Data Visualization in R

tidyverse, ggplot and rmarkdown

Matteo Coronese

November 25th 2019

Sant'Anna School of Advanced Studies - Pisa, Italy

Outline

Data Manipulation: tidyverse

Data Visualization: ggplot

Reporting: rmarkdown

Data Manipulation: tidyverse

R for Data Science: the *tidyverse* package

R is a versatile software for data science. The collection of packages named **tidyverse** makes data manipulation and visualization easier, faster and more intuitive.

```
install.packages('tidyverse')
```

tidyverse is essentially an ecosystem for data manipulation, comprising several packages. Most important ones:

- *dplyr*: Data transformation (e.g. filtering, summarizing, mutating).
- *tidyr*: Heps you achieving tidy datasets (i.e. properly formatted and encoded, no missing values etc.).
- *ggplot*: Data visualization and plotting.
- *readr*: Advanced data importing.

Cheatsheets can help you familiarizing with most basic commands: data wrangling (*dplyr* and *tidyr*), *ggplot* and *readr*.

For a more comprehensive guide, check out R for Data Science (Grolemund and Wickham, 2016).

Meet %>% , the *magrittr* pipeline operator

The *magrittr* package allows you to write subsequent operations in a single pipeline, through the `\%>\%` operator. It passes the object on LHS as first argument of function on RHS:

If you want to perform $x \to f(\cdot) \to g(\cdot)$

Rather than $g(f(x))$ write instead x %>% $f(\cdot)$ %>% $g(\cdot)$

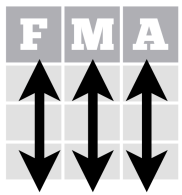
```
data("mtcars")                          #load example data
mean(exp(mtcars$cyl))                   #usual workflow
mtcars$cyl %>% exp() %>% mean()         #pipeline
```

When more complex operations are involved, this results in a much clearer and more manageable code, and avoids unnecessary saving of variables in the workspace.

Also, it makes the usage of *ggplot* much easier.

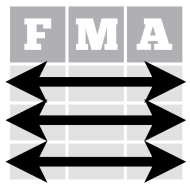
Work with tidy datasets

Use *tidyr* commands (not explained today) to clean up and organize your dataset in a "tidy" way. Always start with a dataset like this:



Each **variable** is saved in its own **column**

&



Each **observation** is saved in its own **row**

Subsetting rows/observations is then very easy: just pass to `filter()` logical condition(s) on column/variables:

```
mtcars %>% filter(cyl==4)                    #single condition
mtcars %>% filter(cyl==4, hp>90)             #multiple conditions
```

Also, subsetting variables, through `select()`:

```
mtcars %>% select(mpg, cyl)                  #two columns
mtcars %>% select(-mpg, -cyl)                #all but two columns
```

Make new variables

Create new (or replace) variables through `mutate()`:

```
mtcars %>% mutate(
   hp_100 = hp*100,                    #new variable/column hp_100
   drat = drat/100                     #replaces drat
)
```

Create new variables conditional on values of other variables (e.g. a dummy) through `case_when()`:

```
mtcars %>% mutate(
   gear_5=case_when(                   #creates new variable gear_5
      gear>=5 ~ 1,                     #logical condition ~ value
      gear<5 ~ 0
   )
)
```

NOTE: If you want to store your changes, just assign through `->` the pipeline to a new object (or use the same name to replace):

```
mtcars %>% mutate(
   hp_100 = hp*100,                    #new variable/column hp_100
   drat = drat/100                     #replaces drat
) -> mtcars                            #overwrites the mtcars object
```

Summarize variables

Some operations involves outputs which are shorter than inputs (e.g. sum, mean). Use `summarise()`

```
mtcars %>% summarise(
   m_cyl=mean(cyl)          #variable m_cyl contains mean cylinders
)
```

Every operation which reduces data dimension can be performed by group(s) through `group_by()`

```
mtcars %>% group_by(gear, am) %>% #every combination of gear and am
   summarise(
      m_cyl=mean(cyl),
      s_hp=sum(hp)
   )
```

Off course, it makes sense to group by a discrete variable. You can always discretize a continuous variable through `mutate()`. **OR operate with long data**.

Wide and Long Data

We typically distinguish between wide data and long data.

- **Wide data**: suitable for working on variables (e.g. `mutate()`).
- **Long data**: suitable for summarizing variables (e.g. `summarise()`). Also, very suitable for plotting with *ggplot*.

Person	Age	Weight
Annah	32	52
Bob	24	98
John	64	76

(a) Wide Data

Person	Age	Value
Annah	Age	32
Bob	Age	24
John	Age	64
Annah	Weight	52
Bob	Weight	98
John	Weight	76

(b) Long Data

With wide data, you can easily e.g. sum by row. With long data, you can easily e.g. group by Year or Variable.

Wide and Long Data II

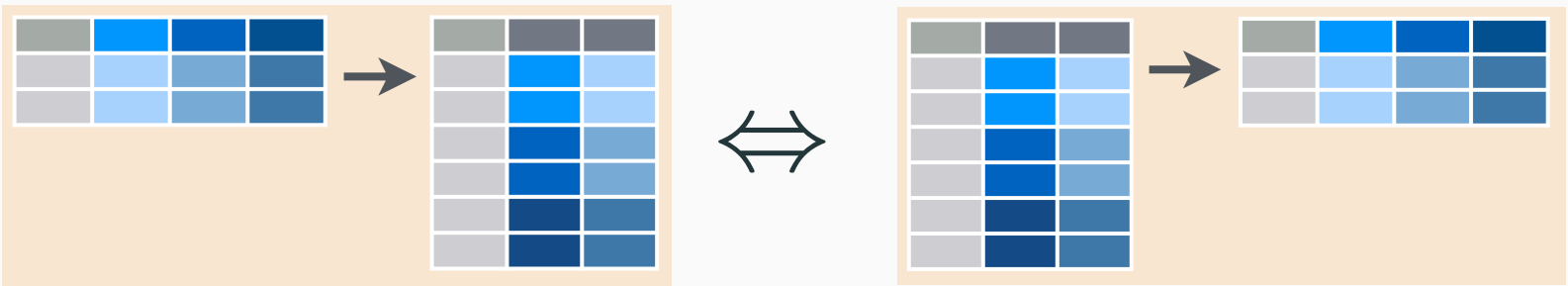
You can obtain long data using `gather()`. You can also pipe with `summarise()` and `group_by()`.

```
iris %>% gather(variable, value, -Species)     #get long data

iris %>% gather(variable, value, -Species) %>% #pipe and summarize
   group_by(variable) %>%                      #group by a var
   summarise(
      m_var= mean(value)
   )

iris %>% gather(variable, value, -Species) %>%
   group_by(variable, Species) %>%             #group by two vars
   summarise(
      m_var= mean(value)
   )
```

