Contents

1 Introduction.

I have invested a lot of time to learn to do something in R only to find myself not remembering what I had done some months later. So I decided to write some personal notes with many examples and as much detail as possible every time I learn a new R library. I thought it would be a good idea to try to polish a bit those notes and make them into some informal manuals. This is not meant to be a formal tutorial, just an extended version of my some notes, in the hope that may be useful to other people.

When I started looking for tutorials for maps, there were lots of resources with examples, but I could not find examples of Mexico. I had no idea where to look for the databases. Even when I found the site, I had no idea which of all the files I found had the information I needed. After finally figuring that out, I thought that was what was most worthy of sharing.

The maps drawn in this tutorial are what's called vector maps. It is important to understand that the information downloaded and loaded to R to generate the maps is not an image in itself. It is more like a database of coordinates of points of the borders of each state and what R will do is graph those points and interpolate straight lines between them, in order to draw the border of the graph. This is explained in several parts of the tutorial, including an introduction of this special kind of object that has a set of coordinates as elements.

The final objective is to to draw a map of all Mexico, divided in states, where each state is filled with a color that tells us the value of a certain variable. Variables can be numeric or categorical. For example, we could have a map where each state is filled with a color indicating the party that governs the state. Or a map where each state is filled with a color indicating a range of values for total population of the state. These maps are called choropleth maps. The tutorials and R code in this repository include instructions on how to download the map of Mexico and basic instructions on how to add data. Basic knowledge of *R* is required and some knowledge of the tidyverse recommended.

In the tutorial, some data is added to the geographic database to be able to plot something on the map. The data is randomly generated and completely meaningless. Some of the examples are not aesthetically pleasing and some are very ugly, but they are made that way on purpose, to exaggerate some features to make the customization options very obvious.

There are 3 R files in this repository.

- mexico.R: R code to download the datasets and create the R map object.
- shapes_intro.R: R code with introduction to the *sf* library (simple features).

- mapamex_eng.R: R code to make the maps with the downladed files. Includes instructions to how to simplify the maps, add data and several examples on how to plot data on the maps.

There is also a file in format called mapasmex_simple.RData, that includes the maps used in the tutorial, ready to be loaded to R without having to be downloaded and processed, and another one called mapasmex_simple_data.RData with the maps with the added fake data. Be careful not to load both files, as the map object with and without the added data has the same name. I'm adding the second file in case the user is unable to add the data or doesn't want to and wants to go directly to plotting the maps.

For this tutorial and the code files, the following libraries are needed:

- dplyr
- ggplot
- tmap
- sf
- rmapshaper
- RColorBrewer

2 Spacial data, simple features and the *sf* library.

Simple features is a set of standards for storage and management of geometric objects used by geographic information systems. Some examples of these geometric objects are points, lines and polygons. In *R*, these objects can be created and managed with the *sf* library. These basic objects belong to a class called *sfg*. These objects can be generated from basic R objects. A point can be generated from a vector, a line can be generated from a matrix where each row is the location of each vertex, and a polygon can be generated by a list. The list to create a polygon has one object, a matrix of vertices, where the first and the last line are the same in order to close the polygon.

```
# create a point from a vector
point_sfg <- st_point(c(5, 2))

# create a line
linestring_matrix = rbind(c(1, 5), c(4, 4), c(4, 1), c(2, 2), c(3, 2))
line_sfg <-st_linestring(linestring_matrix)

# create a polygon
polygon_list = list(rbind(c(1, 5), c(2, 2), c(4, 1), c(4, 4), c(1, 5)))
polygon_sfg <- st_polygon(polygon_list)

point_sfg
```

```
## POINT (5 2)
```

```
line_sfg
```

```
## LINESTRING (1 5, 4 4, 4 1, 2 2, 3 2)
```

```
polygon_sfg
```

```
## POLYGON ((1 5, 2 2, 4 1, 4 4, 1 5))
```

The information to generate the maps we will be using are an object of class *sf* (simple features), which is similar to a *dataframe*, in which one of the columns is made of *sfg* objects that determine the shape of each state. An object of class *sf* can have other columns with data. In fact, an *sf* object is formally comprised of a column of *sfg* objcts (called an *sfc* object, or simple features column) AND a data frame.

To better understand this, let's see a simple example. First, create three polygons (each a *sfg* object). Each of the vectors on the list that creates each polygon correspond to a coordinate on a plane.

```
#Polygon  1
polygon_list1 = list(rbind(c(0,3), c(0,4), c(1,3), c(0,3)))
polygon1 = st_polygon(polygon_list1)

# Polygon 2
polygon_list2 = list(rbind(c(0, 2), c(1, 2), c(1, 3), c(0, 3), c(0, 2)))
polygon2 = st_polygon(polygon_list2)

# Polygon 3
polygon_list3 = list(rbind(c(1,3), c(0,4), c(1,5), c(2,5), c(1,3)))
polygon3 = st_polygon(polygon_list3)
```

These polygons can be grouped (or concatenated) to create an *sfc* object (simple features column). This column has three objects, each one the coordinates of each polygon. This object can now be plotted to show in the same graph all the polygons.

```
#Create a column with 3 polygons (sfc object)

polygon_sfc <- st_sfc(polygon1, polygon2, polygon3)
plot(polygon_sfc)
```

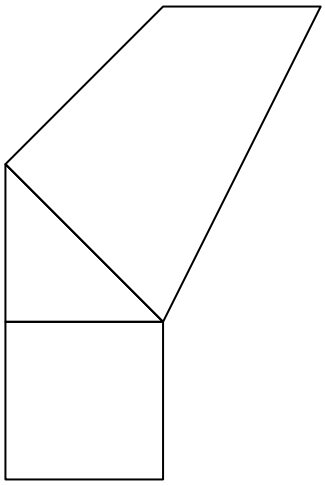


Figure 1: Object polygon_sfc

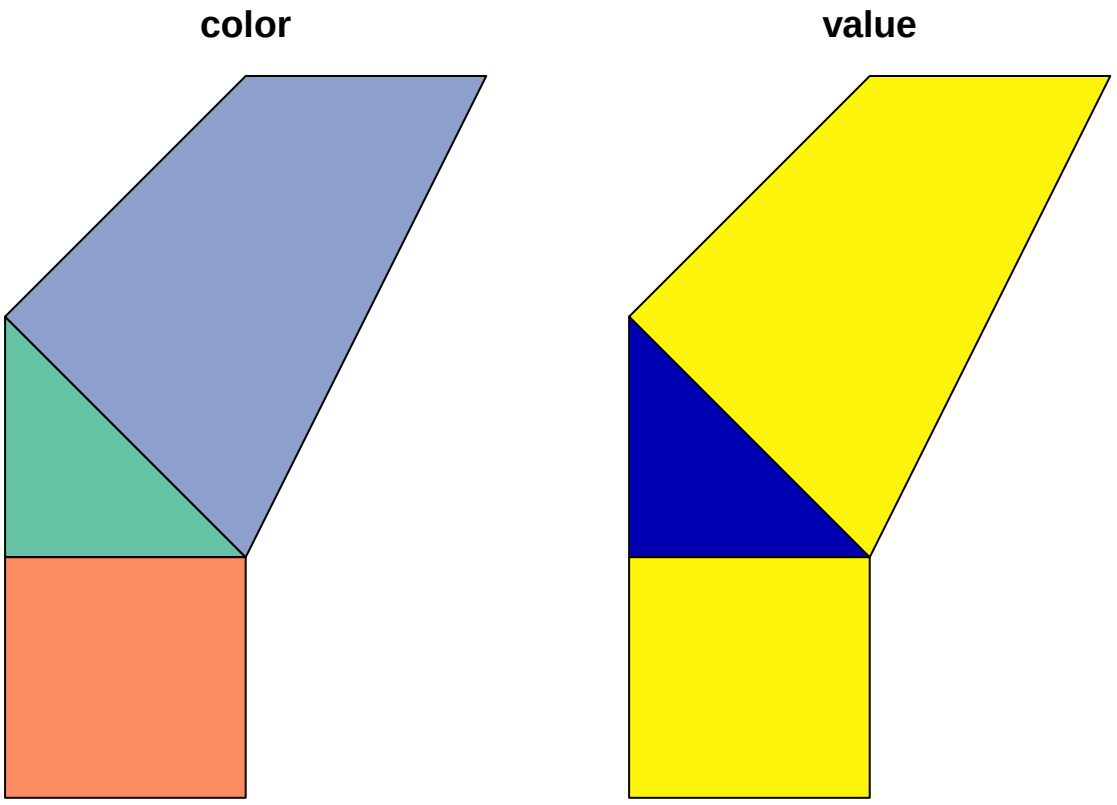
To finally get an *sf* object, we need to add data to the *sfc* object, in the form of a data frame. Here we add some made-up data with two columns corresponding to categorical and numerical data. First a data frame is created (called *poly_attrib* in the example) and then added to the *sfc* with the *st_sf* function.

```
#Create dataframe with made up data
poly_attrib <- data.frame(color= c("blue", "green", "red"), value= c(1,2, 2))

#Use the st-sf function to join the sfc and the data frame  together to get a sf object.
poly_sf <- st_sf(polygon_sfc, poly_attrib)
poly_sf
```

```
## Simple feature collection with 3 features and 2 fields
## Geometry type: POLYGON
## Dimension:     XY
## Bounding box:  xmin: 0 ymin: 2 xmax: 2 ymax: 5
## CRS:           NA
```

```
##   color value                     polygon_sfc
## 1  blue     1 POLYGON ((0 3, 0 4, 1 3, 0 3))
## 2 green     2 POLYGON ((0 2, 1 2, 1 3, 0 ...
## 3   red     2 POLYGON ((1 3, 0 4, 1 5, 2 ...
```

```
plot(poly_sf)
```



To add more data to the *sf* object, it can be treated as if it was a regular *data frame*, and the columns added either with R base functions or *dplyr* functions such as *mutate*.

```
# Create new data
flavor = c("vanilla", "chocolate", "strawberry")

# Add column with dplyr
poly_sf %>% mutate(flavor = flavor)
```

