Plotting choropleth maps in R with tmap

using CSO ESRI shapefiles and StatBank data

Eoin Horgan

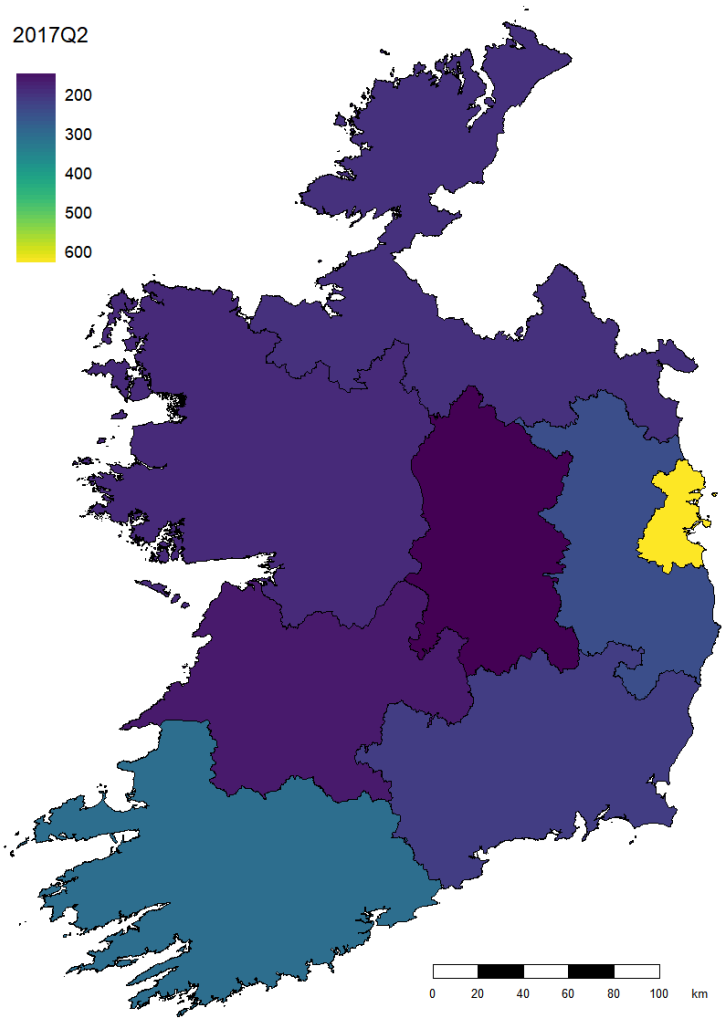
January 27, 2020

Contents

1 Introduction

This tutorial will lay out the use of the tmap package of the R programming language to create choropleth maps, as well as how to import data from the CSO statbank so it can be used in such a map, as in Figure 1.

Figure 1: Persons aged 15 years and over in Employment (Thousand), by NUTS3 region, from table QNQ22.



A choropleth map is a thematic map in which areas are shaded or patterned in proportion to the measurement of the statistical variable being displayed on the map.

We will begin with obtaining and plotting the shapefile data, then show how statbank data can be downloaded, joined onto the shapefile data and used as the colour variable for plotting. We will then discuss further options for customising the appearance of these maps.

The `tmap` package will be used for the plotting, as it is a modern, fast and highly customisable plotting package. Alternative packages will be briefly discussed later.

2 Mapping in R

2.1 Getting the data

The maps used in this tutorial are located on the CSO website at http://census.cso.ie/censusasp/saps/boundaries/ED_SA Disclaimer1.htm. These are the 2011 Census Boundaries, the most up-to-date data available. A variety of files are available, NUTS 3, NUTS 2, administrative counties, electoral divisions, etc. These files are ESRI shapefiles, a mostly open vector format. Each of the zip archives contains a main `.shp` file as well as several necessary auxiliary files.

The process of getting StatBank tables and Irish geographic data is simplified by the use of the `csodata` package, which is developed by the CSO. If you do not already have it installed, then type

```
install.packages("csodata")
```

into the R console. This is only necessary the first time you wish to use the `csodata` package, after the first time it is not necessary. Then we must load and attach the package. This is required every time you want to use a package, but only needs to be done once in an R session.

```
library(csodata) # reading shapefiles and manipulating shapedata
```

then the map data can be downloaded and made available in R very simply by running:

```
shp <- cso_get_geo("NUTS3")
```

a variety of different flags can be used to retrieve different sets of geographic data.

After the package is loaded, you can view the help page for this function using

```
?cso_get_geo
```

to see all of the options. The documentation for any command can be brought up by typing a question mark, followed by the command name.

The `csodata` package uses the `sf` package for loading the shapefiles, and manipulating the map data. The map data object shp is a sf (simple features) object:

```
> class(shp)
[1] "sf"         "data.frame"
```

This is like a normal R data frame, except that it has a special column "geometry". This contains all the vector data required to draw a polygon. Even if you carry out a subset operation on `shp` that would normally exclude this column it is still included with the returned data frame. The only way to remove this column is to use `st_drop_geometry()`. All the functions from the `sf` package have the prefix `st_`, and work on sf objects.