The reverse operation is done through `spread()`.



(a) From wide to long: gather

(b) From long to wide: spread

Data Visualization: ggplot

ggplot and the Grammar of Graphics

`ggplot` is a plotting system based on the concept of *Grammar of Graphics* developed by Leland Wilkinson (2005).

- It envisages a statistical graphic as a mapping from data to aesthetic attributes (colour, shape, size) of geometric objects (points, lines, bars).
- The plot may also contain statistical transformations of the data.
- The plot is drawn on a specific coordinate system.
- Faceting can generate the same plot for different subsets of the dataset.

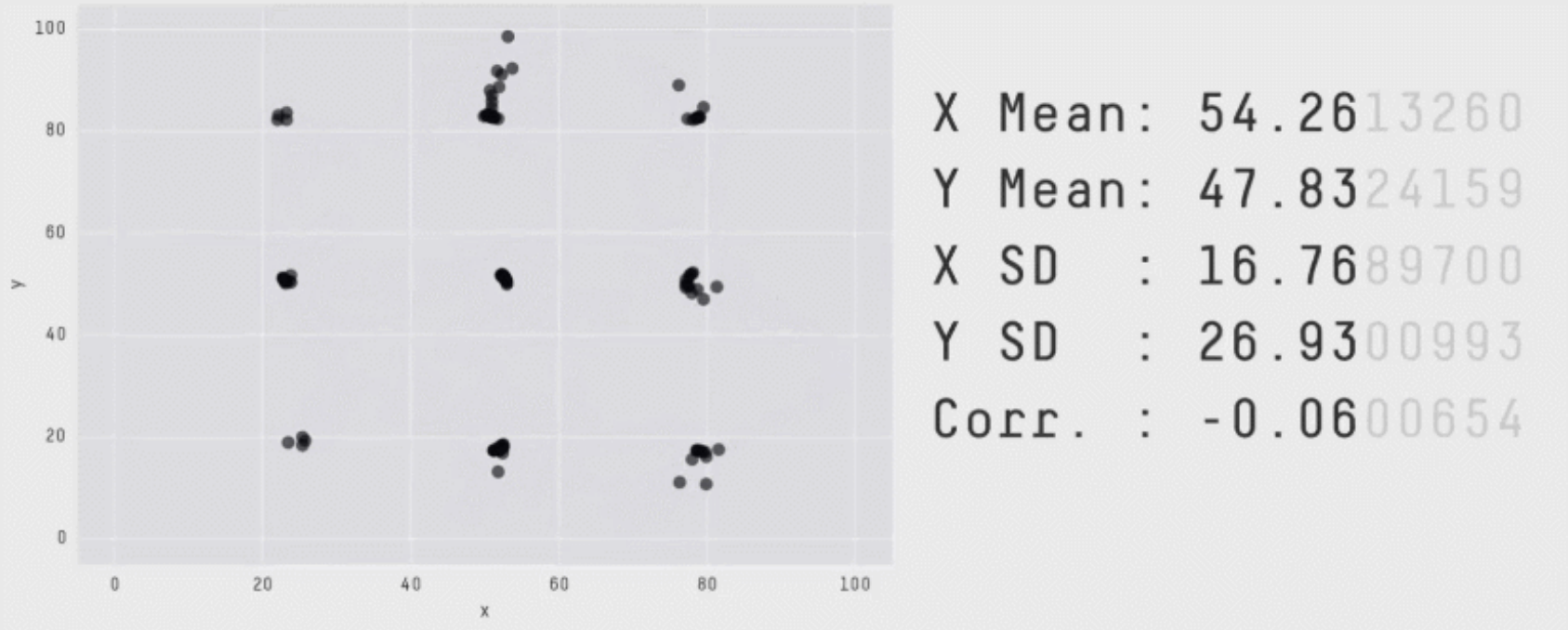
The combination of these independent components makes up a graphic. In addition,

`ggplot`:

- implements the grammar by layers (H. Wickham, 2009).
- incorporates concepts related to statistical graphs and data analysis developed by John W. Tukey (1977), Edward R. Tufte (1990, 1997, 2001).
- allows to account data perception, as documented by William S. Cleveland and Robert McGill.

In other words: **Let ggplot do its job, unless you're very sure!**

Plotting data is important!