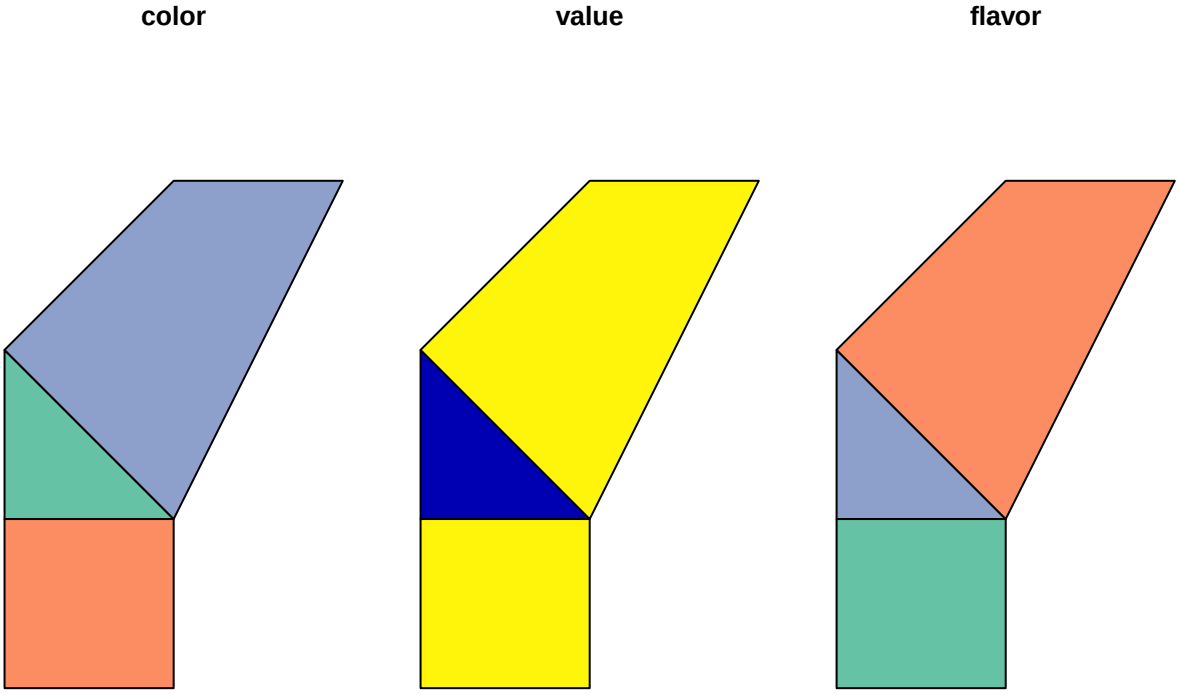
```
## Simple feature collection with 3 features and 3 fields
## Geometry type: POLYGON
## Dimension:     XY
## Bounding box:  xmin: 0 ymin: 2 xmax: 2 ymax: 5
## CRS:           NA
##   color value                     polygon_sfc     flavor
## 1  blue     1 POLYGON ((0 3, 0 4, 1 3, 0 3))    vanilla
## 2 green     2 POLYGON ((0 2, 1 2, 1 3, 0 ...  chocolate
## 3   red     2 POLYGON ((1 3, 0 4, 1 5, 2 ... strawberry
```

```
# Add column with R base

poly_sf$flavor = flavor
```

```
# Other way to add column with R base

#cbind(poly_sf, flavor)
#print header of object with new data
head(poly_sf)
```

```
## Simple feature collection with 3 features and 3 fields
## Geometry type: POLYGON
## Dimension:     XY
## Bounding box:  xmin: 0 ymin: 2 xmax: 2 ymax: 5
## CRS:           NA
##   color value                       polygon_sfc     flavor
## 1  blue     1 POLYGON ((0 3, 0 4, 1 3, 0 3))    vanilla
## 2 green     2 POLYGON ((0 2, 1 2, 1 3, 0 ...  chocolate
## 3   red     2 POLYGON ((1 3, 0 4, 1 5, 2 ... strawberry
```

```
plot(poly_sf)
```



The data downloaded and used in this tutorial are *sf* objects similar to the polygons shown above (but with a lot of vertices), in which each row is a state of Mexico. The column *geometry* is a *sfc* object of "polygons", georeferenced so that, when plotted, each state occupies the correct position with respect of the other states. Georeferenced basically means that the *sfc* object has some metadata so that the "vertices" of each polygon that represents a state are graphed with respect of a common origin. So, to be clear, what we are downloading is not an image of a map, but a dataset with a set of coordinates that *R* will use to draw a map.

To learn more about simple features ir R, see link

3 Download the maps.

The maps used in this tutorial are files of *shp* format (shapefile), downloaded from INEGI page inegi.org.mx. For this tutorial we will be using the simplest map, the one that has only a column with the name of the state and a column with the geographic data (the *sfc*), so each row is made of the name of the state and a *sfg*

object, the coordinates of a polygon representing the shape of that state. This map is located in this webpage.

The file *mexico.R* includes all the instructions needed to download the *shp* file, load it to R (creating an *sf* object) and erase the zip file and the directory. These instructions will read the file called *00ent.shp*, which has the data at state level, which is used in this tutorial. The files *00ent.shp*, *00ent.shx*, *00ent.dbf*, *00ent.prj* y *00ent.shx* also need to be in the directory from which the files are being read. The downloaded directory has other files with geographical data at a different scale. Though those are not explored in this tutorial, the technique is the same. If the user intends to create maps at a smaller scale, it is recommended not to erase the downloaded directory, by adding a # before the instruction *unlink(temp)*. Alternatively, the compressed file can be downloaded directly from the website and manually unzipped and read with the function *st_read*.

```
##Read map at state level
mapamex <- st_read("00ent.shp")
```

If the code ran correctly, there should be an object called *mapamex* in the R environment.

If the user can't download the map from INEGI or wishes not to do so, the object *mapamex* is included in the repository in the format RData. The name of the file is *mapamex.RData* and can be loaded with the *load* function (be sure to use the correct path):

```
load(mapamex.RData)
```

4 Graphing the maps.

In this section we'll explain how to simplify the map, add other data and plot them using libraries *tmap* and *ggplot*. All the code to do this is included in the file *mapamex.R*

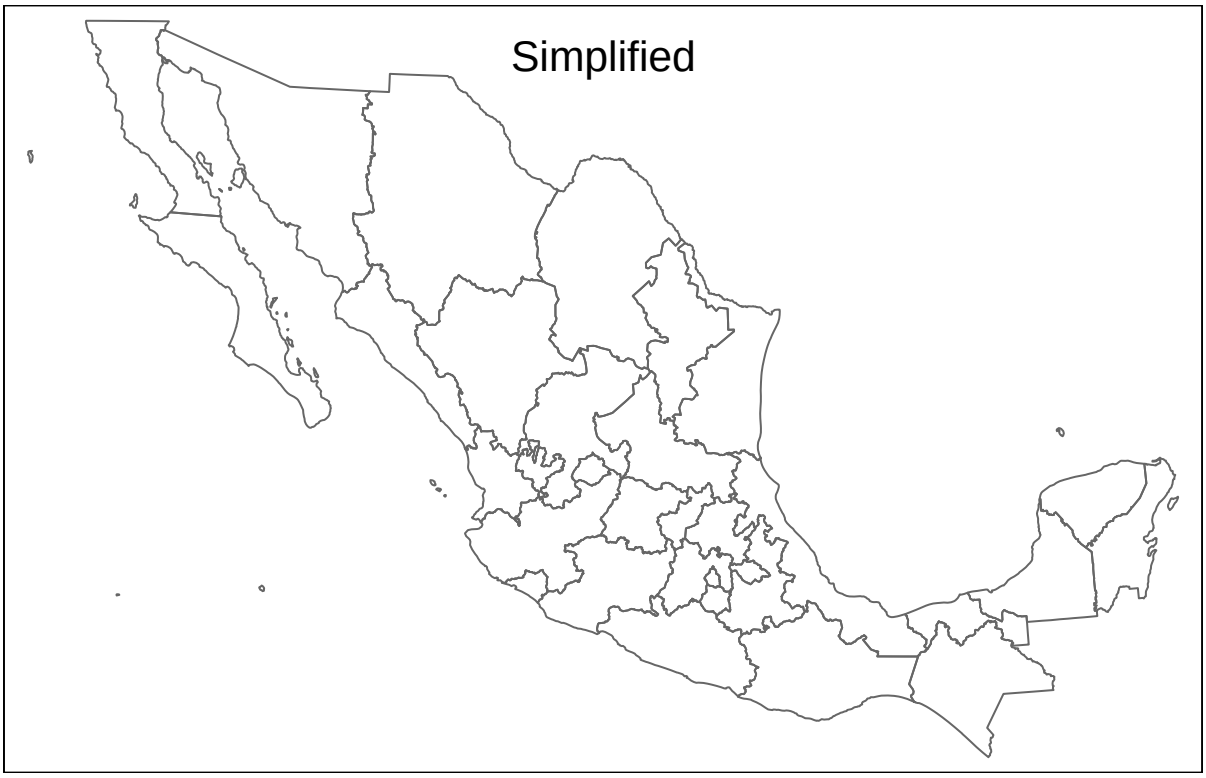
4.1 Simplify the map.

As it was explained in section 2, each state is plotted from a set of coordinates in space. The actual graph is made of the polygon that is obtained from straight lines going from one vertex to the next one, which in the end "draws" the borders of the state. Each state has an enormous amount of coordinates to make it possible to plot a map with a lot of detail or in a large format. For many uses, that much detail is not necessary. We can obtain a map made of polygons with fewer point that essentially looks the same for our plotting purposes. Reducing the number of polygon vertices, or simplifying the objects, has two advantages. The plots are rendered more quickly (meaning that the maps are drawn more quickly) and the objects are smaller (occupy less memory). We can do that with the function *st_simplify*. The parameter dTolerance determines the amount of simplification. The larger this parameter is, the more simplified the object gets.