`shp` is an object with 8 observations and 19 variables. There are 8 NUTS 3 regions, so what are the variables?

```
> names(shp)
[1] "NUTS1"  "NUTS1NAME"  "NUTS2"  "NUTS2NAME"  "NUTS3"  "NUTS3NAME
   "  "GEOGID"  "MALE2011"  "FEMALE2011"  "TOTAL2011"
[11] "PPOCC2011"  "UNOCC2011"  "HS2011"  "VACANT2011"  "PCVAC2011"
    "TOTAL_AREA"  "LAND_AREA"  "CREATEDATE"  "geometry"
```

The data includes the NUTS region code and name for NUTS levels 1, 2 and 3, as well as the number of male and female residents, permanent private housing units, and the land area of the region. An explanation for the variables is on the CSO website.

Lets take a look at some of the more interesting columns:

```
> shp[, 5:7]
Simple feature collection with 8 features and 3 fields
geometry type:  MULTIPOLYGON
dimension:      XY
bbox:           xmin: 17491.14 ymin: 19589.93 xmax: 334558.6 ymax:
   466919.3
epsg (SRID):    NA
proj4string:    +proj=tmerc +lat_0=53.5 +lon_0=-8 +k=1.000035 +x_0=200000
    +y_0=250000 +datum=ire65 +units=m +no_defs
  NUTS3        NUTS3NAME GEOGID geometry
1 IE024 South-East (IE)      R6 MULTIPOLYGON (((226795.5 90...
2 IE025 South-West (IE)      R7 MULTIPOLYGON (((18146.05 95...
3 IE011           Border     R1 MULTIPOLYGON (((306570.4 30...
4 IE012          Midland     R5 MULTIPOLYGON (((223420.6 29...
5 IE013             West     R8 MULTIPOLYGON (((48596.74 26...
6 IE021           Dublin     R2 MULTIPOLYGON (((324832.3 22...
7 IE022         Mid-East     R3 MULTIPOLYGON (((283537.7 29...
8 IE023         Mid-West     R4 MULTIPOLYGON (((195253.8 21...
```

We can see that the projection is a transverse Mercator centred on Ireland. `tmap` can correctly plot most projections, but `sf` can transform between projections using `st_transform()` if required (some web-based interactive maps require the WGS 84 projection, for example). Each region is uniquely identified by a name, NUTS 3 code and a GEOGID. When we import the statbank datafile we will need at least one of these to be in the data so that we have a matching variable to perform a join on.

2.2 Plotting map data

The tmap package is a mapping package specialised for thematic maps, which show features relating to a particular theme or aspect. Install the tmap package (if necessary) and load it:

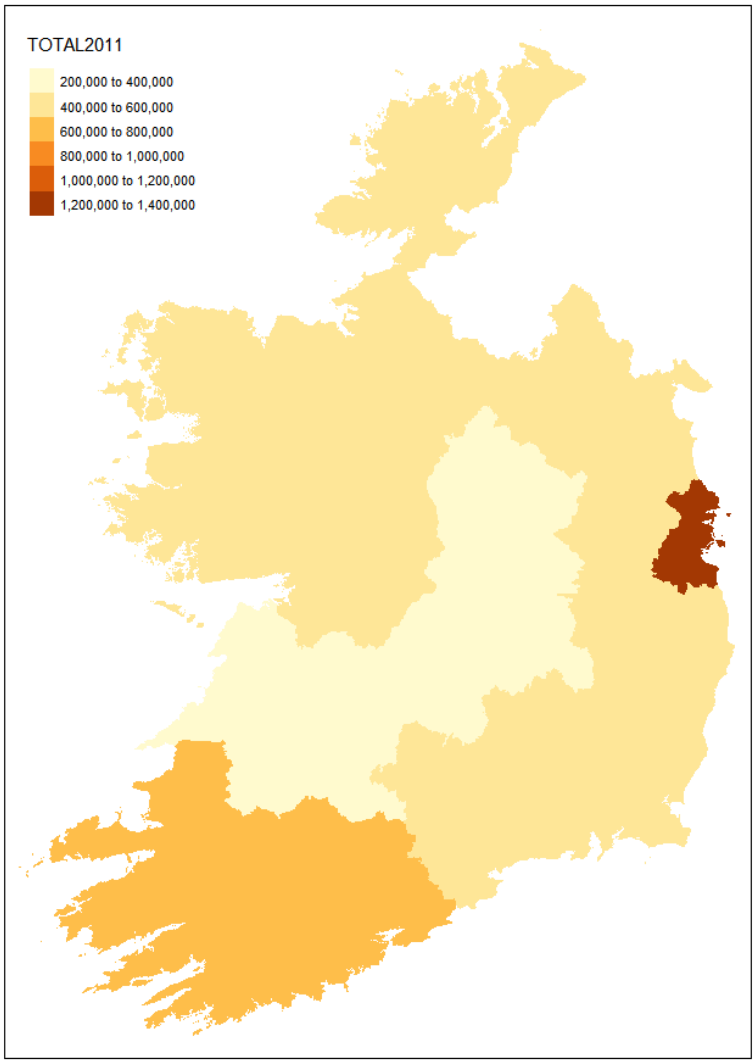
```
# install.packages("tmap") # Create comments using hashmarks
library(tmap)
```