Building by layers

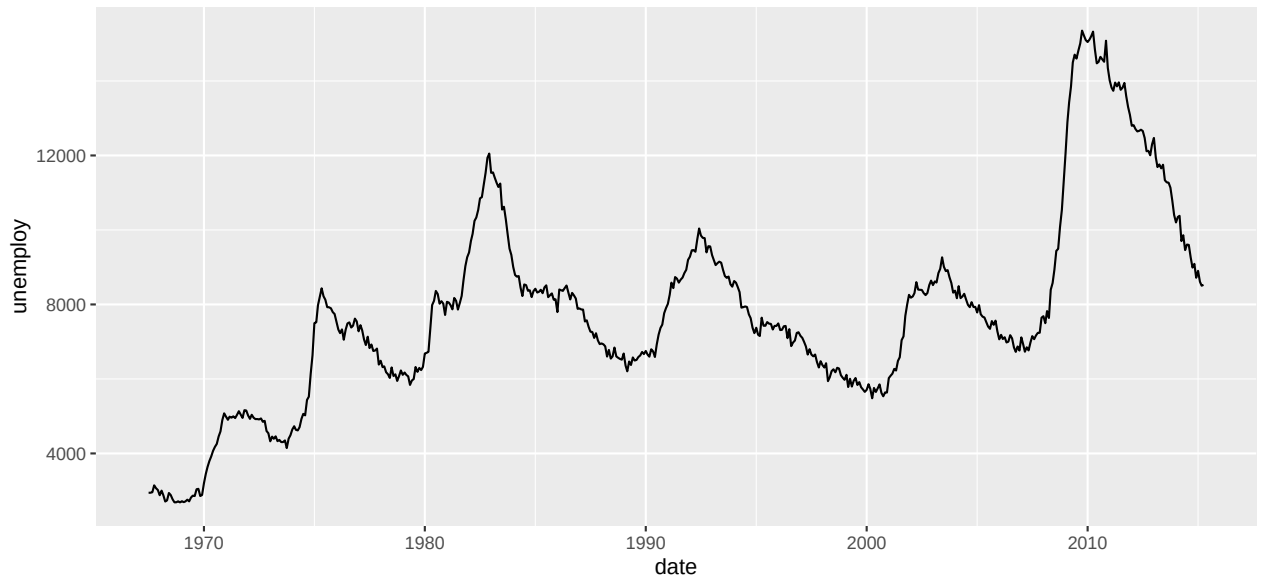
The syntax of `ggplot2` implement the Grammar of Graphics, by layers:

1. First, provide ggplot with the *data.frame* structured **in long form**.
2. Second, call `ggplot()` and specify through `aes()` the aesthetic attributes to be interpreted: variables, colours, shapes, etc.
3. Finally, specify which geometric object should interpret the aesthetic attributes (points, lines, surfaces, etc.). E.g. `geom_line()`, but there is a `geom` for every need.
4. If necessary:
   - modify axis type (scaling, inversion, etc.)
   - specify to facet the plot by some variable (`facet_grid()` or `facet_wrap()`)
   - modify axes scales and labels, grids, title etc. Use `theme()` to modify predefined themes.
5. evaluate and display the graphic.

This is an abstract specification of graphics, which may be confusing at first, but is *very* powerful and versatile.

A first example

```
economics %>%                                   #pass data
   ggplot(aes(x=date, y=unemploy)) +            #specify aesthetics
   geom_line() -> g                             #specify geom
```



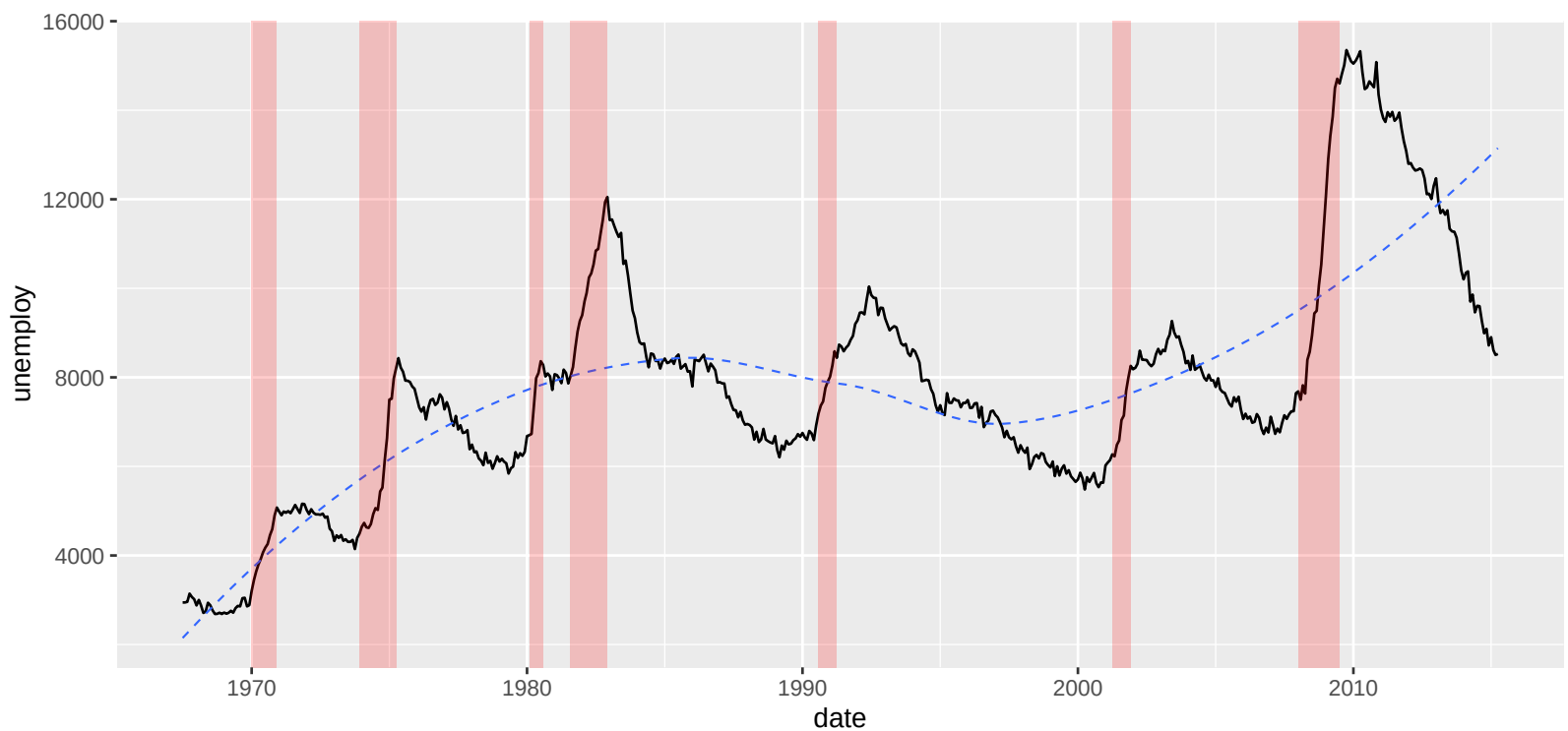
You can elaborate through the pipeline to dynamically change data. You can e.g. filter, but you can summarize, transform in long etc. without need to save.

```
economics %>%
   filter(date>"1990-01-01") %>%                #subset data
   ggplot(aes(x=date, y=unemploy)) +
   geom_line()          #inerith aes from ggplot unless specified
```