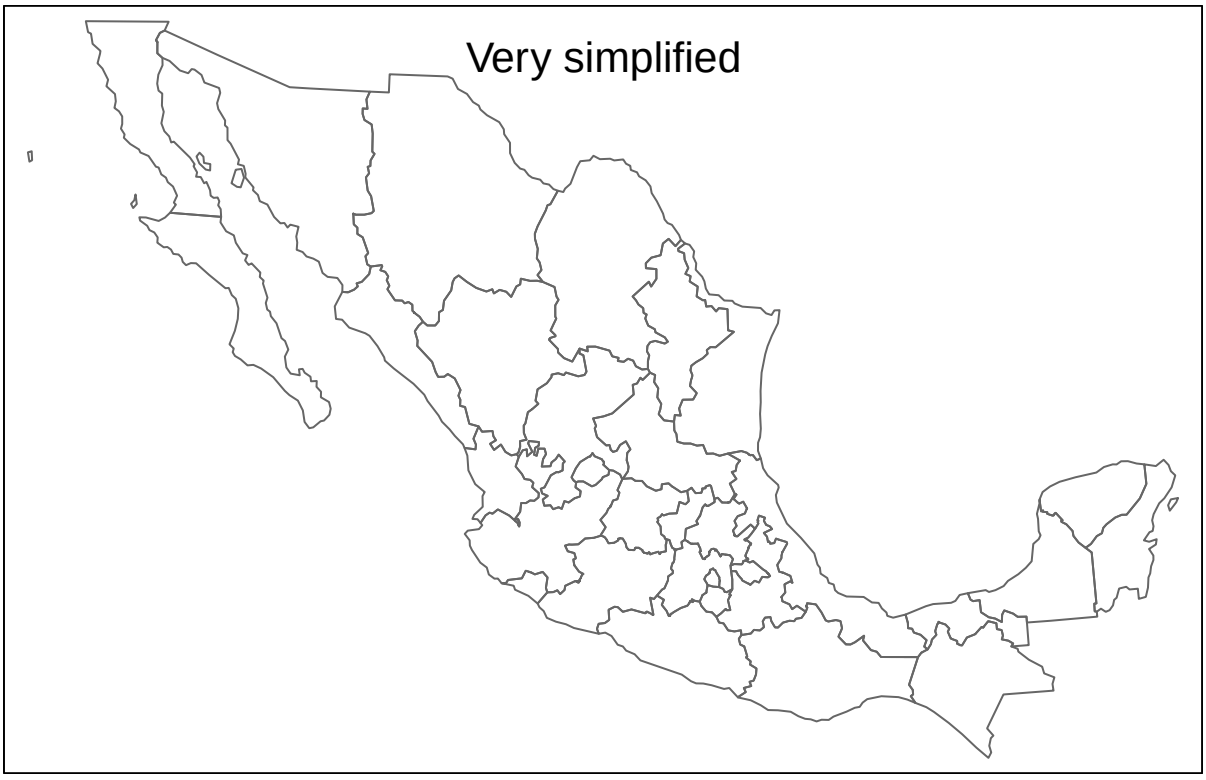
```
#Simplify map
mex_simple <- st_simplify(mapamex, dTolerance=2000,preserveTopology = FALSE )

mex_simple %>%tm_shape() +
    tm_borders()
```

The problem with the function *st_simplify* is that it simplifies the elements of the *sfc* column individually, so after the simplifications, the borders between the states may no longer match. The function *ms_simplify* from the library *rmshaper* solves this, as it simplifies the whole column and makes the borders the same. The parameter *keep* of the funcion *ms_simplify* determines the proportion of vertices that will be kept. This parameter will depend on the scale we want to plot. In the following example, we can see that even keeping just 1% of the vertices it is possible to plot the shape of the country and the states. Depending on what we want to use it for, sometimes a very simplified, even "cartoonish" drawing of the country is desirable and good enough to convey the idea we want to communicate.

```
#simplified map
mapamex_simple <-  ms_simplify(mapamex, keep = 0.01, keep_shapes = TRUE)

#very simplified map
mapamex_muysimple <- ms_simplify(mapamex, keep = 0.002, keep_shapes = TRUE)



mapamex_simple %>%tm_shape() +
                  tm_borders() +
                  tm_layout(title = "Simplified",
                            title.position = c('center', 'top'))
```




```
mapamex_muysimple %>% tm_shape() +
    tm_borders() +
    tm_layout(title = "Very simplified",
              title.position = c('center', 'top'))
```



4.2 Add data to the object *sf*.

In this section we will use the simplified version of the map generated in the previous section, the object named "mapamex_simple". This object can also be found in the repository as a file called *mapamex_simple.RData*, which can be loaded directly to R. Data can be added to this *sf* object in exactly the same way as it is added to any data frame. It can be done with base R or with *dplyr*. To show how it is done, first let's generate some fake data. We will add one categorical data column and two numerical columns. For the categorical data, we sample 32 times with repetition from three options (cat, dog and rabbit) and the two numericals are just samples from distributions. So first we get three vectors called *categorica*, *numerica* and *numerica2*. These objects are called vectors in R, but will be added as *columns* to the data frame.

```
set.seed(100)
categorica <- sample(c("cat", "dog", "rabbit"), 32, replace = T)
numerica <- round(runif(32,30,80),2)
numerica2 <- round(rnorm(32, 40, 35))^2

categorica
```

```
##  [1] "dog"    "rabbit" "dog"    "rabbit" "cat"    "dog"    "dog"    "rabbit"
##  [9] "dog"    "dog"    "rabbit" "dog"    "dog"    "rabbit" "rabbit" "rabbit"
## [17] "rabbit" "dog"    "cat"    "rabbit" "rabbit" "dog"    "cat"    "dog"
## [25] "rabbit" "dog"    "cat"    "rabbit" "rabbit" "cat"    "rabbit" "dog"
```

```
numerica
```

```
##  [1] 69.02 74.21 40.39 45.35 46.53 39.93 41.78 43.74 59.57 42.67 36.17 41.50
## [13] 59.88 40.57 53.19 62.36 78.03 63.82 52.26 47.89 52.79 52.27 42.25 64.72
## [25] 50.61 46.39 58.63 78.35 63.09 61.23 72.83 68.74
```

```
numerica2
```

```
##  [1]  5476  1296  7921   484  3844   484  7396   729  7396  1764   676   576
## [13]   441  2116 11236  1600  5476    81 10816  7744   121   961  1444   729
## [25] 16900  2025   225  3844  2209  1444  1369  3136
```

The most straightforward way to add this data to the *sf* object is with R base:

```
mapamex_simple$categorica  <- categorica
mapamex_simple$numerica <- numerica
```

Data can also be added with *dplyr*, using mutate.

```
mapamex_simple <-
    mutate(mapamex_simple, categorica = categorica, numerica = numerica, numerica2 = numerica2)
```

There are multiple other ways to add data to the map, like with *rbind* or even merging the *sf* object with a data frame. How to accomplish this is beyond the scope of this tutorial.

Here's the header of the *sf* object, after adding the data:

```
head(mapamex_simple)
```

```
## Simple feature collection with 6 features and 6 fields
## Geometry type: MULTIPOLYGON
## Dimension:     XY
## Bounding box:  xmin: 911292 ymin: 692158.7 xmax: 3859532 ymax: 2349605
## Projected CRS: MEXICO_ITRF_2008_LCC
##   CVEGEO CVE_ENT              NOMGEO                       geometry categorica
## 1     01      01       Aguascalientes MULTIPOLYGON (((2492248 114...        dog
## 2     02      02      Baja California MULTIPOLYGON (((1208688 187...     rabbit
```

```
## 3     03      03  Baja California Sur MULTIPOLYGON (((1711892 136...        dog
## 4     04      04             Campeche MULTIPOLYGON (((3709416 998...     rabbit
## 5     05      05 Coahuila de Zaragoza MULTIPOLYGON (((2712006 174...        cat
## 6     06      06               Colima MULTIPOLYGON (((1157633 768...        dog
##   numerica numerica2
## 1    69.02      5476
## 2    74.21      1296
## 3    40.39      7921
## 4    45.35       484
## 5    46.53      3844
## 6    39.93       484
```

4.3 Creating maps.

We finally get to the section where we draw maps. It is a simple introduction, using our made up data, but it will serve as an introduction to make maps with more serious or useful content. The maps will be graphed with the library *tmap*. The syntax of *tmap* is very similar to *ggplot*, so previous knowledge of the latter will be useful. For the examples we will be using the *sf* object created in previous section.

4.3.1 Draw the borders.

The simplest map that can be created is one that only has the borders of the states. The library *tmap* works with functions that are added sequentially to the map with a *+*. Each of these functions added with *+* is a graphical feature that is added on top of the previous one, and these features are called layers. The function *tm_shape* is used to determine which *sf* object will be plotted. The first layer added is a plot of the polygons, representing the borders of the states, using the function *tm_borders*.

```
tm_shape(mex_simple) +
    tm_borders()
```

4.3.2 Map categorical data.

Now we can start adding more information to the map. The first layer we will add is one that plots other variables of the *sf* object on top of the map. One way to do this is by filling the interior of each state with a color that indicates either a category or a range of values, which is done with the function *tm_fill()*. In this first example, the map will be filled with our categorical data. So the parameters passed to the function are the column used and the type of data. In this case, the column name is *categorica* and the type of data is *cat*. In the next example we show how these two parameters are passed to the *tm_fill* .

```
tm_shape(mapamex_simple) +
    tm_borders() +
    tm_fill(col ="categorica", style = "cat")
```

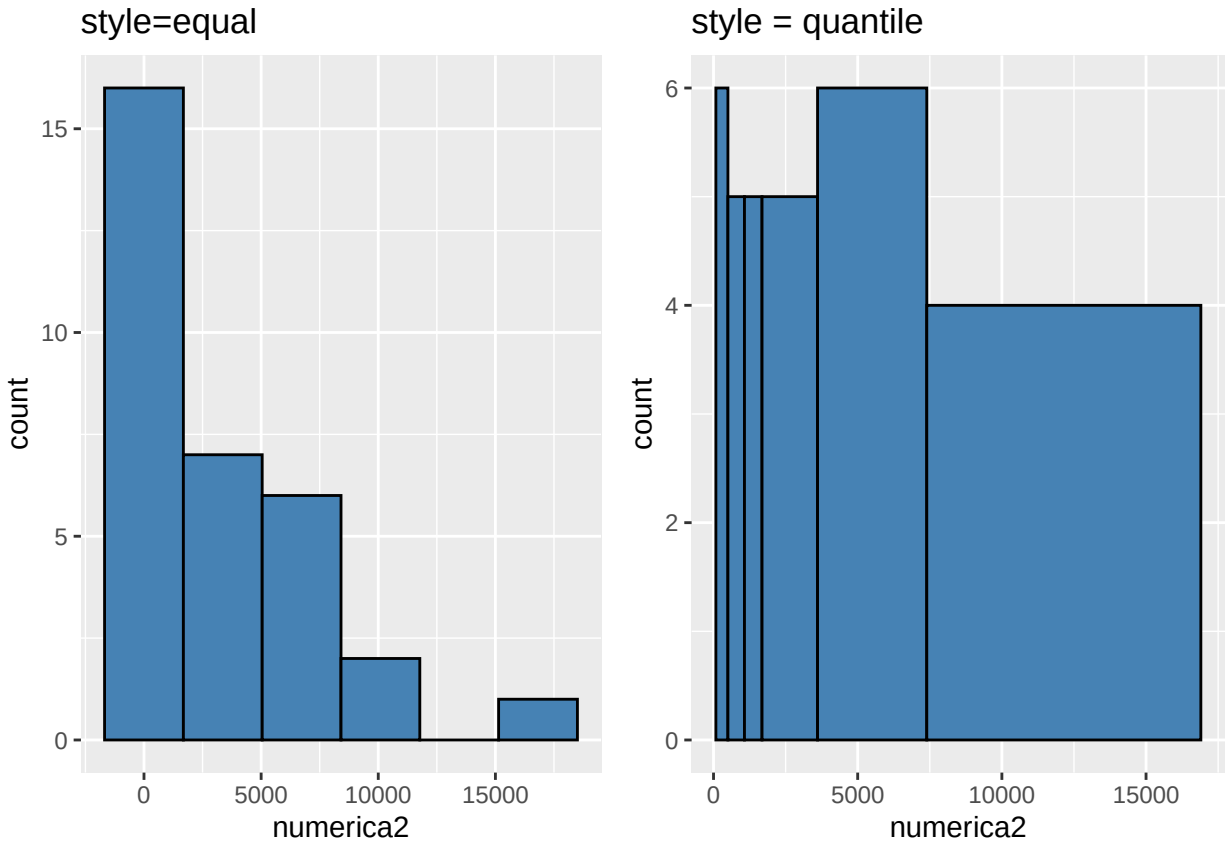
One can also use the function *tm_polygons*, which is equivalent to *tm_borders + tm_fill*.

```
tm_shape(mapamex_simple) +
    tm_fill(col ="categorica", style = "cat")
```