`tmap` uses `ggplot2` style commands – an initial declaration of a plot with the data to be plotted, followed by commands added onto the first in order to refine the visualisation. As an example we will do a simple choropleth using the `TOTAL2011` column, the total population of the region in 2011. This data comes included with the shapefile, so it is useful for showing the capabilities of tmap without needing to merge another data set onto `shp`.

```
t <- tm_shape(shp) + tm_fill(col="TOTAL2011")
t
```

This creates a tmap object `t` based on the data source `shp`, and then fills its polygons, with colour dependant on the value in the `TOTAL2011` column. The image will not be

Figure 2: Total population in 2011, from ESRI shapefile provided by CSO.



rendered and displayed until you call on the tmap object, using either "`t`" or "`print(t)`". Doing so results in Figure 2.

This is the basic usage of the `tmap` package. Now it is only a matter of customising the appearance of the plot and adding new data onto the shapefiles so it can be plotted.

2.3 Customising map appearance

This is just an overview of the most important features required for customising a map. For more depth, readers should consult the documentation for the tm_fill and tm_layout commands on rdocumentation.org or in your console. Remeber, to bring up the relevant documentation for `tm_fill`, use:

```
?tm_fill
```

2.3.1 Colour palette

The colour palette used should be passed to `tm_fill` as `palette = <your palette>`. I prefer using `palette = viridisLite::viridis(20)` for continuous data. Viridis ranges from dark purple to light green, maintains a gradient when printed in grayscale, and is easier to read for colourblind users[1]. By default a legend will be generated and automatically positioned to avoid overlap with the map. There are many options relating to the legend, most importantly the default title of the legend can be overridden using the argument `title` and the legend can be shown in ascending order (smallest on bottom) using `legend.reverse = TRUE`.

The supplementary package `tmaptools` includes an interactive colour explorer that can be used to see and compare sequential, categorical or diverging palettes and get the code required to generate them. Enter `library(tmaptools); palette_explorer()` in the console to activate it. If you choose a diverging palette then you should also set the argument `midpoint`, which defaults to 0.

Additionally `colorNA ="<colour name>"` can be used to set the color used for NA values. See the ggplot2 colour names for acceptable names or type `colors()` in the console to get a list of all the R colours.

```
t <- tm_shape(shp) +
     tm_fill(col="TOTAL2011", palette = viridisLite::viridis(20),
       colorNA = "grey50", legend.reverse = TRUE,
       title = "Population 2011")
t
```

2.3.2 Clustering

By passing the argument style to `tm_fill`, you can choose whether to use discrete bins for the data or a continuous scale. From the `tm_fill` documentation: *"method to process the color scale when col is a numeric variable. Discrete options are "cat", "fixed", "sd", "equal", "pretty", "quantile", "kmeans", "hclust", "bclust", "fisher", "jenks" and "log10_pretty". A numeric variable is processed as a categorical variable when using "cat", i.e. each unique value will correspond to a distinct category. For the other discrete options (except "log10_pretty"), see the details in classIntervals*[2].

Continuous options are "cont", "order" and "log10". The first maps the values of col to a smooth gradient, the second maps the order of values of col to a smooth gradient, and the third uses a logarithmic transformation."

Also note that when using one of the discrete options the argument `n` can be used to determine the number of bins to sort the data into.

```
t <- tm_shape(shp) +
     tm_fill(col="TOTAL2011", style="fisher", n = 3)
t
```

[1] See https://cran.r-project.org/web/packages/viridis/vignettes/intro-to-viridis.html for more about viridis and the other viridis palettes

[2] https://www.rdocumentation.org/packages/classInt/versions/0.1-7/topics/classIntervals

2.3.3 Borders

Borders can be drawn between regions by using the command `tm_borders`. The colour can be chosen with argument `col="<colour name>"`, and like colorNA can be any of the R colour names. The line width is set with argument `lwd=<desired line width>`.

```
t <- tm_shape(shp) +
     tm_fill(col="TOTAL2011") +
     tm_borders(col = "black", lwd = 1)
```

