster/grid data; each of these data classes can contain geometry only (these have names like `SpatialPoints` or `SpatialPolygons`) or could contain geometry plus related data attributes (these have names like `SPatialPointsDataFrame` or `SpatialPolygonsDataFrame`). Each spatial object can contain all the information spatial data might include: the spatial extent (min/max x, y values), the coordinate system or spatial projection, the geometry information, the attribute information, etc.

Because of the flexibility and power of the `sp*` class of objects, they became a standard up until the last few years. Interestingly, it was perhaps the sophistication of the `sp*` class that began to undermine it. `sp*` class data was well-designed from a programming point of view, but was still a little cumbersome (and frankly confusing) for more applied analysts and new users. Analysis in *spatial epidemiology* is not primarily about computer programming, but about producing transparent and reliable data pipelines to conduct valid, reliable, and reproducible analysis. Thus epidemiologists, and other data scientists, desired spatial tools that could be incorporated into the growing toolbox of data science tools in `R`.

These calls for a more user-friendly and intuitive approach to spatial data led the same team (e.g. Bivand, Pebesma, others) to develop the *Simple Features* set of spatial data classes for `R`. Loaded with the `sf` package, this data format has quickly become the standard for handling spatial data in `R`. The power of the `sf` class, as discussed below, is that it makes *spatial data* behave like *rectangular data* and thus makes it amenable to manipulation using any tool that works on `data.frame` or `tibble` objects. Recognizing that many users and functions prefer the older `sp*` objects, the `sf` package includes a number of utility functions for easily converting back and forth.

In this class we will use `sf*` class objects as the preferred data class, but because some of the tools we'll learn still require `sp*` we will

occasionally go back and forth.

`sf*` data classes are designed to hold all the essential spatial information (projection, extent, geometry), but do so with an easy to evaluate `data.frame` format that integrates the attribute information and the geometry information together. The result is more intuitive sorting, selecting, aggregating, and visualizing.

1.3.3 Benefits of `sf` data classes

As Robin Lovelace writes in his online eBook, Gecomputation in R, `sf` data classes offer an approach to spatial data that is compatible with QGIS and PostGIS, important non-ESRI open source GIS platforms, and `sf` functionality compared to `sp` provides:

1. Fast reading and writing of data
2. Enhanced plotting performance
3. `sf` objects can be treated as data frames in most operations
4. `sf` functions can be combined using `%>%` pipe operator and works well with the `tidyverse` collection of `R` packages (see Tips for using `dplyr` for examples)
5. `sf` function names are relatively consistent and intuitive (all begin with `st_`)

1.3.4 Working with spatial data in `R`

Here and in lab, one example dataset we will use, called `ga.mvc` quantifies the counts and rates of death from motor vehicle crashes in each of Georgia's $n = 159$ counties. The dataset is *vector* in that it represents counties as polygons with associated attributes (e.g. the mortality information, county names, etc).

1.3.4.1 Importing spatial data into `R`

It is important to distinguish between two kinds of data formats. There is a way that data is *stored on a computer hard drive*, and then there is a way that data is organized and managed *inside a program* like `R`. The *shapefiles* (`.shp`) popularized by ESRI/ArcMap is an example of a format for storing spatial data on a hard drive. In contrast, the discussion above about the `sf*` and `sp*` data classes refer to how data is organized *inside R*.

Luckily, regardless of how data is *stored* on your computer, it is possible to import almost any format into `R`, and once inside `R` it is possible to make it into either the `sp*` or `sf*` data class. That means if you receive data as a `.shp`

shapefile, as a `.gpkg` geopackage, or as a `.tif` raster file, each can be easily imported.

All `sf` functions that act on spatial objects begin with the prefix `st_`. Therefore to import (read) data we will use `st_read()`. This function determines **how** to import the data based on the extension of the file name you specify. Look at the help documentation for `st_read()`. Notice that the first argument `dsn=`, might be a complete file name (e.g. `myData.shp`), or it might be a folder name (e.g. `mygeodatabase.gdb`). So if you had a the motor vehicle crash data saved as both a shapefile (`mvc.shp`, which is actually six different files on your computer), and as a geopackage (`mvc.gpkg`) you can read them in like this:

```
# this is the shapefile
mvc.a <- st_read('GA_MVC/ga_mvc.shp')

# this is the geopackage
mvc.b <- st_read('GA_MVC/ga_mvc.gpkg')
```

We can take a look at the defined data class of the imported objects within `R`:

```
class(mvc.a)
```

```
## [1] "sf"         "data.frame"
```

```
class(mvc.b)
```

```
## [1] "sf"         "data.frame"
```

Notice how the two objects have the same *class* (e.g. type of data stored within `R`), even though they were two different kinds of files stored on the computer: one was a shapefile and one a geopackage. This is because `st_read()` can automatically detect the storage format based on the extension, and use the appropriate interpreter to import that data. This is nice because it means you can bring many types of spatial data into `R`!

You will also notice that when we examined the `class()` of each object, they are each classified as **both `sf` and `data.frame`** class. That is incredibly important, and it speaks to an elegant simplicity of the `sf*` data classes! That it is classified as `sf` is perhaps obvious because it is a *spatial* object; but the fact that each object is *also classified* as `data.frame` means that we can treat the object for the purposes of data management, manipulation and analysis as a relatively simple-seeming object: a rectangular `data.frame`. How does that work? We

will explore this more in lab but essentially each dataset has rows (observations) and columns (variables).