4.3.3 Map numerical data.

To graph a numerical variable on a map, the numerical data is made discreet by dividing it into bins. These bins can be defined by dividing the range into intervals of either equal length or with the same amount of data. The variable is usually divided in several intervals and interval given a color in a continuous color scale. There are two parameters used in *tm_fill* to achieve this. The parameter *n* determines in how many intervals the range will be divided and, the parameter *style* determines how this division will me made. To divide the range in intervals of the same length we use *style= "equal"* and intervals containing the same amount of data *style = "quantile"*. If the data is not very skewed, both methods will yield very similar intervals. If the data is very skewed, the intervals will have very different lengths but will have a better distribution of colors.

To better understand the difference between the two ways of dividing the range, we can plot histograms of the same variable, dividing the data into 6 bins in the two ways previously described:

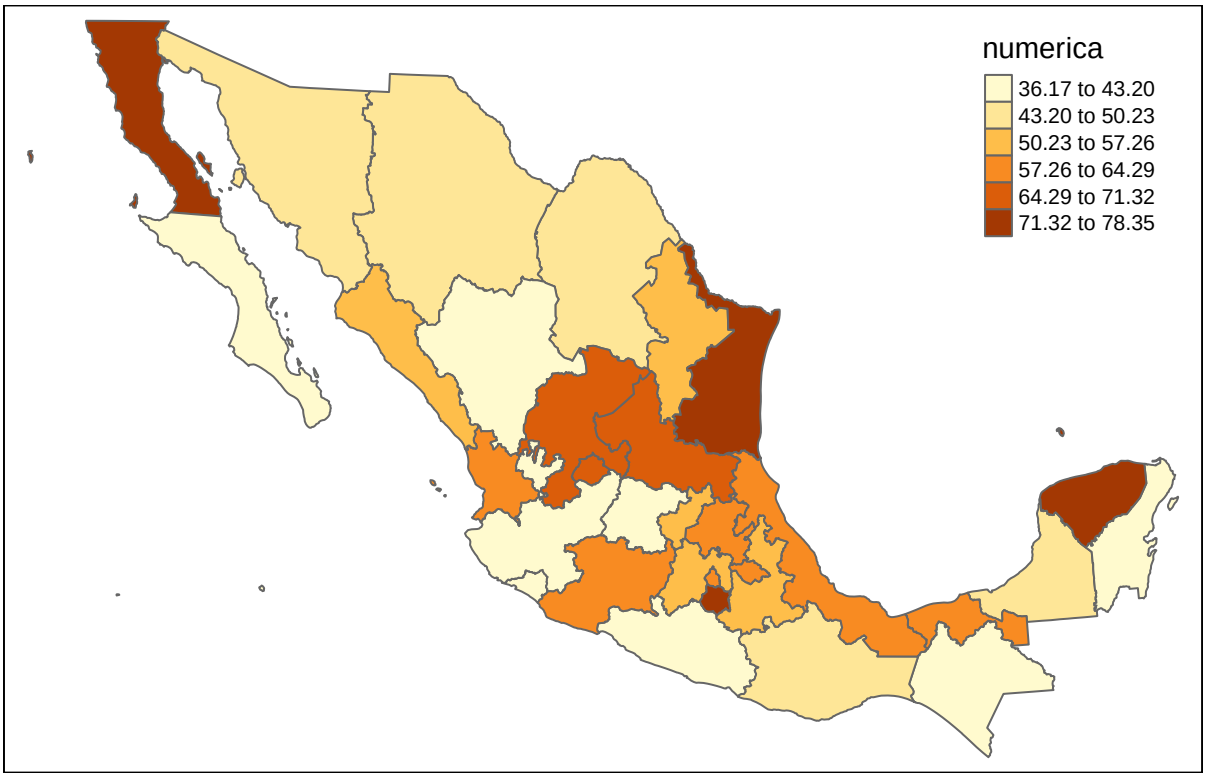


Although the values are made discrete, the colors used for numerical data are tones of the same color in order to communicate an increase or decrease of a value. This is different to plotting a variable that is intrinsically discrete, in which case we want very different colors to create contrast.

Here are some examples of maps with both options (equal and quantile):

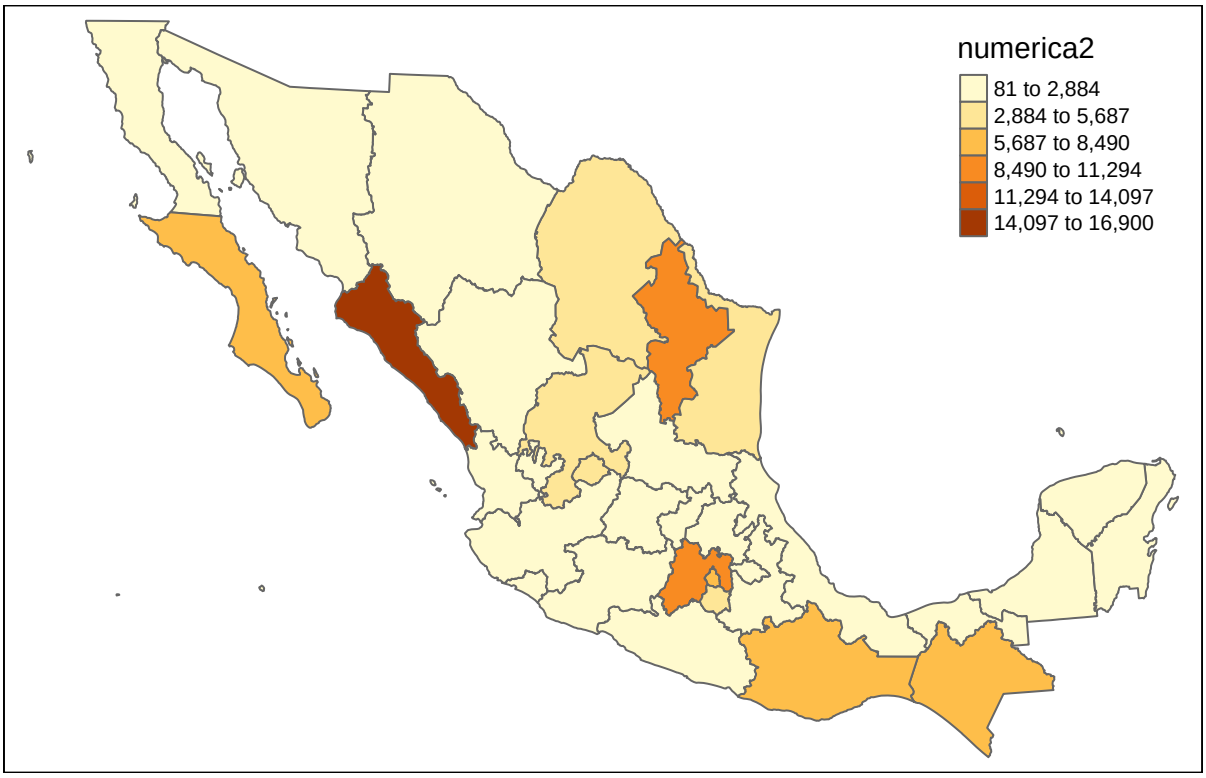
On the next map the variable called *numerica* is divided with *style = "equal"*. It works well for this one because the data is uniformly distributed

```
# Variable "numerica" with range divided into 6 equal intervals

tm_shape(mapamex_simple) +
    tm_borders() +
    tm_fill(col ="numerica", n=6, style = "equal")
```



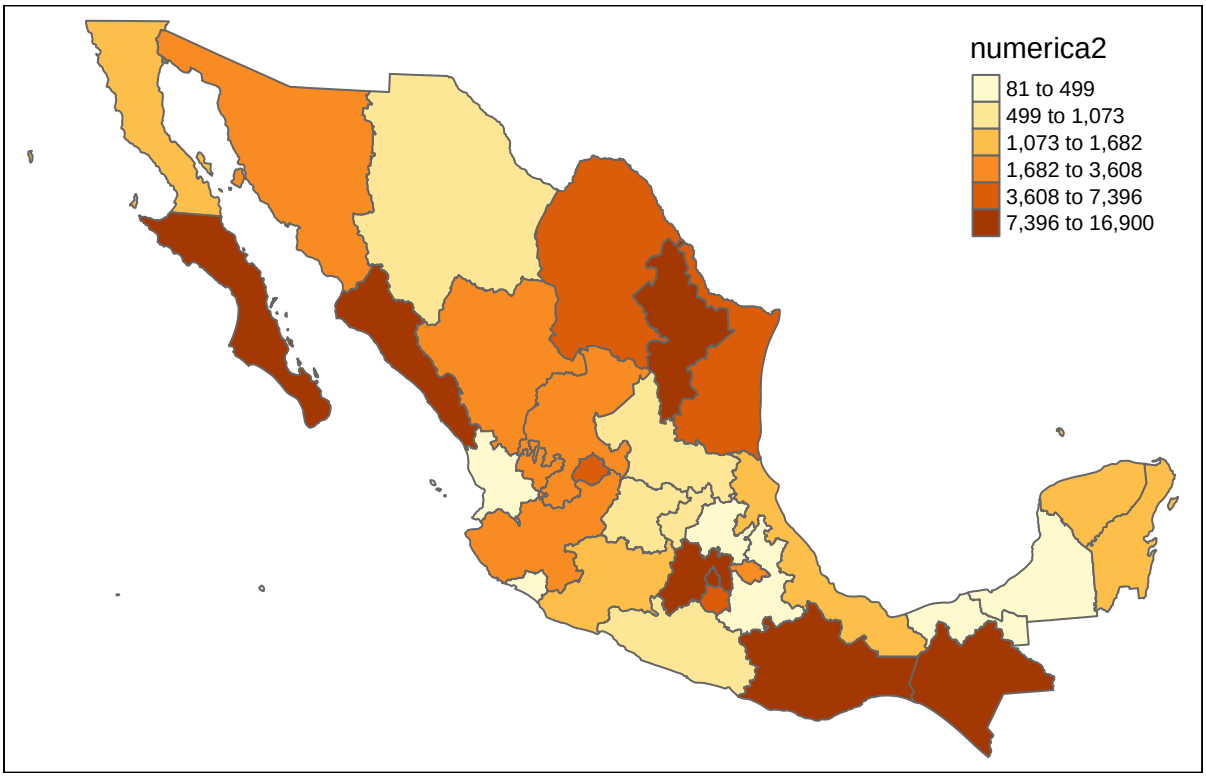
Now let's graph on the map the variable *numerica2* also using equal spacing. The data in this variable is very skewed so the map doesn't show much variation.

```
tm_shape(mapamex_simple) +
    tm_borders() +
    tm_fill(col ="numerica2", n=6, style = "equal")
```