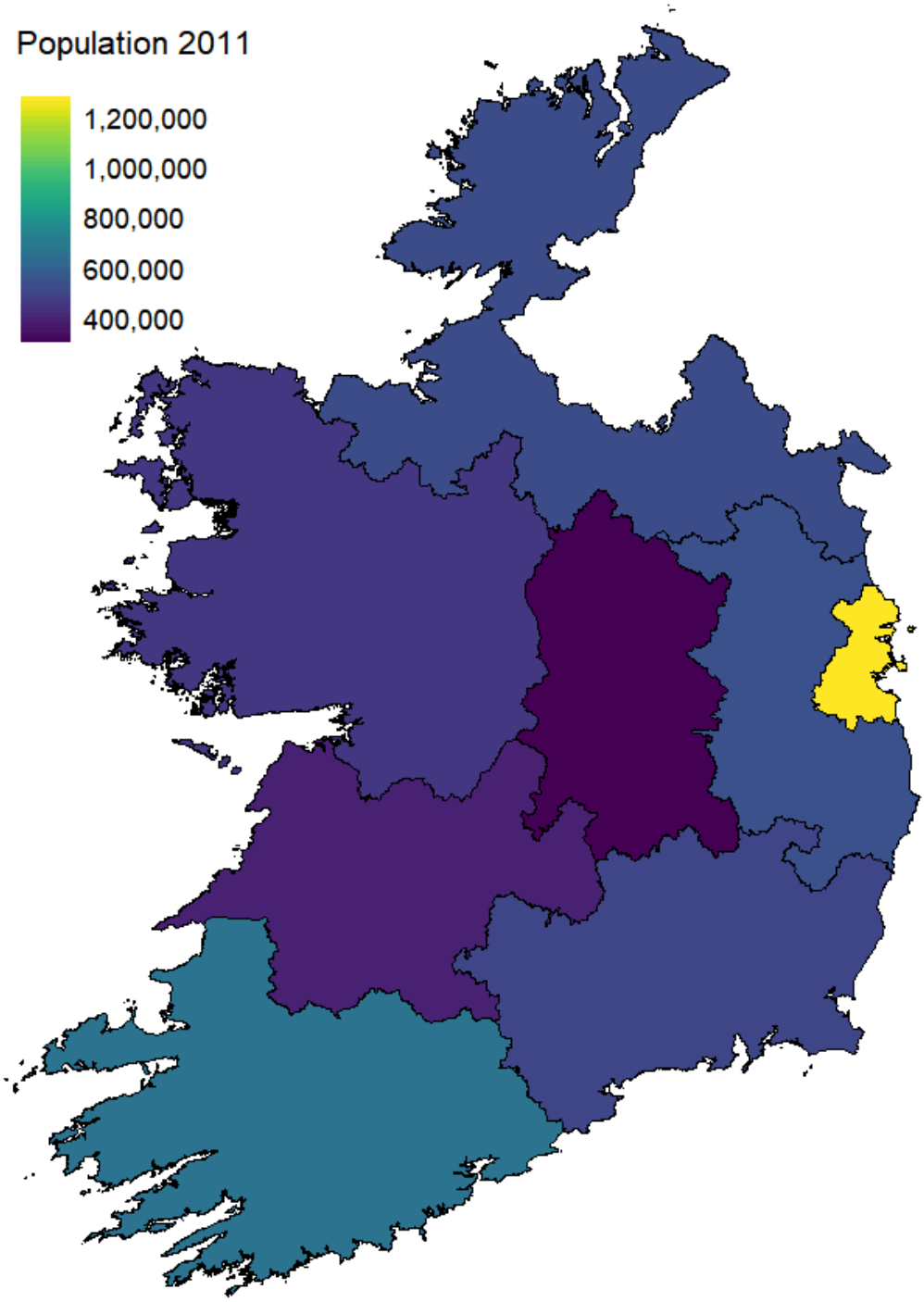
2.3.4 Miscellaneous options

Many other visual options can be set using the `tm_layout` function, for example adding a title to the map with `title = "<Your title>"`, removing borders with `frame = FALSE` or changing the scale of all text and the legend using `scale = <A number, larger than 1 for increased scale>`. The last option is particularly useful when saving figures, if you increase the resolution when saving the legend is not increased proportionally, so the scale argument is necessary so that the legend is still readable.

```
t <- tm_shape(shp) +
     tm_fill(col="TOTAL2011", palette = viridisLite::viridis(20),
       style="cont", legend.reverse = TRUE,
       title = "Population 2011") +
     tm_borders(col = "black") +
     tm_layout(frame = FALSE, scale = 1.6)
t
```

Using all of these options together results in Figure 3.

Figure 3: Total population in 2011. Same data as in Figure 2, only aesthetic options differing.