We can see the variable/column names like this:

```
names(mvc.a)
```

```
## [1] "GEOID"      "NAME"       "MVCRATE_17" "geometry"
```

```
names(mvc.b)
```

```
## [1] "GEOID"      "NAME"       "MVCRATE_17" "geom"
```

We can see that each dataset has the same *attribute* variables (e.g. `GEOID`, `NAME`, `MVCRATE_17`), and then a final column called `geometry` in one and called `geom` in another.

These geometry columns are different from your usual run-of-the-mill column variables in that they don't hold a single value. Instead, each *'cell'* in those columns actually contains an embedded list of x, y coordinates defining the vertices of the polygons for each of Georgia's counties. So all of the *spatial location* information for each row is contained in that single variable called `geom` (or alternately, `geometry`).

Another way to learn about an `sf` object is to use the `head()` function. In addition to displaying the top few rows of data (which is the typical behavior of the `head()` function), for `sf` objects `head()` will also print some of the important *metadata* about the file.

```
head(mvc.a)
```

```
## Simple feature collection with 6 features and 3 fields
## Geometry type: MULTIPOLYGON
## Dimension:     XY
## Bounding box:  xmin: -84.64195 ymin: 31.0784 xmax: -82.04858 ymax: 34.49172
## Geodetic CRS:  WGS 84
##   GEOID                      NAME MVCRATE_17                       geometry
## 1 13001  Appling County, Georgia   53.99276 MULTIPOLYGON (((-82.55069 3...
## 2 13003 Atkinson County, Georgia   35.96260 MULTIPOLYGON (((-83.141 31....
## 3 13005    Bacon County, Georgia    0.00000 MULTIPOLYGON (((-82.62819 3...
## 4 13007    Baker County, Georgia   31.25000 MULTIPOLYGON (((-84.64166 3...
## 5 13009  Baldwin County, Georgia   28.94936 MULTIPOLYGON (((-83.42674 3...
## 6 13011    Banks County, Georgia   32.19921 MULTIPOLYGON (((-83.66862 3...
```

To summarize, `sf` within `R` is powerful because:

1. We are not limited to how data arrive to us. If you acquire data (from the web, a colleague, etc) as a shapefile, a geopackage, a raster or other formats, they can all be imported into `R`.
2. Once inside of `R` (and stored in `sf` data objects), we can treat the datasets almost as if they were aspatial, rectangular datasets. That means we could use subsetting, filtering, recoding, merging, and aggregating *without losing the spatial information*!

1.3.4.2 Exporting spatial data from `R`

While importing is often the primary challenge with spatial data and `R`, it is not uncommon that you might modify or alter a spatial dataset and wish to save it for future use, or to write it out to disk to share with a colleague. Luckily the `sf` package has the same functionality to write an `sf` spatial object to disk in a wide variety of formats including *shapefiles* (`.shp`) and *geopackages* (`.gpkg`). Again, `R` uses the extension you specify in the filename to determine the target format.

```
# Write the file mvc to disk as a shapefile format
st_write(mvc, 'GA_MVC/ga_mvc_v2.shp')

# Write the file mvc to disk as a geopackage format
st_write(mvc, 'GA_MVC/ga_mvc_v2.gpkg')
```

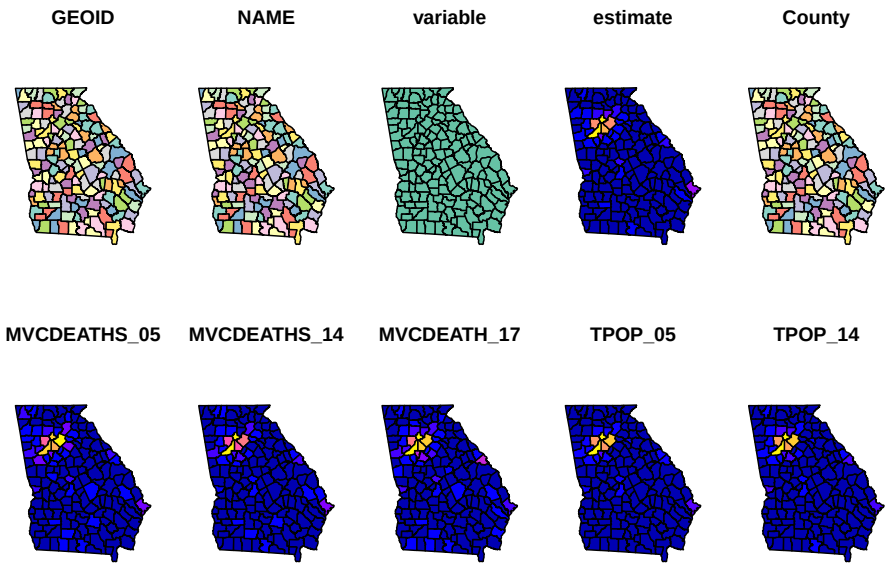
1.3.5 Basic visual inspection/plots

What if you want to **see** your spatial data? In base-R there is a powerful function called `plot()` that can be used to create easy or incredibly complex visualizations or graphical representation of data. In the package `sf`, the functionality of `plot()` is extended to handle the uniqueness of spatial data. That means that if you call `plot()` on a spatial object **without having loaded** `sf`, the results will be different than if `plot()` called **after loading** `sf`.