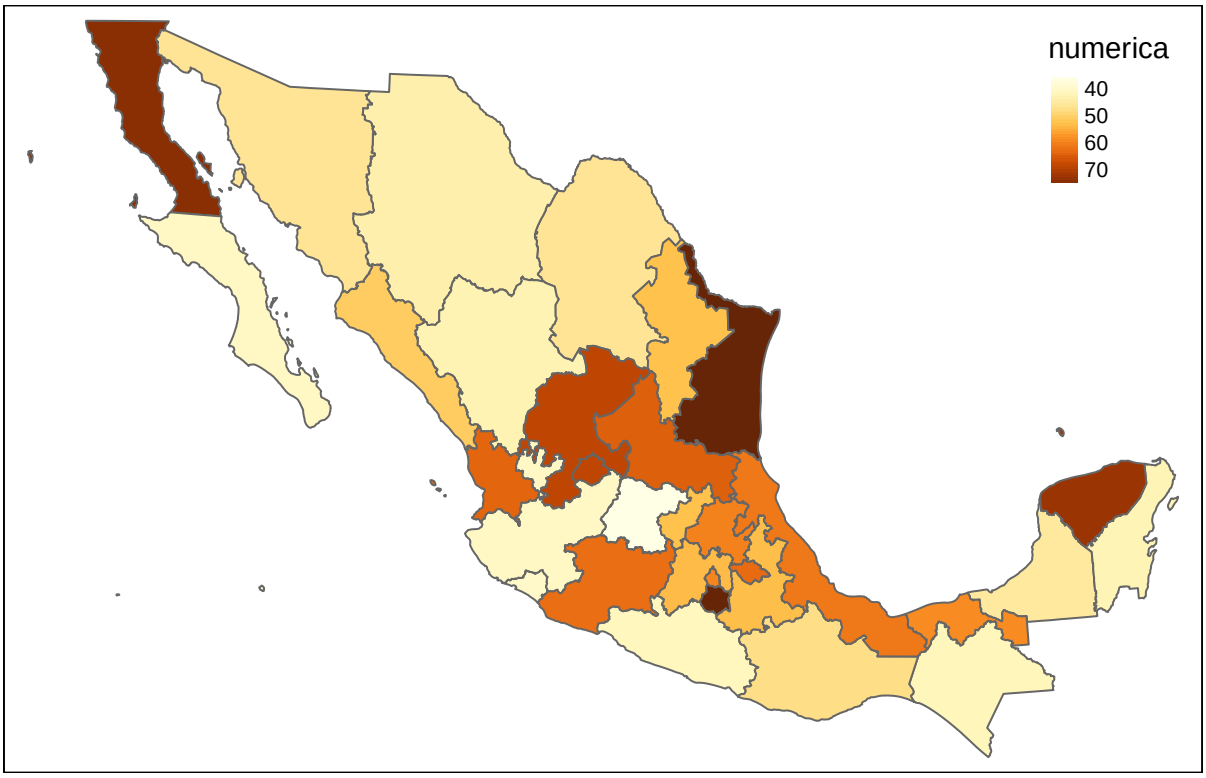
If we repeat the map but using the option *quantile* instead of *equal*, we get better results.

```
#Numerical variable with range divided in 6 intervals with the same number of observations

tm_shape(mapamex_simple) +
    tm_borders() +
    tm_fill(col ="numerica2", n=6, style = "quantile")
```



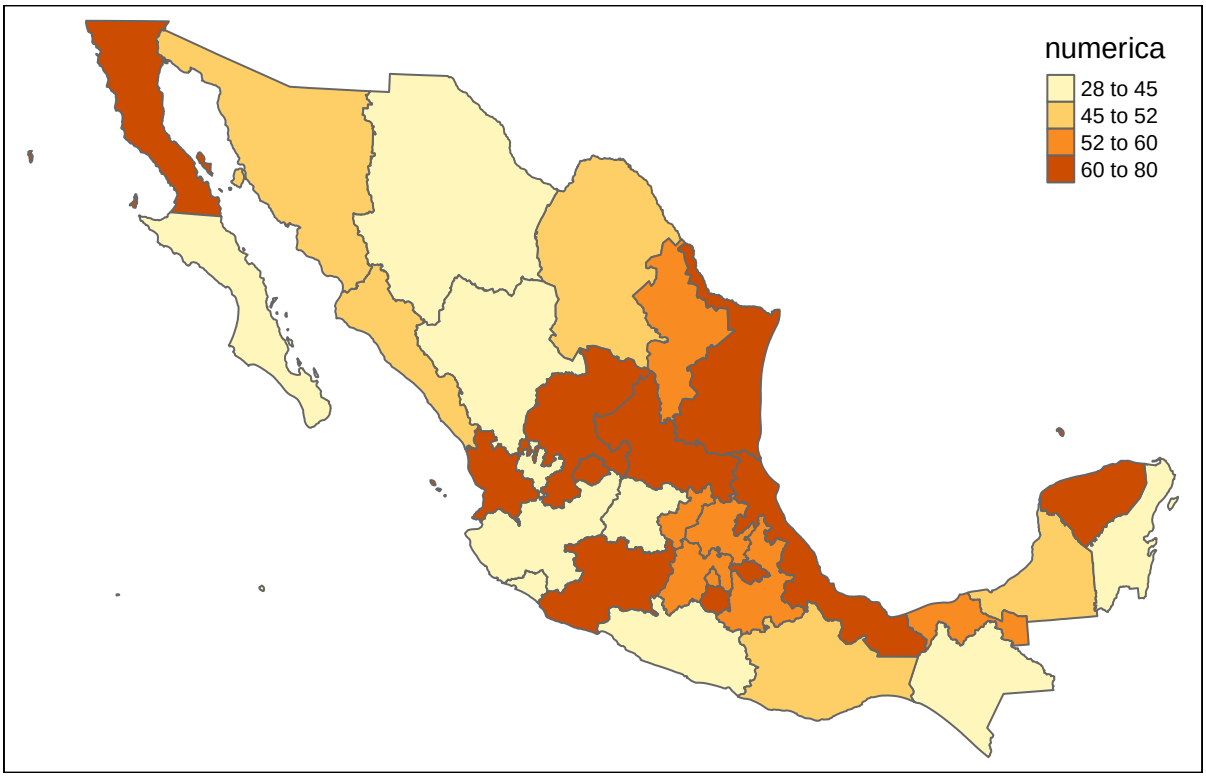
Another way to plot a numerical variable is with the option *cont* (continuous). The variable value is indicated in a tone from a continuous gradient of tones of a color.

```
#Numerical variable with range plotted on a continuous gradient of tones
 tm_shape(mapamex_simple) +
    tm_borders() +
    tm_fill(col ="numerica",  style = "cont")
```



The range can also be divided manually, by specifying first the position of the breaks:

```
#Numerical variable with range specified manually
my_breaks = c(28, 45, 52, 60, 80)

tm_shape(mapamex_simple) +
    tm_borders() +
    tm_fill(col ="numerica",  breaks = my_breaks)
```



For more options, see https://geocompr.robinlovelace.net/adv-map.html

4.4 Choosing colors.

In the previous section, the maps were colored with the default options of the library *tmap*. The colors can be also chosen by the user. They can be chosen either manually or using professionally made palettes.

4.4.1 Choosing colors manually.

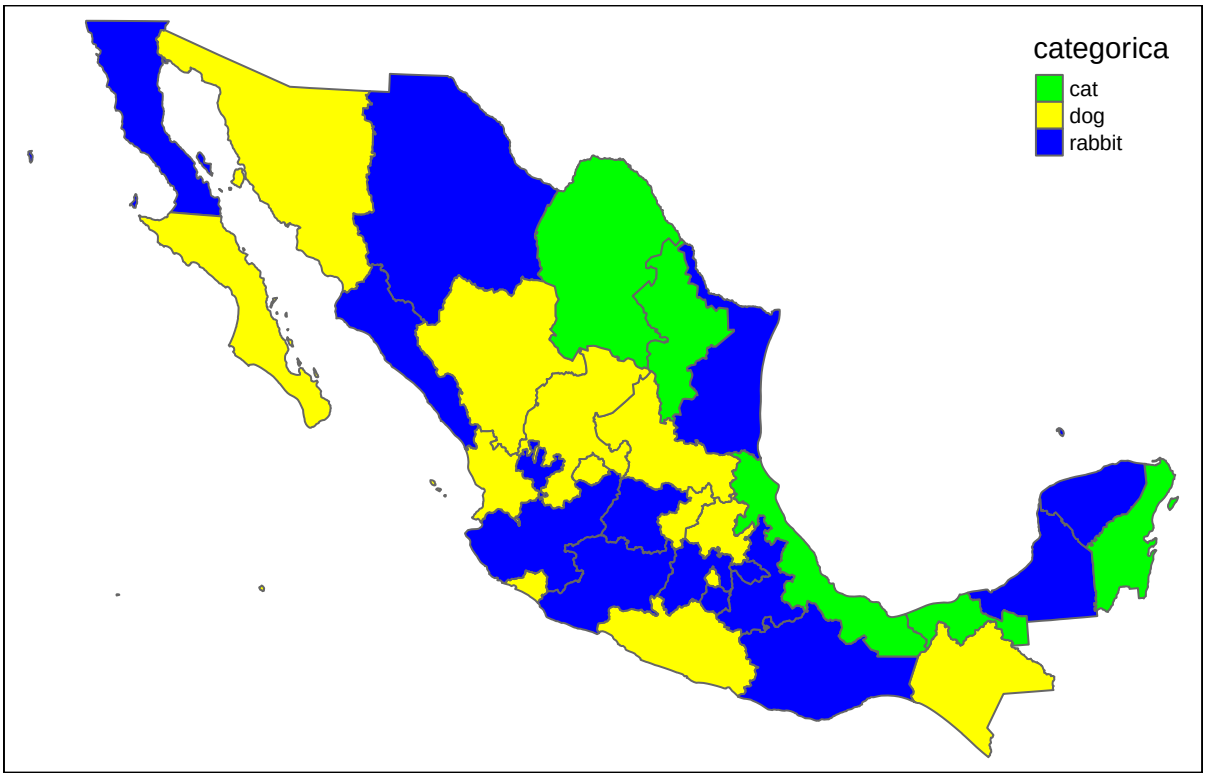
The easiest way to choose colors for the maps is manually creating a vector where the elements are the names of the colors or their hexadecimal, and passing it as an option in the function *tm_fill*. The names of the colors can be found in this cheat sheet along with other information that will be useful in the next section. The hexadecimal code for colors can be found in this website.