Add Layers!

You can add as many layers as you want, possibly using multiple (compatible) data sources.

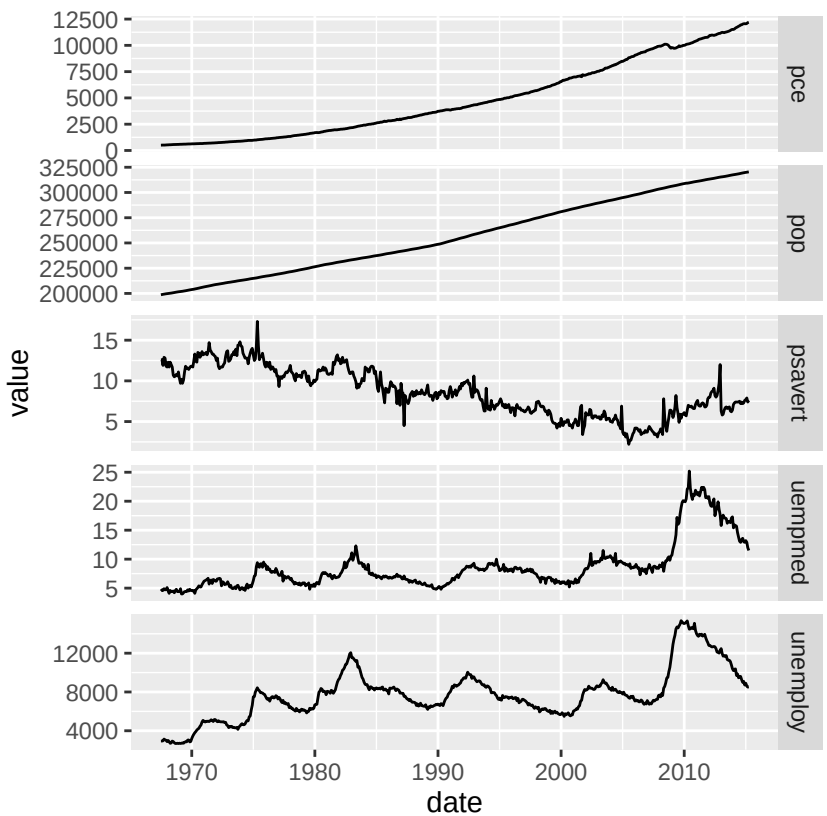
```
economics %>%
ggplot(aes(x = date, y = unemploy)) +
   geom_line() +
   geom_smooth(method = "loess",            #kernel smoothed line
          se=F, size=0.4, linetype=2) +
   geom_rect(data = recessions,             #new dataset
          aes(xmin = start, xmax = end, ymin = -Inf, ymax = +Inf),
          inherit.aes = FALSE, fill = "red", alpha = 0.2)
```



Faceting

Long data comes handy when faceting, i.e. drawing multiple graphs depending on a discrete variable (in this case created through `gather()`).

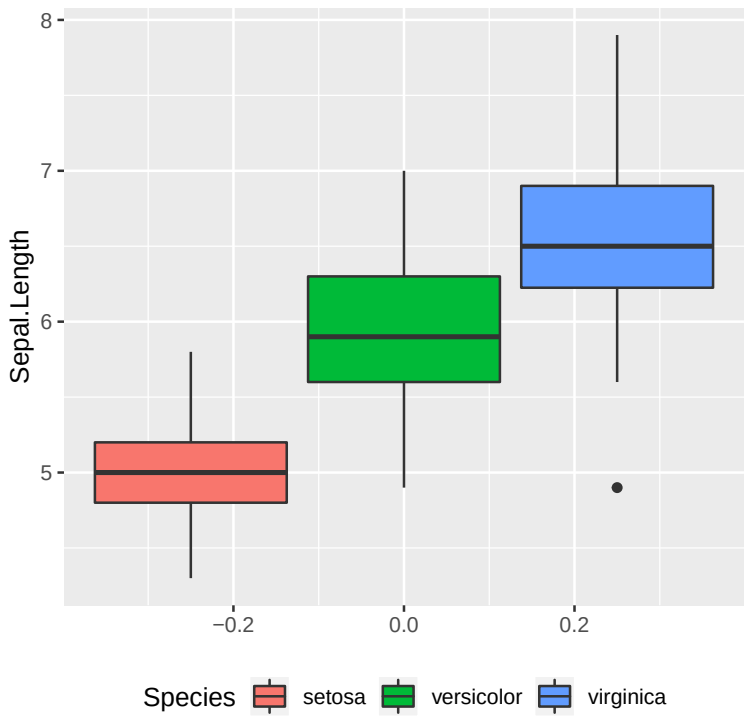
```
economics %>%
   gather(variable, value, -date) %>%
   ggplot(aes(x=date, y=value)) +
   geom_line() +
   facet_grid(variable~., scales = "free_y") #row~column
```



Grouping

The same approach can be used to group, i.e. display on the same graph multiple objects from data subsets depending on a discrete variable.

```
iris %>%
ggplot() +
geom_boxplot(aes(y=Sepal.Length, fill=Species)) +
theme(legend.position = "bottom")
```