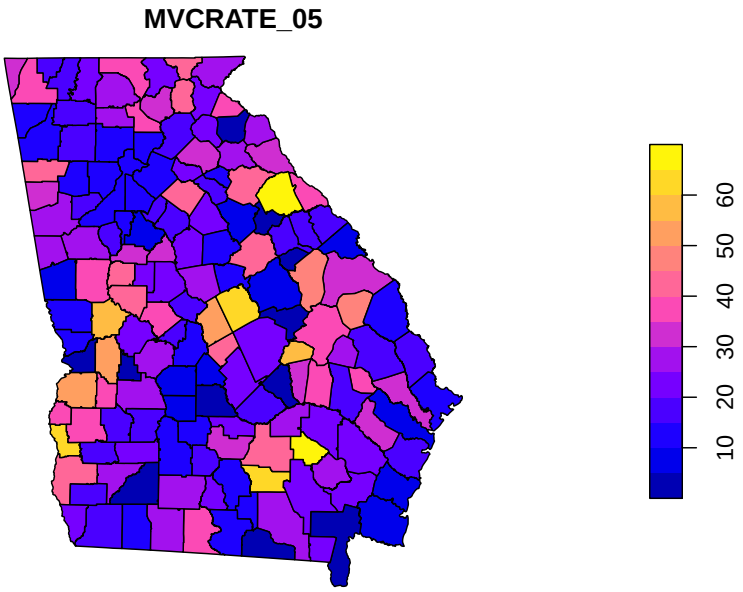
When you `plot()` with `sf`, by default it will try to make a map **for every variable in the data frame**! Try it once. If this is not what you want (it usually *is not*), you can force it to only plot *some* variables by providing a vector of variable names.

```
plot(mvc) # this plots a panel for every column - or actually the first 10 columns
```

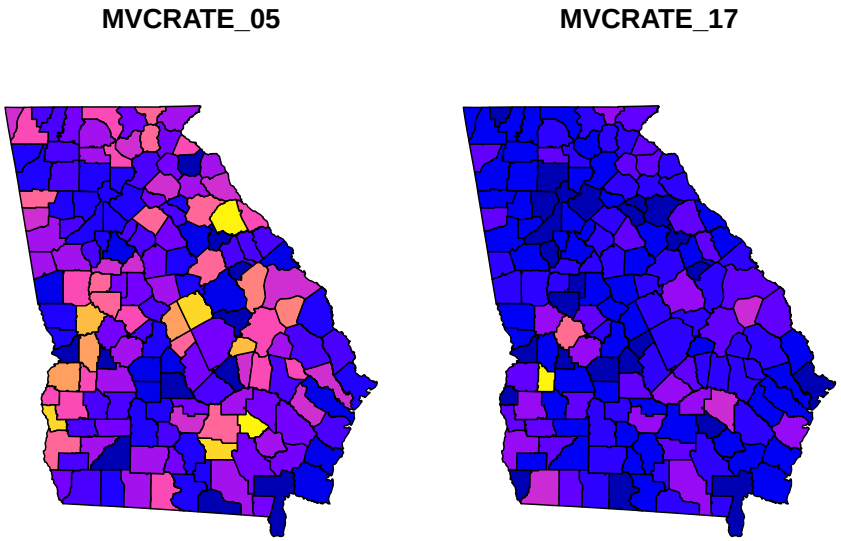
```
## Warning: plotting the first 10 out of 17 attributes; use max.plot = 17 to plot
## all
```




```
plot(mvc['MVCRATE_05']) # this plots only a single variable, the MVC mortality rate fo
```




```
plot(mvc[c('MVCRATE_05', 'MVCRATE_17')]) # this plots two variables: MVC rate in 2005 & 2017
```

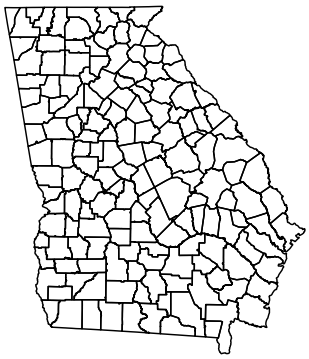


Sometimes you want to know something about the spatial size, extent, or shape

of your data. To do this you can easily plot only the geometry of the spatial object (e.g. not attributes). Here are two approaches to quickly plot the geometry:

```
plot(st_geometry(mvc)) # st_geometry() returns the geom information to plot
```

```
plot(mvc$geom)  # this is an alternative approach...directly plot the 'geom' column
```



1.3.6 Working with CRS and projection

Maps are used to describe the geographical or spatial location of particular objects as a representation of where those things are on planet Earth. Most maps are printed on paper or screens. In other words, maps identify locations from somewhere on planet earth and represent them as flat.

But the world does not have latitude or longitude lines painted on the ground, and the earth is not flat! Instead the earth is nearly spherical (really it is a *geoid*) and there is no universal reference for where to start measuring.

For these two reasons, all maps require at a minimum a *coordinate reference system* (CRS) to define how the numbers in our coordinates relate to actual places. In addition most maps are best interpreted after formally *projecting* the data to account for the artifact induced by pretending earth is flat.

The most unambiguous way to describe a CRS and/or projection is by using the **EPSG** code, which stands for *European Petroleum Survey Group*. This consortium has standardized hundreds of projection definitions in a manner adopted by several `R` packages including `rgdal` and `sf`.

A given dataset already has a CRS (and possibly a projection). If CRS and projection information was contained in the original file you imported, it will *usually* be maintained when you use `st_read()`. However sometimes it is missing and you must first find it. Once it is known, you might choose to *change* or *transform* the CRS or projection for a specific purpose. We will discuss this further in class.

If there is NO CRS information imported it is critical that you find out the CRS information from the data source!

This course is not a GIS course (e.g. it is assumed you have already had some exposure to geographic information systems generally), and learning about the theory and application of coordinate reference systems and projections is not our primary purpose this semester. However some basic knowledge *is necessary* for successfully working with spatial epidemiologic data. Here are several resources you should peruse to learn more about CRS, projections, and EPSG codes:

- A useful overview/review of coordinate reference systems in `R`
- Robin Lovelace's Geocompuation in R on projections with `sf`
- EPSG website: This link is to a searchable database of valid ESPG codes
- Here are some useful EPSG codes

Figure 1.1: Comparing CRS

The choice of CRS and/or projection has a substantial impact on how the rendered map looks, as is evident in the figure above (source of image).