In the following example, the map with the variable *categorica* is drawn again but now using a palette of colors chosen manually by the user, using the color names:

```
mypal <- c("plum2", "springgreen3", "steelblue2")

tm_shape(mapamex_simple) +
    tm_borders() +
    tm_fill(col ="categorica", style = "cat", pal = mypal)
```

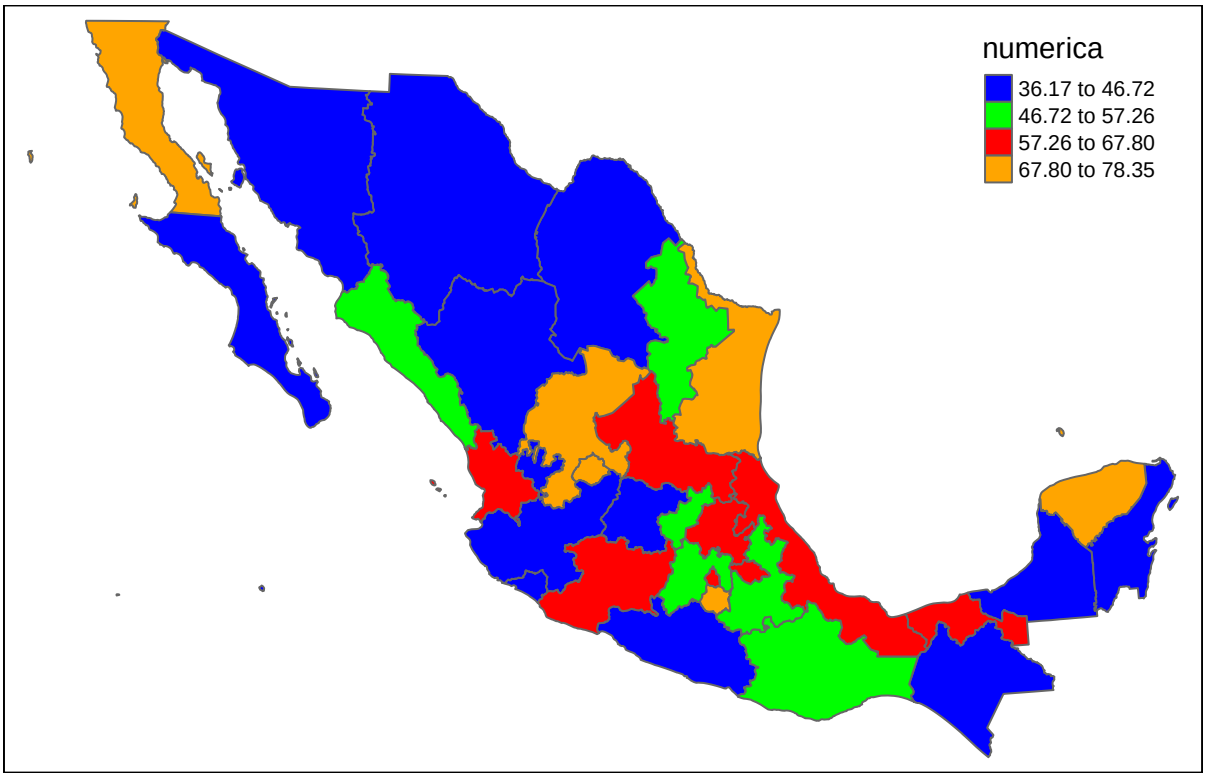
It can be customized further by choosing the color for each category of a categorical variable:

```
mypal <- c("rabbit" = "blue", "cat" = "green", "dog"= "yellow")

tm_shape(mapamex_simple) +
    tm_borders() +
    tm_fill(col ="categorica", style = "cat", pal = mypal)
```



The same can be done with numerical variables. A vector is created with the number of colors equal to the number of intervals that be used to divide the range. The particular colors chosen for this example are not ideal for numerical values, but they are just a quick way to exemplify how to choose the colors for this type of data. (It is generally recommended that for a numerical variable, a sequential palette is used and in this example we are using contrasting colors, just to show how it is done). We'll learn more about palettes in the next section.

```
pal <-  c("blue", "green", "red", "orange")

tm_shape(mapamex_simple) +
    tm_borders() +
    tm_fill(col ="numerica", n=4, style = "equal", pal=pal)
```

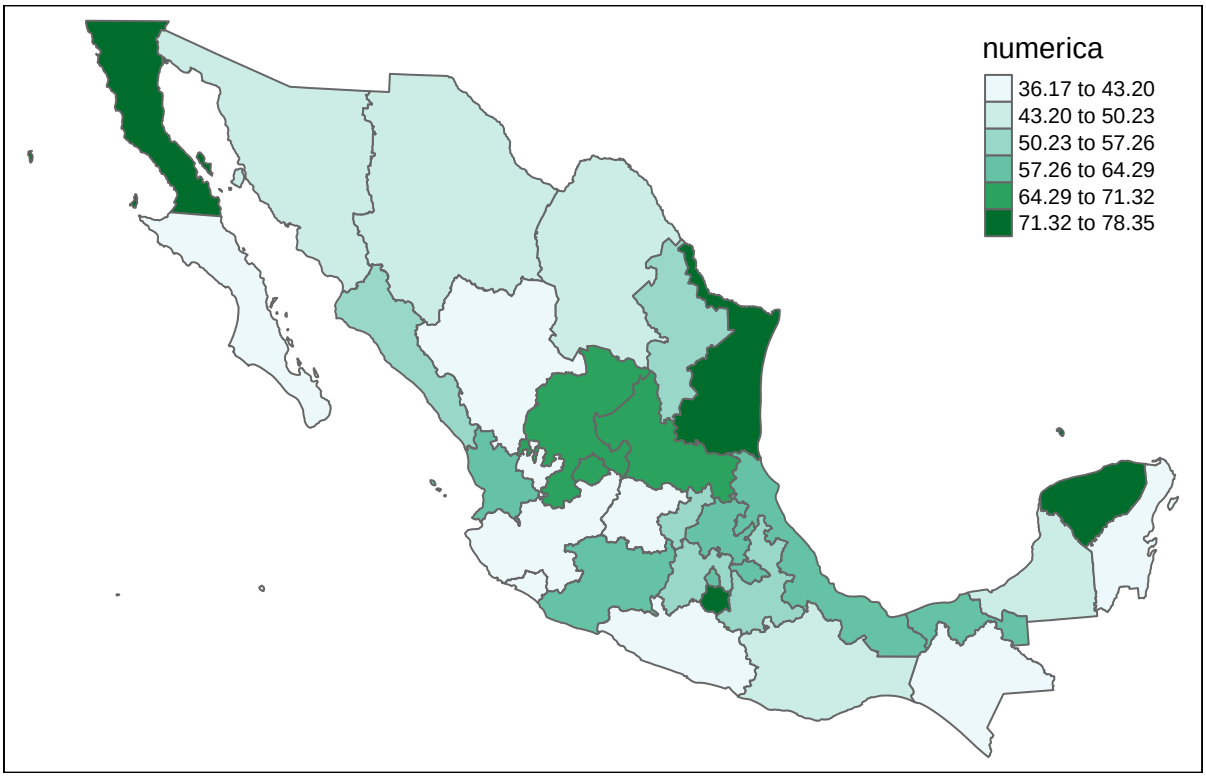


4.4.2 Specify colors with Color Brewer.

Even though it is possible to specify the colors manually, in most cases it is better to choose the colors from a professionally mad palette. There are several sites with palettes made specifically to better communicate every kind of information, which can be used for graphs and maps. The library *RColorBrewer* includes the palettes Color Brewer, prepared specifically for cartography. To see examples of Color Brewer palettes used n maps, see link.

For numerical variables it is recommended to use sequential palettes. These contain sequences of tones of the same color or several color, that progressively go from a lighter to a darker gone, giving the person immediate information about a value being higher or lower than others, without even having to see the legend. In the following example we will use a sequential palette in tones of green. The palette is defined with the function *brewer.pal* with two parameters, the number of tones we need and the name of the palette.

```
# Choose the palette from color Brewer
pal <- brewer.pal(6,"BuGn")

tm_shape(mapamex_simple) +
    tm_borders() +
    tm_fill(col ="numerica",  n=6, style = "equal", palette = pal)
```



For a categorical variable, we use a palette with contrasting colors:

```
pal <- brewer.pal(3, "Pastel1")

tm_shape(mapamex_simple) +
    tm_borders() +
    tm_fill("categorica", palette = pal,   style ="cat")
```

To learn more about color palettes, see link

4.5 Titles and legends.

Titles and legends can be customized in a lot of ways. In the following sections we will give a brief introduction on how to add titles and customize them, along with the legends, by changing font size, font size, colors and their location within the map.