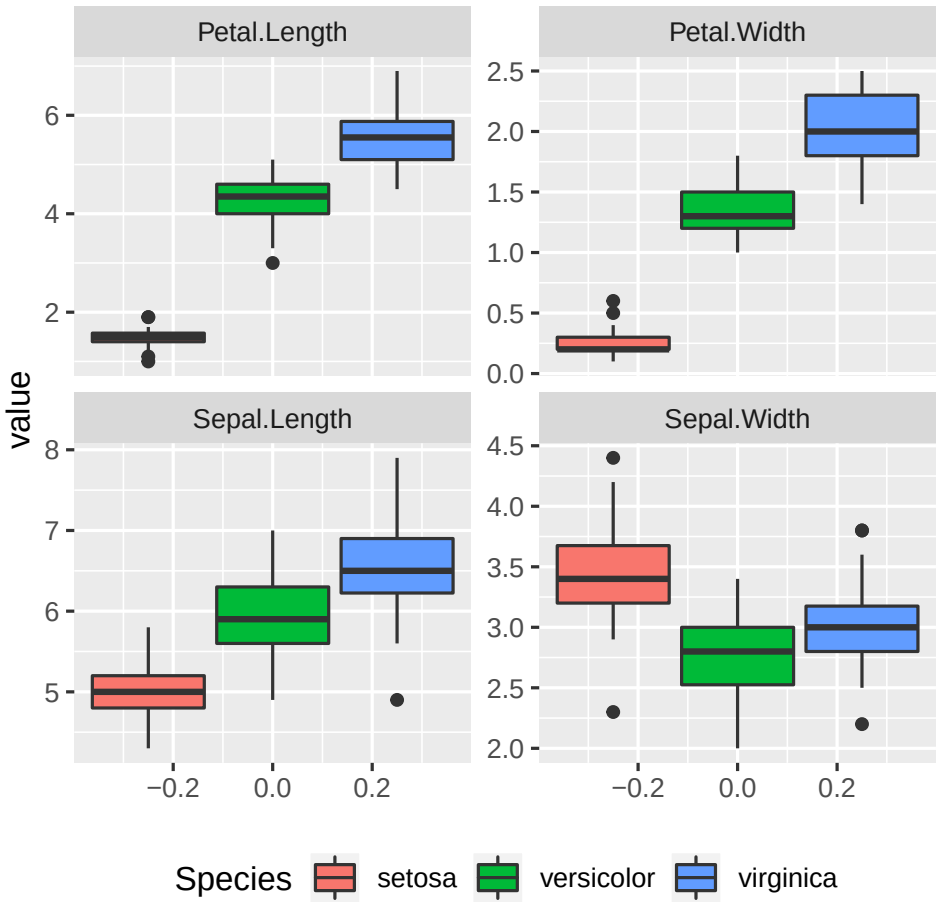


Grouping happens through `aes()` specifying `group`, but also through `fill()`, `color()`, `linetype()`, `alpha()`, `size()` and so on. You can combine them too.

Faceting + Grouping

Now combine faceting and grouping. `facet_wrap` is more flexible than `facet_grid`.

```
iris %>% gather(variable, value, -Species) %>%
   ggplot() +
   geom_boxplot(aes(y=value, fill=Species)) +
   facet_wrap(variable~., scales = "free_y", nrow=2) +
   theme(legend.position = "bottom")
```