3 Importing StatBank data

3.1 Importing StatBank tables

Here we again use the `csodata` package to simplify the process of acquiring our data. We can download the QNQ22 dataset (Persons aged 15 years and over by NUTS 3 Regions, Quarter and Statistic) from the CSO statbank into the "df" object with one command;

```
df <- cso_get_data("QNQ22")
```

Lets take a look at `df`

```
> names(df)
[1] "Statistic"  "NUTS.3.Regions" "NUTS.3.Regions.id" "1997Q4"
   "1998Q1"  "1998Q2"  "1998Q3"  "1998Q4"  "1999Q1"
[9] "1999Q2"  "1999Q3"  "1999Q4"  "2000Q1"  "2000Q2"  "2000Q3"
   "2000Q4"  "2001Q1" ...
# Cut for brevity, continues onto "2017Q2"
```

There is one column for every quarter. Statistic is a factor[3] with five levels, namely:

```
> levels(df$Statistic)
[1] "Persons aged 15 years and over in Employment (Thousand)"
[2] "Unemployed Persons aged 15 years and over (Thousand)"
[3] "Persons aged 15 years and over in Labour Force (Thousand)"
[4] "ILO Unemployment Rate (15 - 74 years) (%)"
[5] "ILO Participation Rate (15 years and over) (%)"
```

Most importantly for us, `df$NUTS.3.Regions.id` contains the codes of the NUTS 3 regions, and we can join this into `shp` on `shp$NUTS3`.

```
> levels(df$NUTS.3.Regions)
"State"       "Border"      "Midland"     "West"        "Dublin"      "
   Mid-East"    "Mid-West"    "South-East" "South-West"
```

```
> levels(df$NUTS.3.Regions.id)
"-" "IE11" "IE12" "IE13" "IE21" "IE22" "IE23" "IE24" "IE25"
```

```
> shp$NUTS3
"IE024" "IE025" "IE011" "IE012" "IE013" "IE021" "IE022" "IE023"
```

However if we do this and then check our results with `summary(is.na(shp))` we will find multiple NA values. Unfortunately the codes of NUTS 3 regions are not identical, as `shp` includes a preceding 0. We need to manually adjust one of the tables, for example by using:

```
shp$NUTS3 <- str_remove(shp$NUTS3, "0")
```

[3] See here if you are unfamiliar with factors in R.

3.2 Joining data

We will use the package `dplyr`[4] in order to join the data frames.

```
# install.packages("dplyr")
library(dplyr)
```

`dplyr` makes seven types of joins available, but we will only need to use a basic left join.

```
shp <- subset(shp, select = -c(NUTS1, NUTS1NAME, NUTS2, NUTS2NAME,
              MALE2011, FEMALE2011, TOTAL2011, PPOCC2011, UNOCC2011,
              HS2011, VACANT2011, PCVAC2011, TOTAL_AREA,
              LAND_AREA, CREATEDATE))
shp <- left_join(shp, df, by = c("NUTS3" = "NUTS.3.Regions.id"))
```

Since there are many unused columns in the data frame we can drop columns we are not using, by passing the `subset` function with a minus sign in front of a vector of column names. Then we can join `shp` and `df` by the columns `shp$NUTS3` and `df$NUTS.3.Regions.id`. If you get a warning about coercing character factors to character vectors during the join then it can be safely ignored.

Next we will choose the quarter and statistic to plot

```
var  <- "2017Q2" # Choose from "1997Q4" to "2017Q2"
stat <- as.character(levels(shp$Statistic)[4]) # Choose from 1 to 5
# [1] Persons aged 15 years and over in Employment (Thousand)
# [2] Unemployed Persons aged 15 years and over (Thousand)
# [3] Persons aged 15 years and over in Labour Force (Thousand)
# [4] ILO Unemployment Rate (15 - 74 years) (%)
# [5] ILO Participation Rate (15 years and over) (%)
```