We already saw the CRS/projection information of the `mvc` object when we used the `head()` function above; it was at the top and read `WGS 84`.

Recall there are two main types of CRS: purely **geographic** which is to say coordinate locations are represented as *latitude* and *longitude* degrees; and **projected** which means the coordinate values have been transformed for representation of the spherical geoid onto a planar (Euclidean) coordinate system. `WGS 84` is a ubiquitous geographic coordinate system common to boundary files retrieved from the U.S. Census bureau.

An important question when you work with a spatial dataset is to understand whether it is primarily a geographic or projected CRS, and if so which one.

```
st_is_longlat(mvc)
```

```
## [1] TRUE
```

This quick logical test returns `TRUE` or `FALSE` to answer the question *"Is the `sf` object simply a longitude/latitude geographic CRS?"*. The answer in this case is `TRUE` because `WGS 84` is a geographic (longlat) coordinate system. But what if it were `FALSE` or we wanted to know more about the CRS/projection?

```
st_crs(mvc)
```

```
## Coordinate Reference System:
##   User input: WGS 84
##   wkt:
## GEOGCRS["WGS 84",
##     DATUM["World Geodetic System 1984",
##         ELLIPSOID["WGS 84",6378137,298.257223563,
##             LENGTHUNIT["metre",1]]],
##     PRIMEM["Greenwich",0,
##         ANGLEUNIT["degree",0.0174532925199433]],
##     CS[ellipsoidal,2],
##         AXIS["geodetic latitude (Lat)",north,
##             ORDER[1],
##             ANGLEUNIT["degree",0.0174532925199433]],
##         AXIS["geodetic longitude (Lon)",east,
##             ORDER[2],
##             ANGLEUNIT["degree",0.0174532925199433]],
##     USAGE[
##         SCOPE["Horizontal component of 3D system."],
##         AREA["World."],
##         BBOX[-90,-180,90,180]],
##     ID["EPSG",4326]]
```

This somewhat complicated looking output is a summary of the CRS stored with the spatial object. There are two things to note about this output:

- At the top, the *User input* is `WGS 84`
- At the bottom of the section labeled `GEOGCRS` it says `ID["EPSG",4326]`

While there are literally hundreds of distinct EPSG codes describing different geographic and projected coordinate systems, for this semester there are three worth remembering:

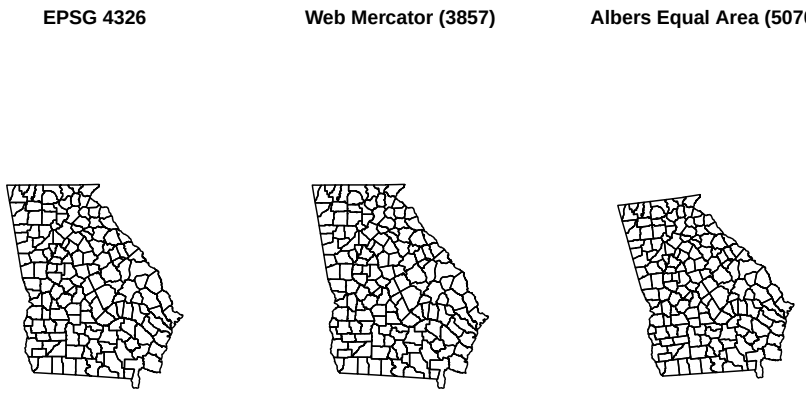
- **EPSG: 4326** is a common geographic (unprojected or long-lat) CRS
- **EPSG: 3857** is also called *WGS 84/Web Mercator*, and is the dominant CRS used by Google Maps
- **EPSG: 5070** is the code for a projected CRS called *Albers Equal Area* which has the benefit of representing the visual area of maps in an equal manner.

Once the CRS/projection is clearly defined, you may choose to transform or *project* the data to a different system. The `sf` package has another handy function called `st_transform()` that takes in a spatial object (dtaaset) with one CRS and outputs that object *transformed* to a new CRS.

```
# This uses the Albers equal area USA,
mvc.aea <- st_transform(mvc, 5070)

# This uses the Web Mercator CRS (EPSG 3857) which is just barely different from EPSG 4326
mvc.wm <- st_transform(mvc, 3857)

# Now let's look at them side-by-side
plot(st_geometry(mvc), main = 'EPSG 4326')
plot(st_geometry(mvc.wm), main = 'Web Mercator (3857)')
plot(st_geometry(mvc.aea), main = 'Albers Equal Area (5070)')
```



Do you see the difference between the three? Because EPSG 4326 and 3857 are both unprojected (e.g. they are long/lat), they appear quite similar but are not identical. Albers Equal Area, on the other hand, is more distinct. In general we will prefer to use *'projected'* rather than *'unprojected'* (long/lat only) data for both visualization and analysis. That means that whenever you bring in a new dataset you will need to check the CRS and project or transform as desired.

Important: It is important to distinguish between defining the current projection of data and the act of *projecting* or *transforming* data from one known system to a new CRS/projection.
We cannot transform data until we correctly define its current or original CRS/projection status. The above function tells us what the current status is. In some cases data do not have associated CRS information and this might be completely blank (for instance if you read in numerical x, y points from a geocoding or GPS process). In those cases you can **set** the underlying CRS using `st_set_crs()` to define it, but this assumes you **know** what it is. There are two arguments to this function: the first is `x = objectName`, and the second is `value = xxx` where *'xxx'* is a valid EPSG code.