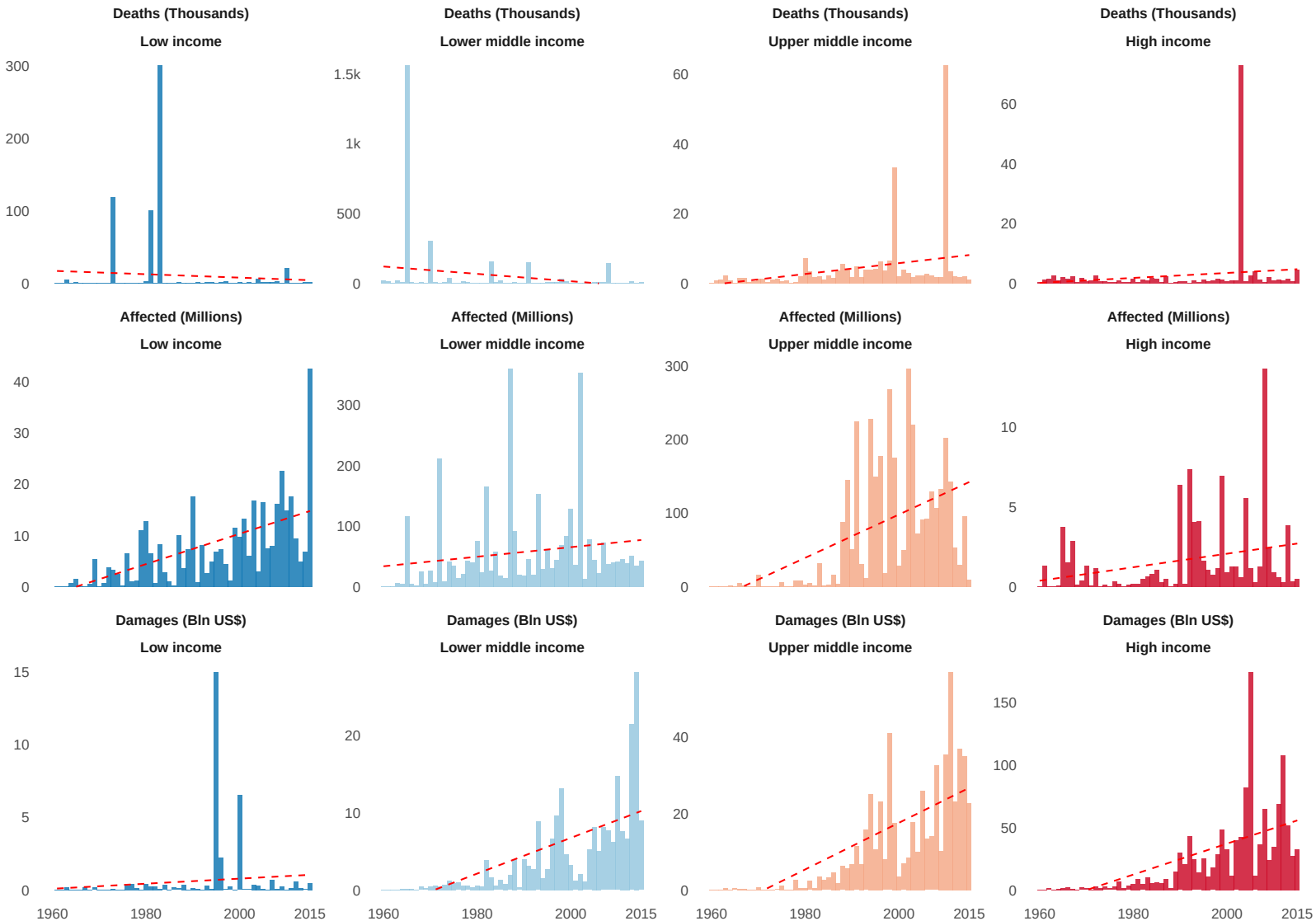
Some Tips on Appearance

- Themes:
  - Predefined themes (colors, spacing,) can be set globally using `theme.set()` (e.g. `theme.set(theme_light())`) or locally in the ggplot pipe (e.g. `+ theme_bw()`).
  - You can amend every aspect of a theme through `theme()` (e.g. legend position and appearance, background grid, margins, text formatting etc.)
- Coloring: there are infinite ways to customize e.g. colors. Do that through `scale_color_SOMETHING` . Some examples:
  - `scale_color_manual` to manually choose colors for every value of grouping variable. See also `scale_color_discrete` .
  - `scale_color_continuous` to specify a gradient for a continuous variable. See also `scale_color_gradient` .
  - `scale_color_hue` for evenly spaced colors for discrete variable.
  - You can do the same with every grouping variable: fill (e.g. `scale_fill_manual`), linetype (e.g. `scale_linetype_discrete`), alpha etc.

Go to colorbrewer.org to choose the coolest palette.

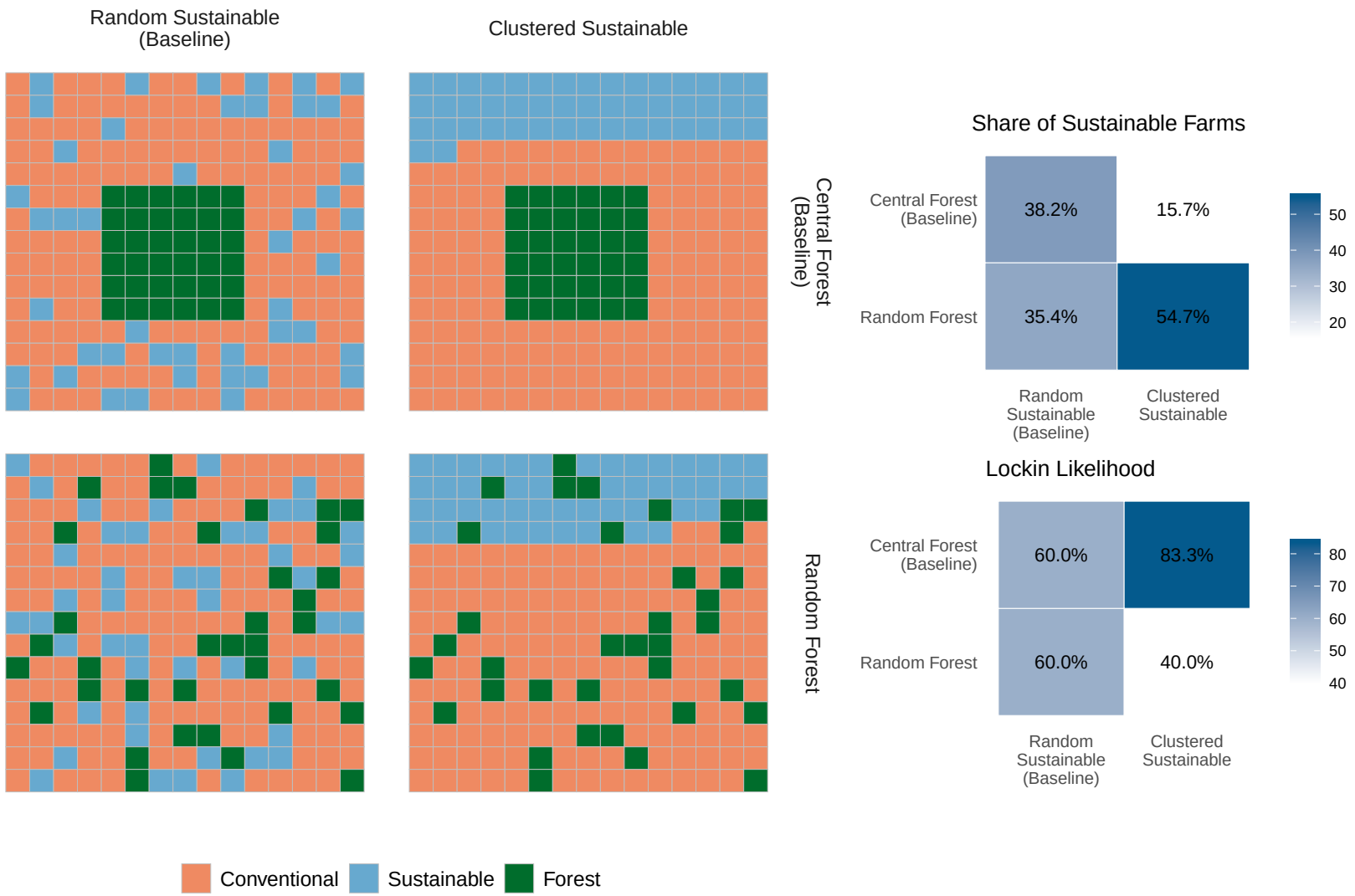
Advanced Charts I

Use `geom_bar` or `geom_hist` to produce bar charts and histograms.