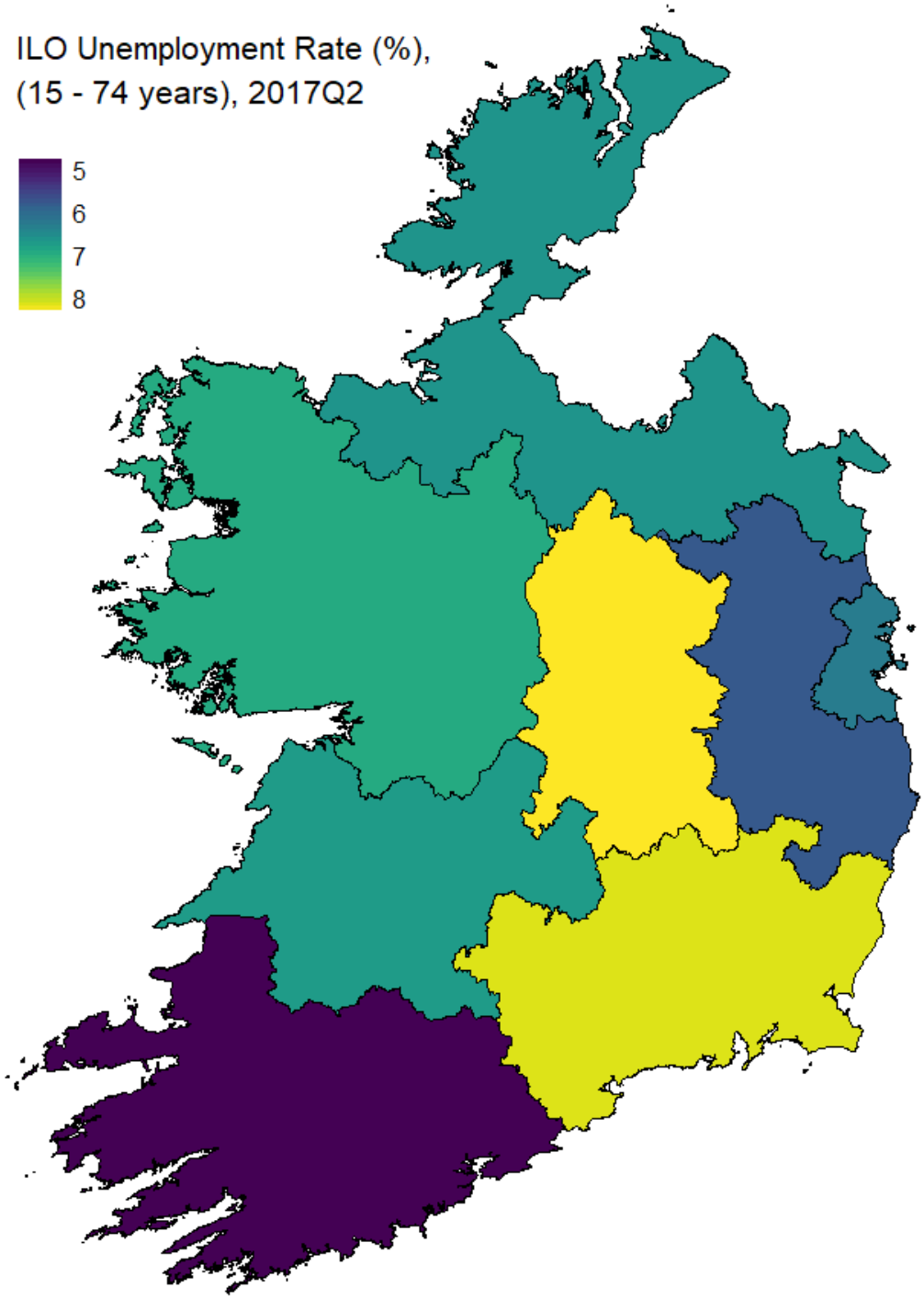
Finally, we can plot this data using what we learned in § 2.2 to obtain Figure 4.

```
t <- tm_shape(shp[shp$Statistic == stat, ]) +
    tm_fill(col=var, palette = viridisLite::viridis(20),
    style = "cont", colorNA = "grey50",
    title = "ILO Unemployment Rate (%),\n(15 - 74 years), 2017Q2",
    popup.vars=c("GEOGID", var)) +
    tm_borders(col = "black") +
    tm_layout(frame = FALSE, scale = 1.1, legend.width = 0.7)
t
```

After matching, remember to check your joined data using `summary(is.na(shp))` to see if there are any NA values, which may indicate missing data in the original dataset or a failure to match elements in the two datasets.

[4] `dplyr` is part of the `tidyverse`, a collection of packages for data science. Discussion of its features and merits is beyond the scope of this tutorial, but if the reader is interested in R then it's well worth looking into.

Figure 4: ILO Unemployment Rate (15 - 74 years) (%) by NUTS 3 region, 2017 quarter two, from table QNQ22.