Table 1.2: Vocabulary for Week 1

Term	Definition
Data, attribute	Nonspatial information about a geographic feature in a GIS, usually stored in a table and linked to the feature by a unique identifier. For example, attributes of a county might include the population size, density, and birth rate for the resident population
Data, geometry	Spatial information about a geogrpahic feature. This could include the x, y coordinates for points or for vertices of lines or polygons, or the cell coordinates for raster data
Datum	The reference specifications of a measurement system, usually a system of coordinate positions on a surface (a horizontal datum) or heights above or below a surface (a vertical datum)
Geographic coordinate system	A reference system that uses latitude and longitude to define the locations of points on the surface of a sphere or spheroid. A geographic coordinate system definition includes a datum, prime meridian, and angular unit
Geopackage	A data storage format suitable for containing vector or raster data in a compact and portable file. It is an alternative (and in my opinion a superior alternative!) to ESRI shapefiles.
	A method by which the curved surface of the earth is portrayed on a flat surface. This generally requires a systematic mathematical transformation of the earth's graticule of lines of longitude and latitude onto a plane. Some projections can be visualized as a transparent globe with a light bulb at its center (though not all projections emanate from the globe's center) casting lines of latitude and longitude onto a sheet of paper. Generally, the paper is either flat and placed tangent to the globe (a planar or azimuthal

Chapter 2

Cartography for Epidemiology I

2.1 Getting Ready

2.1.1 Learning objectives

Table 2.1: Learning objectives by weekly module

After this module you should be able to...
Design a cartographic representation of epidemiologic data that is consistent with best practices in public health data visualization.
Apply data processing functions to accomplish foundational data management and preparation for spatial epidemiology (e.g. summarize, aggregate, combine, recode, etc)

2.1.2 Additional Resources

- CDC Guidance for Cartography of Public Health Data (complements required reading)
- When Maps Lie
- Color Brewer Website for color guidance in choropleth maps

2.1.3 Important Vocabulary

2.2 Spatial Thinking in Epidemiology

Making pretty maps is not the full extent of spatial epidemiology. However, *epidemiologic cartography* can sometimes be the beginning and end of *spatial epidemiology* for a given purpose. And even when an epidemiologic analysis goes well beyond mapping (perhaps to incorporate *aspatial* analysis, or to incorporate more sophisticated *spatial* analysis), the ability to produce a clear, concise, and interpretable map is an important skill.

As Robb, et al[1] write:

> Disease mapping can be used to provide *visual cues* about disease etiology, particularly as it relates to environmental exposures….Mapping where things are allows visualization of a baseline pattern or spatial structure of disease, potential detection of disease clusters, and the initial investigation of an exposure-disease relationship.

There are aspects of cartography and map design that are general to most thematic maps of quantitative data. But there are some issues that seem especially pertinent to us as epidemiologists or quantitative population health scientists. These include the decisions we make about color choice and the process of categorizing numerical data for visual representation in a map.

Why are these especially important for epidemiology? A primary purpose of a map is to visually represent *something meaningful* about the *spatial or geographic variation* in health or a health-relevant feature (e.g. an exposure or resource). Communicating what is *meaningful* and representing *variation* that matters is not solely a technical GIS task; it requires epidemiologic insight.

For instance our approach to representing ratio measures such as an *odds ratio* or *risk ratio* should be different from how we represent *risk* or *rate* data, because we understand that the scale and units are distinct in each case. Similarly, we understand that characterizing variation or heterogeneity in a *normal* or *Gaussian* (bell-shaped curve) distribution is different from a uniform or a highly skewed distribution with a long right tail. This insight into how scales and values are differently interpreted epidemiologically must be translated into sensible choices in mapping.

[1] Robb SW, Bauer SE, Vena JE. Integration of Different Epidemiological Perspectives and Applications to Spatial Epidemiology. Chapter 1 in Handbook of Spatial Epidemiology. 2016. CRC Press, Boca Raton, FL.

2.2.1 Color choices

For most thematic maps, color is the most flexible and important tools for communication. Color, hue, and contrast can accentuate map elements or themes and minimize others. The result is that you can completely change the story your map tells with seemingly small changes to how you use color. This means you should be clear and explicit about *why you choose* a given color or sequence of colors, and beware of unintentionally misrepresenting your data by your color choices.

In producing choropleth maps, we often talk about collections of colors as *color ramps* or *color palettes*, because a single color by itself is not very interesting. A quick scan of either the `tmaptools::palette_explorer()` utility, or the Color Brewer website will demonstrate that there are many colors to choose from, so is it just a matter of preference? Perhaps, but there are some guidelines to keep in mind.

2.2.1.1 Sequential palettes

All color palettes use the color hue, value, or saturation to represent or symbolize the values of the underlying statistical parameter of interest. When a parameter or statistic is naturally ordered, sequential and monotonic, then it makes sense to choose colors that range from light to dark. Conventionally *lighter* or more neutral tones represent lower or smaller numbers and *darker* colors and more intense tones represent higher or larger numbers. The dark colors jump out at the viewer more readily, so occasionally the inverse is used to emphasize small values, but this should be done with caution as it can be counter intuitive.

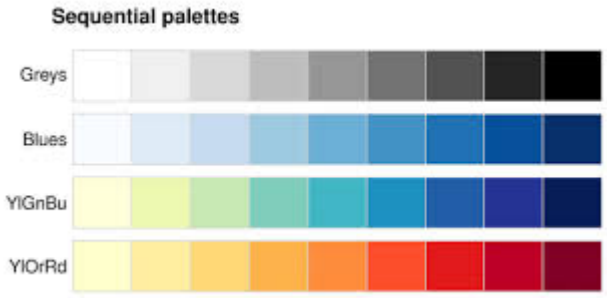


Figure 2.1: Sequential color palettes

Sequential palettes are useful for epidemiologic parameters such as prevalence, risk, or rates, or continuous exposure values where the emphasis is on distinguishing higher values from lower values.