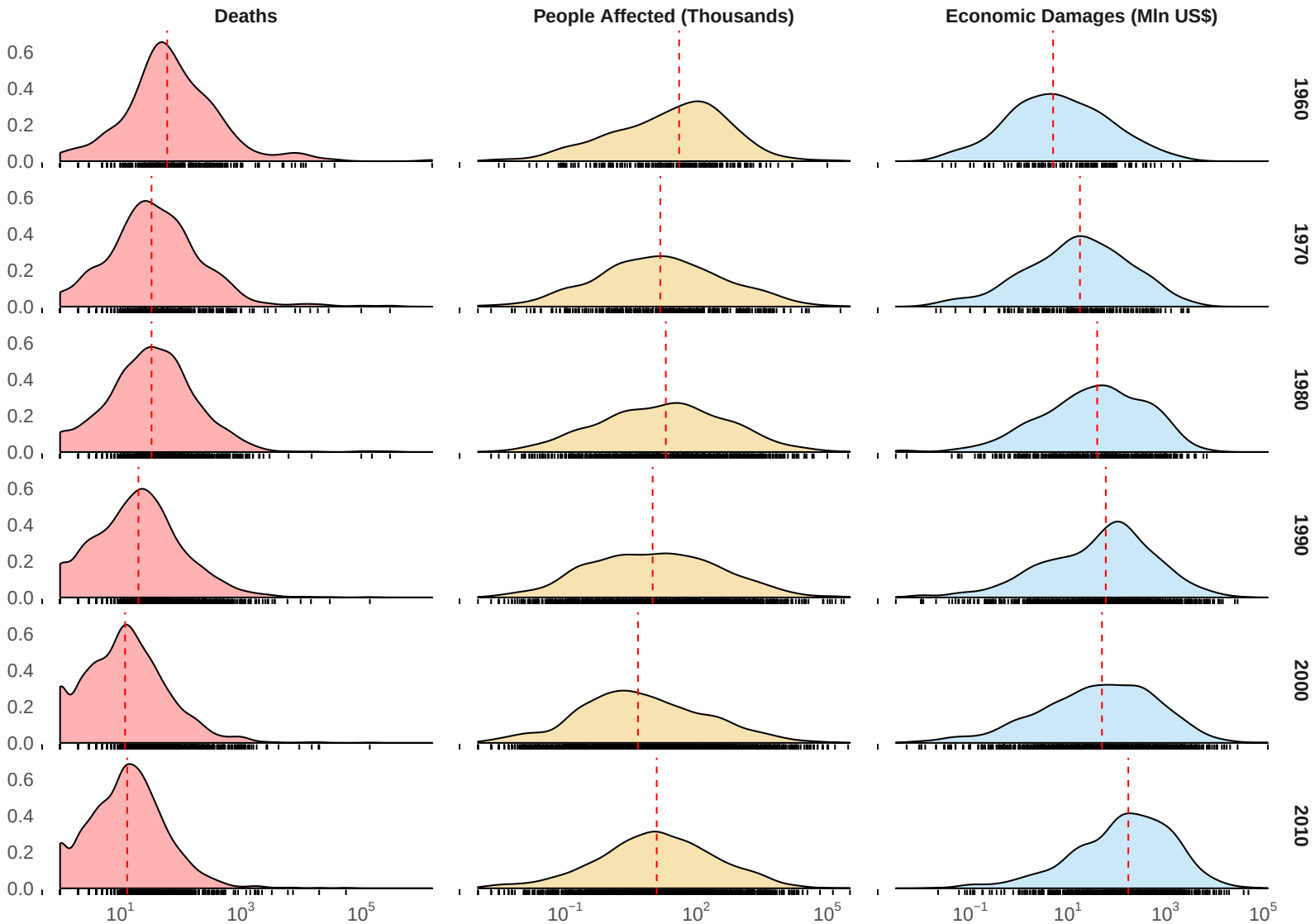
Advanced Charts II

Use `geom_tile` or `geom_raster()` for raster data and/or heatmaps.