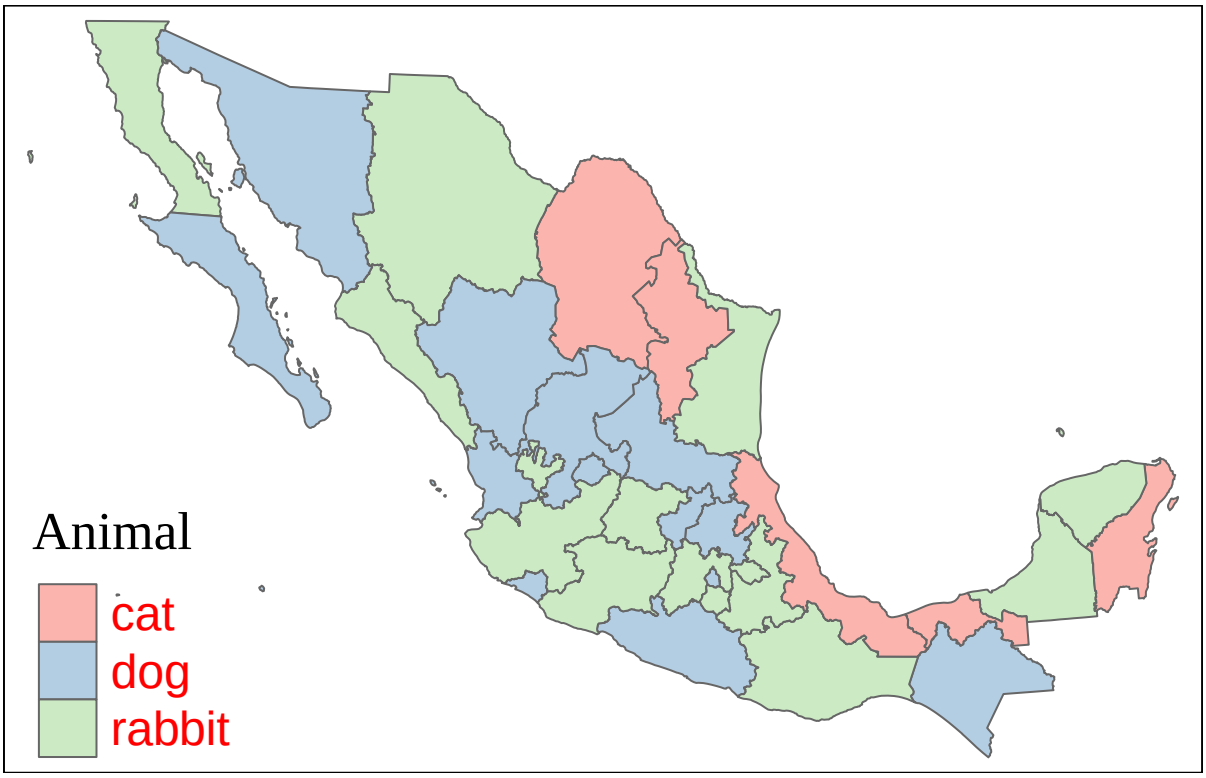
When we draw a map the way it was shown in previous sections, by filling each state with a color indicating the value of the variable, a legend is automatically generated. This legend tells us to what value or category of the variable each color corresponds. The title of this legend is the name of the variable. In many cases the names of the variables are meaningless to our intended audience, so they can be changed to communicate more clearly what we want. For example, in our fake data, the name of the variable is *categorica*, but the values the variable takes are names of animals. The following example shows how to change the title of the legend. To change the title of the legend, we need to add an option. In this case we want the title to be "Animal", so we add the option *title = "Animal"*.

Other elements from the legend that can be customized are the position of the legend and the sizes, types and colors of the fonts. This is done by adding a new layer with the function *tm_legend*. Some options for this functions are:

- The position of the legend. Using a vector with two components with values between 0 and 1, where *(0,0)* is the lower left corner, and *(1,1)* the upper right corner.
- *title.size* Size of of the font for the title text.
- *text.size* Size of the font for the text (the values or the categories that each color represents)
- *text.color* and *title.color* Colors for the letters of the values of the categories and the title of the legend.

```
pal <- brewer.pal(3, "Pastel1")

tm_shape(mapamex_simple) +
    tm_borders() +
```

```
    #fill states with the variable "categorica" and change title of legend
    tm_fill("categorica", palette = pal,   style ="cat", title = "Animal") +
    #Customize legend position, text size, color and font
    tm_legend(position = c("left", 'BOTTOM'),
              title.size = 2,
              text.size = 1.5,
              text.color = "red",
              title.fontfamily = "serif")
```

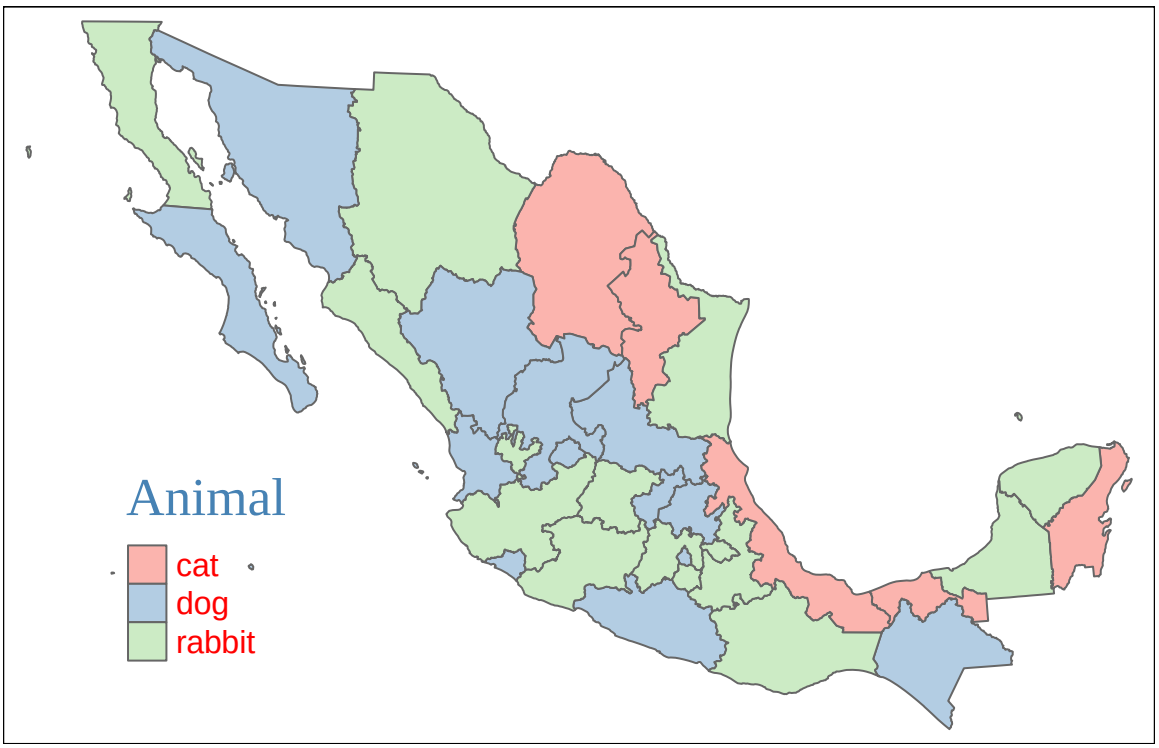


To add and customize title, we use the function *tm_layout*. We can pass through this function not only the text of the title, but the position in the map panel, the size of the text, the color of the text and many more options.

The following example shows a map with several customized elements in the title.. The resulting map is very ugly, but it has been exaggerated to make it very obvious which elements are being tweaked. The aspects of the title that have been customized are the name of the title, the size of the fonts, the color of the fonts and the title position (horizontally, it can be centered or justified to either side).

```
tm_shape(mapamex_simple) +
    tm_borders() +
    tm_fill("categorica", palette = pal,   style ="cat", title = "Animal") +
    tm_legend(position = c(0.1,0.1),
              title.color = "steelblue",
              title.size = 2,
              text.size = 1,
              text.color = "red",
              title.fontfamily = "serif") +
    tm_layout(main.title = "Pets in México",
              main.title.size = 2,
              main.title.color = "green",
```

```
            main.title.position = "center")
```



The title can also be placed inside the map panel, using the option "title" instead of "main.title". The options for customizing font color, size and font family are the same as with the main title. The difference with this title is that it can be placed anywhere inside the panel, so just like with the legends, we need to specify both the horizontal and vertical position. This can be done with a vector of two values between 0 and 1, or with a vector with either "left", "center" or "right" as the first value and "top", "center" or "bottom" as the second value:

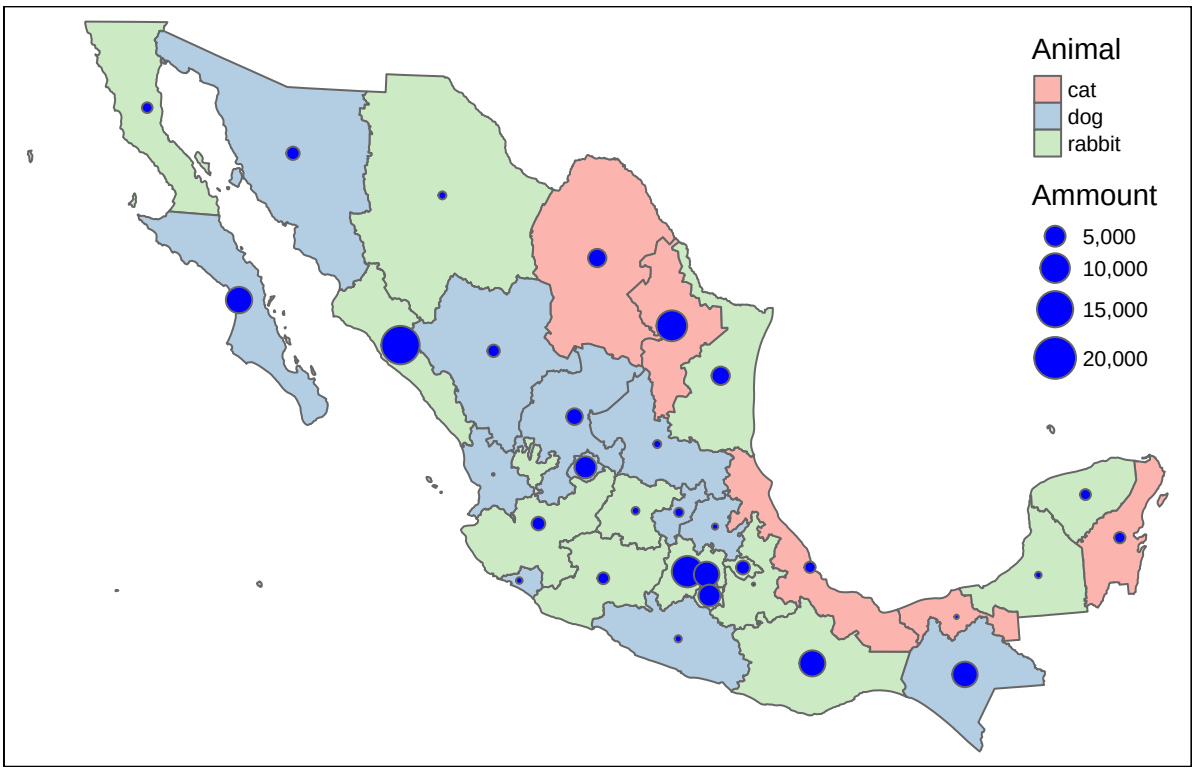
```
tm_shape(mapamex_simple) +
    tm_borders() +
    tm_fill("categorica", palette = pal,   style ="cat", title = "Animal") +
    tm_legend(position = c(0.1,0.1),
              title.color = "steelblue",
              title.size = 2,
              text.size = 1,
              text.color = "red",
              title.fontfamily = "serif") +
    tm_layout(title = "Pets in México",
              title.size = 2,
              title.color = "black",
              title.position = c("center", "top")
    )
```

4.6 Plotting variables with other graphical elements.

Besides filling out each polygon with colors indicating the value or category of the variable, there are other ways to plot a variable on the map. For example, we can add symbols with a different shape or color for each category or circles where the size is proportional to the value of the variable. This way we can plot several variables on the same map.