2.2.1.2 Diverging palettes

A less common choice, but one that is especially important for some epidemiologic parameters, is the diverging palette. In this pattern, the *neutral color* is in the center of the sequence, with two different color hues become darker and more intense as they go out from the center.

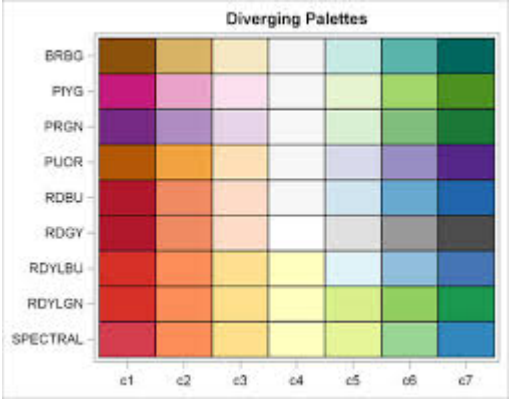


Figure 2.2: Diverging color palettes

You might choose this color sequence for one of two reasons:

1. You wish to show how units vary around the overall mean or median, highlighting those that are *larger than* versus *smaller than* the overall mean/median. For instance diverging palettes might emphasize areas with particularly high burden of disease (and therefore in need of additional attention), as well as those with unexpectedly low burden of disease (and therefore worthy of understanding about protective factors).
2. You are mapping *any epidemiologic measure of effect (e.g. ratio or difference measure)* where there are values both above and below the null ratio of 1.0 (for ratio) or 0 (for difference). For example if you map *Standardized Mortality/Morbidity Ratios*, *risk or odds ratios*, or *prevalence ratios*, you potentially have diverging data. The exception would be if all of the ratio values were on the same side of the null (e.g. all were $>> 1$ or $<< 1$).

In the map above, the SMR (a ratio of the county-specific prevalence of very low birth weight infants to the overall statewide live birth prevalence) varies from 0.13 to 2.30. In other words, a county with an SMR of 1.0 has the *average* prevalence of very low birthweight. This range of data is not sequential in the same way as a *risk* or *prevalence*. Instead the neutral color is assigned to counties in the range of $0.90 - 1.10$, around the null. This is a way of indicating these counties are *average* or *typical*. In contrast, counties with increasing *excess morbidity* have darker green, and substantially *lower morbidity* are darker purple.

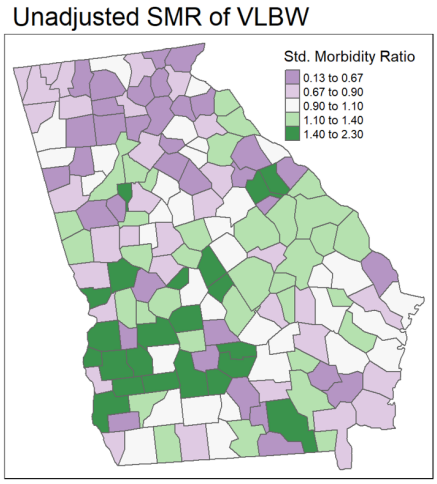


Figure 2.3: Mapping ratio measure with divergent palette

2.2.1.3 Qualitative palettes

Qualitative refers to categories that are not naturally ordered or sequential. For instance if counties were assigned values for the single leading cause of death in the county, we might choose a qualitative palette, as a sequential or diverging palette might mislead the viewer into thinking there is some natural ordering to which causes should be more or less intense in their color.

2.2.2 Choropleth binning styles

A second topic relevant to the intersection of *cartography* and *epidemiologic thinking* is the means by which we choose cut-points for visualizing data. In other words for a map to visually represent some underlying statistical value, we have to assign or map numerical values to colors. How you do that depends greatly on the intended message or story your map needs to tell. Are you interested in distinguish units that rank higher or lower in values? Or are you primarily focused on finding extreme outliers, with variation in the '*middle*' of the distribution of less interest? These distinct purposes give rise to different decisions about how to assign colors to numerical values in your data.

As discussed in the lecture, there are numerous methods or styles for categorizing continuous data for choropleth mapping (e.g. identical data is summarized under four different styles in figure above). Cynthia Brewer (of ColorBrewer fame) and Linda Pickle (**?**) sought to evaluate which styles are most effective for communicating the spatial patterns of epidemiologic data.

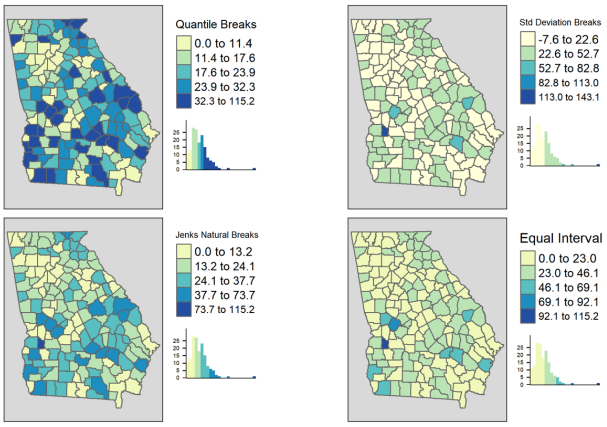


Figure 2.4: Comparing binning styles with same data

As cartographers, Brewer & Pickle were critical of the epidemiologists' over-reliance on quantile cut-points, given many other strategies that seemed to have cartographic advantages. However, after randomizing map '*readers*' to interpret maps of the same underlying epidemiologic data using *seven different styles*, they determined that readers could most accurately and reliably interpret the disease patterns in maps using *quantile cut-points*. While there are benefits of the other styles for some purposes, for the common use of communicating which spatial areas *rank higher or lower* in terms of disease burden, quantiles are most straightforward.