4 Other considerations

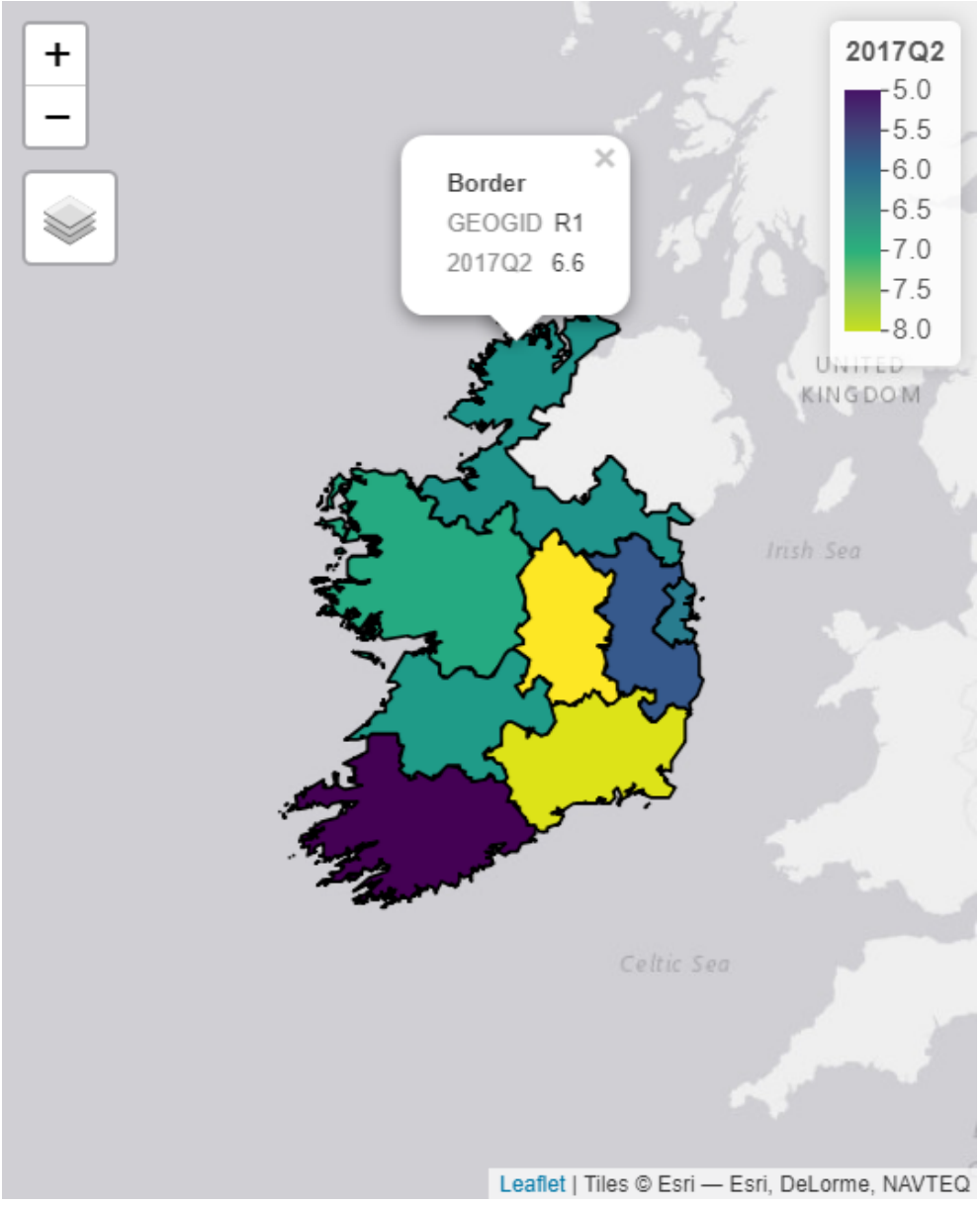
4.1 Interactivity

You may have noticed the argument `popup.vars` in the last code excerpt of the previous section. This is an option that is only used in tmaps interactive mode. To see it in action type this into the console:

```
tmap_mode("view")
t
```

You should see something like Figure 5.

Figure 5: ILO Unemployment Rate (15 - 74 years) (%) by NUTS 3 region, 2017 quarter two. Data from table QNQ22, interactive plot.



The popup variables are set by us using the `popup.vars` argument, but the first element (in the image, the bolded Border at the top of the popup) is taken from the first column

of the data frame passed to tmap, in this case the `NUTS3NAME` column. This also appears when hovering over one of the regions.

One thing to note is that by default the map files encode any text fields as "UTF-8", and unfortunately due to this Unicode characters may not be displayed properly on the popup. This is most likely to occur when attempting to display place names that include a fada. In order to display these correctly, you can convert the name field to "latin1" encoding using the `iconv` function. For example, with the electoral division dataset:

```
shp$EDNAME <- iconv(shp$EDNAME, to = "latin1")
```

Which will result in fadas properly displaying in the popup.

This method is an easy way to convert a tmap map you have already made into an interactive map, but if your main goal is to make interactive maps, then I recommend not using tmap. This is because the tmap object you create is converted into a leaflet map and then displayed, but using pure leaflet is significantly faster and gives more options for controlling interactive plots. For example you can change the colour variable without redrawing the polygons, allowing the map to update almost instantly.

In order to return to normal plotting mode, use:

```
tmap_mode("plot")
```

4.2 Other tmap features

This is a brief discussion of some of the other features of `tmap` which we have not shown in this tutorial, but which may be of interest to the reader.

4.2.1 Layers

Multiple `sf` objects can be plotted in the same tmap object in order to plot them next to each other or to plot layers on top of each other.