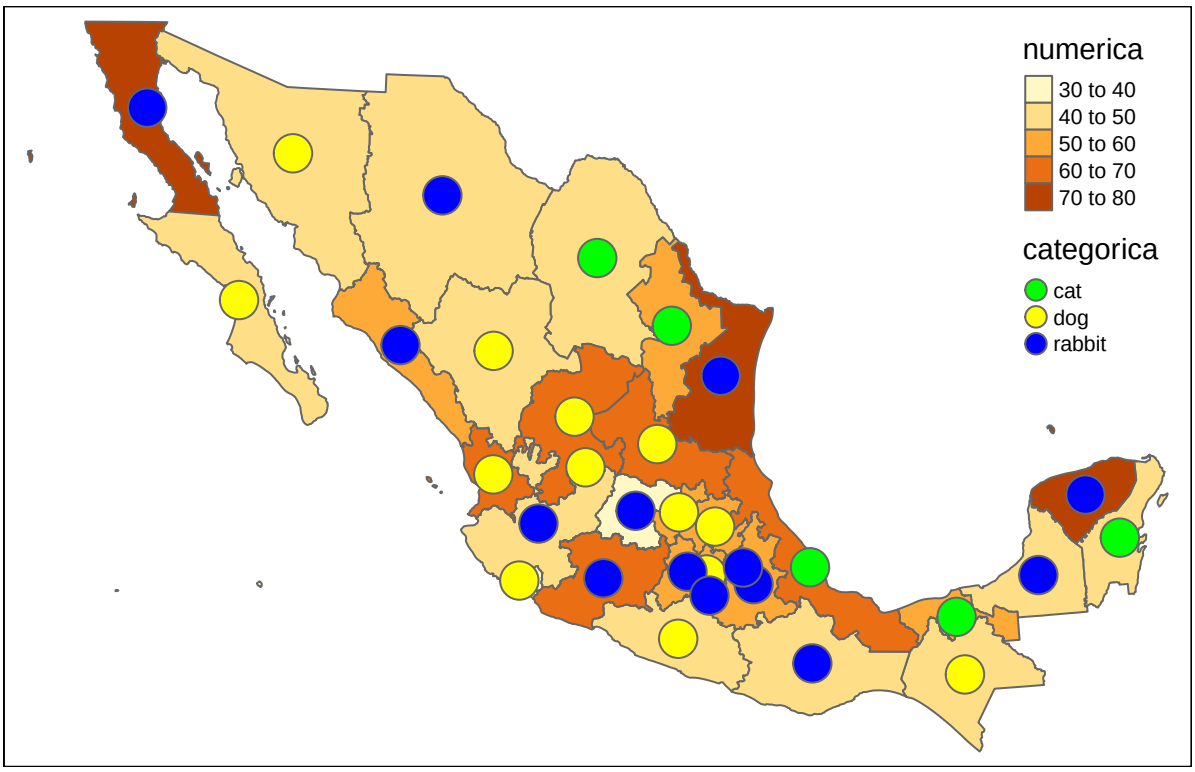
In the following map two variables are plotted. Each state is filled with a color indicating the category of the variable *categorica*, while the size of a circle gives an approximate value of a numerical variable.

```
tm_shape(mapamex_simple) +
    tm_borders() +
    tm_fill(col ="categorica",
            palette = pal,
            style ="cat",
            title = "Animal") +
    tm_bubbles(size= "numerica2",
               legend.size.is.portrait = TRUE,
               title.size = "Ammount", col="blue") +

    tm_layout(main.title = "Pets in México")
```



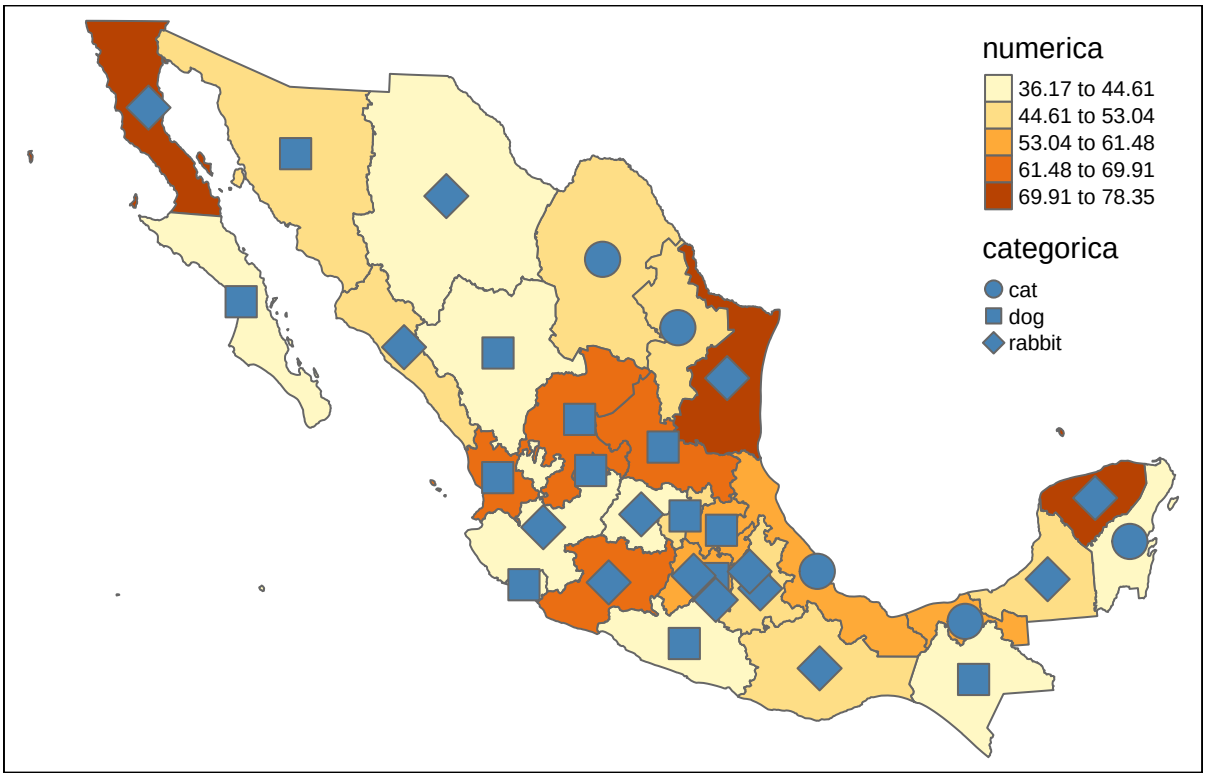
In this map we do the opposite. Fill the states with a numerical variable, while plotting the categorical variable as a different colored dot.

```
pal = c("green", "yellow", "blue")

tm_shape(mapamex_simple) +
    tm_borders() +
    tm_fill(col ="numerica") +
    tm_bubbles(col= "categorica",
               palette = pal,
               legend.size.is.portrait = TRUE) +
    tm_layout(main.title = "Pets in México")
```



Another way to plot a categorical variable is by using symbols of different shapes. The user can specify the size and color of the symbols, while the shape is given by the value of the categorical variable.

```
tm_shape(mapamex_simple) +
    tm_borders() +
    tm_fill(col ="numerica",
            style ="equal" ) +
    tm_symbols(shape= "categorica",
               legend.size.is.portrait = TRUE,
               col = "steelblue",
               size = 1.5    )
```



5 Saving the maps.

Once we have a map we like, it can be saved into several formats, including jpg or png. The first step is to create an object with the map. This is done by choosing a name for the map and assigning the map to this name. For example, to create an object called #mymap#, we simply add "mymap <-" to the code #before# creating the map:

```
mymap <-
    tm_shape(mapamex_simple) +
    tm_borders() +
    tm_fill("categorica", palette = pal,   style ="cat", title = "Animal") +
    tm_legend(position = c(0.1,0.1),
              title.color = "steelblue",
              title.size = 2,
              text.size = 1,
              text.color = "red",
              title.fontfamily = "serif") +
    tm_layout(title = "Pets in México",
              title.size = 2,
              title.color = "black",
              title.position = c("center", "top")
    )
```

Once the object is created, we can use the function *tmap_save*. The easiest way to create an image of the map is to use the *tmap_save*, with two arguments. The first argument is the name of the object we created (in this case "mymap") and the second argument is the name we want the file to have (which can include the path), with the appropriate extension (jpg, png, etc), written in quotes:

```
tmap_save(mymap, "mymap.jpg")
tmap_save(mymap, "mymap.png")
```

The size of the map can be changed in order to make the image bigger or smaller. The options used are *width* and *height*. By altering just one dimension, the other gets automatically adjusted, keeping the proportions.

```
tmap_save(mi_mapa, "mimapa2.jpg",width=4000)
```

6 Recap.

So we reach the end of this little tutorial. I hope the reader found it useful.

There are many other resources for mapping with *tmap*. Good and easy to follow introductions be found here and here. A very comprehensive guide can be found in this book. It explains very thoroughly the differencie between a vector map (as the ones used here) and and a raster map (an image) and instructions on how to combine them.