2.2.2.1 Mapping time series

It is common in spatial epidemiology that we want to map the spatial patterns of disease for several different snapshots in time as a series to observe the evolution of disease burden over time. But changing patterns over time raises additional questions about how to make cuts to the data. There are several options for determining the cut-points when you have a time series:

1. Pool all of the years data together before calculating the cut-points (e.g. using quantiles). Use the pooled cut-points for all years.
2. Create custom year-specific cut-points that reflect the distribution of data for each year separately.
3. Create cut-points based on a single year and apply them to all other years.

The map above of Georgia motor vehicle crash mortality data in three different years (2005, 2014, 2017), was created in `tmap` using the `tm_facet()` option where the the `by =` was year. As a result, the quantile cut-points represent

Figure 2.5: Georgia MVC deaths by year with a common scale

the breaks *pooling all observations across the three years*. In other words the cut-points come from 159 counties times three years: 477 values.

By having a common legend that applies to all three maps, this strategy is useful for comparing *differences in absolute rates* across years.

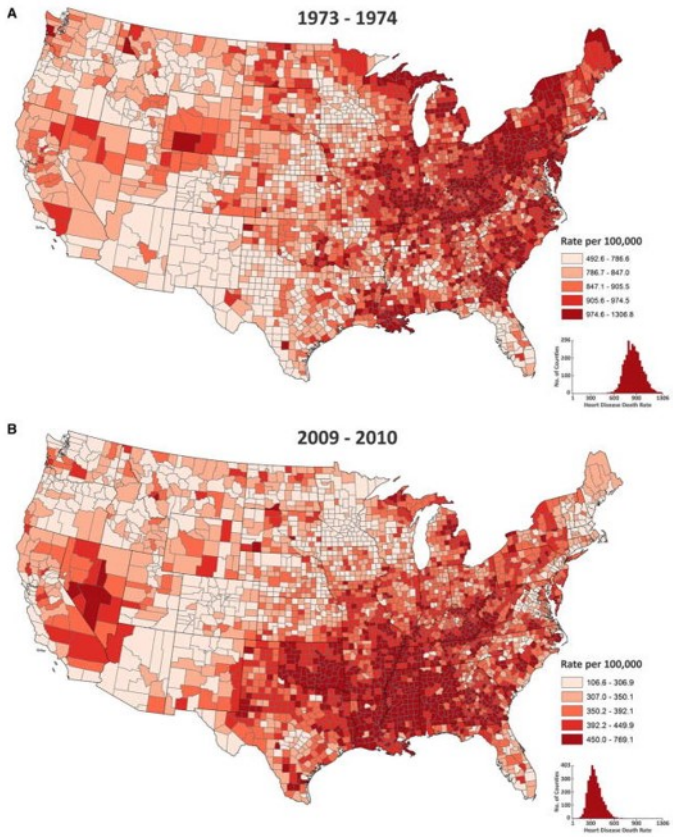


Figure 2.6: U.S. heart disease mortality with a year-specific scales

The map above of heart disease mortality rates by county in two years (1973-4; 2009-10) uses quantile breaks calculated *separately for each time period*. This was done in part because the heart disease mortality rate declined so much between these years that a scale that distinguished highs from lows on one map would not distinguish anything on the other map. In this case what is being compared is not the *absolute rates* but the *relative ranking of counties* in the two years.

2.3 Spatial Analysis in Epidemiology

Every spatial epidemiology project must include attention to data acquisition, cleaning, integration, and visualization. The specific workflow is driven largely by the overarching epidemiologic question, purpose, or goal. In this section we use a specific question to illustrate key steps to data preparation for epidemiologic cartography.

> **Case Example Objective**: Create a choropleth map visualizing geographic variation in the all-cause mortality rate for U.S. counties in 2016-2018. Compare this to a choropleth map of % uninsured in U.S. counties.

This objective will be directly relevant for the lab this week as well as for the *Visualizing US Mortality, Visual Portfolio*, an assignment due later in the semester.

Although this specific question dictates specific data needs, these four types of data are frequently needed to produce a map of a health outcome or state:

1. **Numerator data**, in this case representing the count of deaths per county in the target year
2. **Denominator data**, in this case representing the population at risk for death in each county in the target year
3. **Contextual or covariate data**, in this case the prevalence uninsured for each U.S. county
4. **Geometric data** representing the shapes and boundaries of U.S. counties

2.3.1 Obtaining and preparing numerator data

The event of interest (e.g. the numerator in a risk, rate, or prevalence) can come from many sources. If you are conducting primary data collection, it arises from your study design and measurement. When using secondary data, it is common to use surveillance data (e.g. vital records, notifiable diseases, registries, etc) or administrative data as a source of health events.

When using secondary data sources owned or managed by another entity, one challenge that can occur is *suppression of data* to protect privacy. For example the National Center for Health Statistics mortality data available from CDC WONDER suppresses the count of deaths, as well as the crude mortality rate, whenever the *numerator count is less than ten events*. There can also be instances when a local or state public health agency fails to report data to NCHS, producing missing values.

Depending on the data format, it is possible that either **missing** or **suppressed** data could be inadvertently imported into R as *zero-count* rather than missing. It is therefore critically important to understand the data source and guidelines. The decision about how to manage zero, missing, and suppressed data is an epidemiologic choice, but one that must be addressed *before creating a map*.