4.2.2 Facets

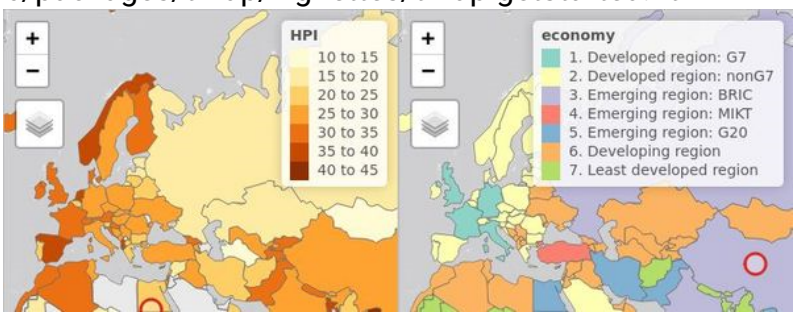
A tmap object can be created with multiple facets, for example, displaying multiple maps side-by-side with different data, or splitting one large map into multiple smaller ones by its constituent polygons and displaying them next to each other (for example splitting a map of Ireland's counties by province to create four smaller maps). It's also possible to join these facets together into an animated `.gif` using the `tmap_animation` function.

4.3 Saving figures

If you wish to save your figures for any reason then there are two options available to you. If you are using RStudio, then you can click on Export above the plot window, then Save as Image. This will bring up a new window allowing you to choose where to save the image, the format used and allowing you to resize while maintaining the aspect ratio.

If you prefer to do so programmatically you can use :

Figure 6: Example of facets from the tmap: get started! vignette: https://cran.r-project.org/web/packages/tmap/vignettes/tmap-getstarted.html




```
tmap_save(tm = NULL, filename = NA, width = NA,
          height = NA, units = NA)
```

Where `tm` is the tmap object (in these examples `t`). `filename` is the filename, optionally including a full path. `units` is one of "in", "cm", "mm" or "px" (pixels) and `width` and `height` are the image size in your chosen dimensions.

Note that these methods resize the legend and text in different ways. You will usually need to use `scale` larger than one when using RStudio and smaller than one with `tmap_save`.

4.4 Other packages

We have only briefly touched on the usage of two packages, `sf` and `tmap` but the R ecosystem of packages for mapping is large and growing by the day, and there are many other good options that can be used. There are far too many packages to give a fair treatment of them all, so only a brief description will be given here, and the reader can investigate further if they are interested by any of them.

4.4.1 Spatial data manipulation

Before `sf` became the standard for vector geometry `sp`, along with the auxiliary packages `rgdal` and `rgeos` were the main packages used for spatial data manipulation. You may still see them used in some older guides online, but they have largely been superseded by `sf`, which is easier to work with and much, much faster.

4.4.2 General plotting packages

`plotly` is a package that can be used for general plotting as well as mapping, however it suffers from being so general its maps were slow to render and not as customisable as other packages.

`ggplot2` is a widely used general plotting package that can also be used for mapping. Due to it's popularity there is a huge amount of information about it online, if you're having trouble getting your map to look correct. However it's only possible to make it interactive by converting to a plotly object. The `tmap` style of added commands is based on the syntax of `ggplot2`'s commands.

`googleVis` is a package that can be used for general plots as well as interactive maps on top of a google maps basemap.

`Highcharter` is a package for interactive general charts, stock plotting over time and mapping. It has shiny integration and a fairly extensive series of examples, and by default produces good looking graphs and maps. It uses `tidyverse` style piped commands, instead of `ggplot2` style addition. However unlike the other options on this list it is a paid product, requiring a licence for commercial use.

4.4.3 Mapping packages

`Leaflet` is conversion of a popular JavaScript library for interactive maps. It's one of the faster interactive mapping packages, and has a wide range of basemaps that can be used[5]. It also has integration with shiny, so that it can be run in and dynamically updated in a shiny app. It uses `tidyverse` style piped commands.

`mapview` is a package for quickly viewing spatial data in interactive maps. It is intended to be used to gain an understanding of a dataset, not for presentation grade visualisations.

`rCharts` is a package used to access JavaScript visualisation library's in R. It uses a formula style syntax based on the lattice package and requires some familiarity with JavaScript. Some of the JavaScript libraries available through this package, like Highcharts and leaflet already have native R implementations.

4.5 More information

If you felt that your skills in R were lacking or prevented you from following along then try going through this tutorial https://www.tutorialspoint.com/r/index.htm, or acquire and read a copy of R in a Nutshell by Joseph A. Adler.

If you liked this tutorial but felt it didn't go far enough in discussing mapping techniques then have a look at Geocomputation with R by Robin Lovelace, Jakub Nowosad and Jannes Muenchow. You can purchase a physical copy, but it's also available for free online at https://geocompr.robinlovelace.net/index.html.

If you understood everything and want more interactivity try looking up shiny apps, which can allow you to make html webapps and allow them to be run by users without R installed. See https://shiny.rstudio.com/tutorial/ for a tutorial on shiny.

[5] See http://leaflet-extras.github.io/leaflet-providers/preview/ to see them all.