How to deal with data suppression. There are many reasons your target data may fall below thresholds for suppression. Perhaps the outcome event is quite rare, or you are stratifying by multiple demographic factors, or perhaps you are counting at a very small geographic unit. If suppression is problematic for mapping, consider pooling over multiple years, reducing demographic stratification, or using larger geographic areas to increase event count and reduce the number of suppressed cells.

For this example, we have downloaded all-cause mortality counts by county from CDC WONDER for 2016-2018 (pooling over three years to reduce suppression). In Lab we will discuss the procedure for acquiring data from the web. After importing the data this is how it appears.

```
head(death)
```

```
##    FIPS             County Deaths Population     crude
## 1 01001 Autauga County, AL    536      55601  964.0114
## 2 01003 Baldwin County, AL   2357     218022 1081.0836
## 3 01005 Barbour County, AL    312      24881 1253.9689
## 4 01007    Bibb County, AL    276      22400 1232.1429
## 5 01009  Blount County, AL    689      57840 1191.2172
## 6 01011 Bullock County, AL    112      10138 1104.7544
```

2.3.2 Obtaining and preparing denominator or contextual data

The mortality data accessed from CDC included both numerator (count of deaths) and denominator (population at risk). However there are instances where you may have one dataset that provides the health event data (numerator), but you need to link it to a population denominator in order to calculate risk, rate, or prevalence.

The U.S. Census Bureau maintains the most reliable population count data for the U.S., and it is available in aggregates from Census Block Group, Census Tract, Zip code tabulation area, City or Place, County, State, and Region.

Census data can be aggregated as total population or stratified by age, gender, race/ethnicity, and many other variables. The census data also contains measures of social, economic, and housing attributes which may be relevant to measure *context* or *exposures* in spatial epidemiologic analyses. There are two broad types of data demographic and socioeconomic data released by the Census Bureau.

- **Decennial Census** tables which (theoretically) count 100% of the population every 10 years. These can be cross-classified by age, race/ethnicity, sex, and householder status (e.g. whether head of house owns or rents and how many people live in house)
- **American Community Survey (ACS)** tables which provide a much larger number of measures but are based on samples rather than complete counts. The ACS began in the early 2000's and is a continually sampled survey. Despite being collected every year, for many small areas (e.g. census tracts or even counties) there are not enough responses in a single year to make reliable estimates. Therefore ACS data pooled into 5-year moving-window datasets. For instance the 2015-2019 ACS (the most recent release) reports estimates for all responses collected during that time period, and these are available from the Census Block Group up.

You may have accessed Census or ACS data directly from the Census Bureau website for other classes or tasks in the past. In the interest of *reproducibility* and efficiency, we introduce the `tidycensus` package in `R`. It is an excellent tool for acquiring either Decennial Census or ACS data directly within `R`. The advantage of doing so is twofold:

1. It can be quicker once you learn how to do it; 2, It makes your data acquisition fully reproducible without any unrecorded steps happening in web browsers. In other words you have actual code that details what you downloaded and what you did to it (rather than un-documented steps of clicking and downloading from a browser).

We will practice the code in the next few sections in lab. It is included here as a primer. In these sections I walk through **one way** to download and prepare data to quantify the county-level prevalence of the population who are uninsured, as this might be a covariate of interest when examining spatial variation in mortality. I selected the code below because it is *relatively* efficient, although you may find some of it complex or confusing. I include it for those who would like to explore other data-manipulation functions in `R`. Please note that you do not need to learn all of the functions in this Census data acquisitions section below for this course, although you might find these or related approaches useful. Note also that there are many ways to accomplish anything in `R`, and you could achieve the same ends with different strategies.

2.3.2.1 Setting up Census API

To access any Census products (e.g. attribute tables or geographic boundary files) using the `tidycensus` package, you need to *register* yourself by declaring your API key. If you haven't already done so, go here to register for the key.

```
# Only do this if you haven't already done it; it should only need to be done once.

tidycensus::census_api_key('YourKeyHere', install = T)
```

2.3.2.2 Choosing Variables

By far the biggest challenge of requesting data from the Census Bureau is knowing *what you want*, and *where it is stored*. Census data are distributed as aggregated counts contained in *specific tables* (each has a unique ID), and made up of *specific variables* (also a unique ID composed of table ID plus a unique ID). There are two ways to find variables:

- You could go to the Census website and browse around. For instance the Census Data Explorer website is one way to browse the topics and variables
- You could download all of the variables for a given year into `R`, and use filters to search it.

This code queries the Census website (assuming you have internet connection) and requests a list of all variables for the ACS 5-year pooled dataset (e.g. `acs5`) and for the window of time ending in 2018 (e.g. 2014-2018). I also specify `cache`

`= T` which just means to save the results for quicker loading if I ask again in the future.

```
library(tidycensus)

all_vars <- load_variables(year = 2018, dataset = 'acs5', cache = T)

head(all_vars)
```

It may be easiest to look at the dataset using the `View()` function. When you do so, you see the three variables, and you have the option to click the **Filter** button (upper left of View pane; looks like a funnel). The *Filter* option is one way to search key words in either the `label` or `concept` column.

We are interested in capturing the prevalence of uninsured in each county. Try this:

- Go to View mode of variables (e.g. `View(all_vars)`)
- Click the *Filter* button
- Type `insurance` in the `concept` field
- Type `B27001` in the `name` field

Figure 2.7: Screenshot of RStudio View() of ACS variables

What we want is a list of the specific tables and variable ID's to extract from the Census. In lab we will use some more detailed code to accomplish this goal.

You may have noticed that the full list of ACS variables has nearly 27,000 variables! In the code below I use some tricks to filter the huge list of all variables to get only the names I want. It relies on the `tidyverse` package