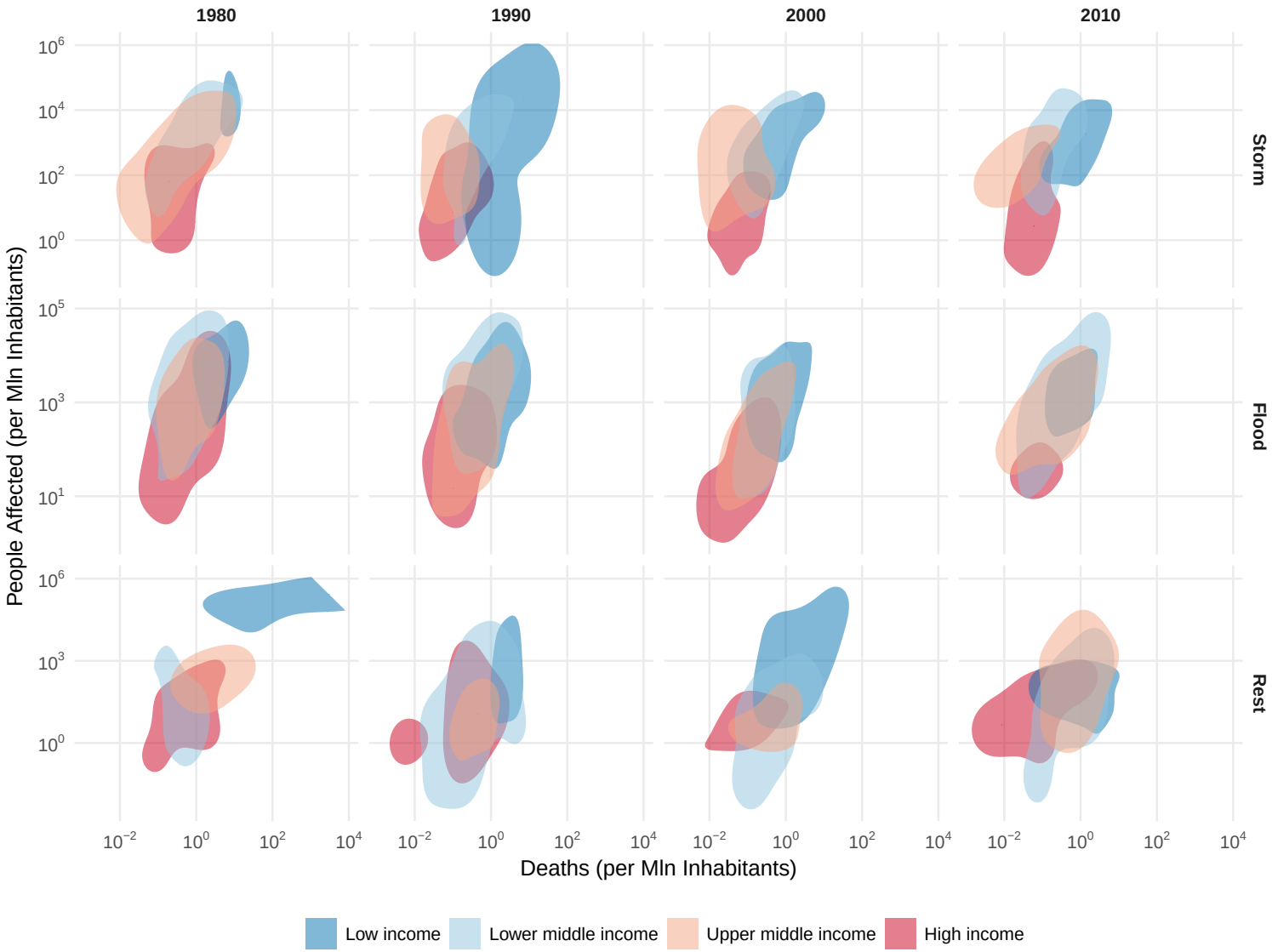
Advanced Charts III

Use `geom_density` to produce kernel smoothed density estimates.



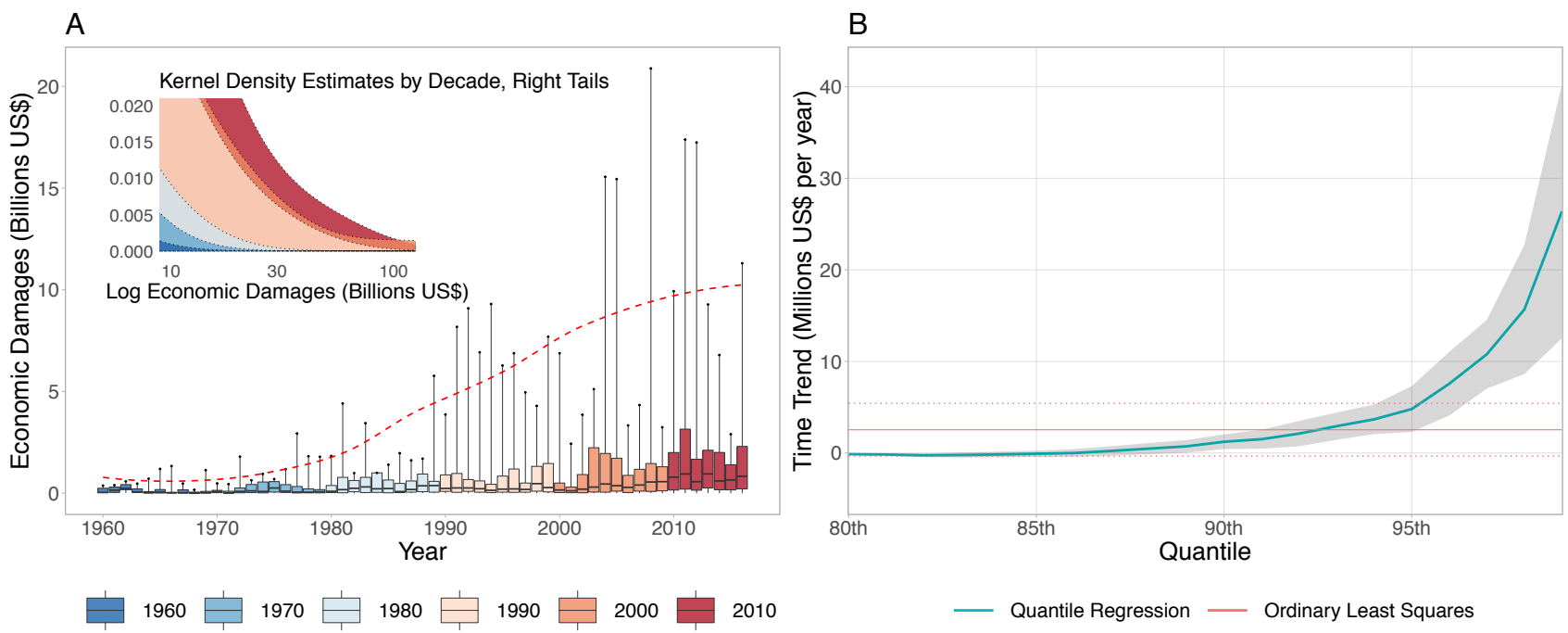
Advanced Charts IV

Use `geom_density2d` to do that in two dimensions.



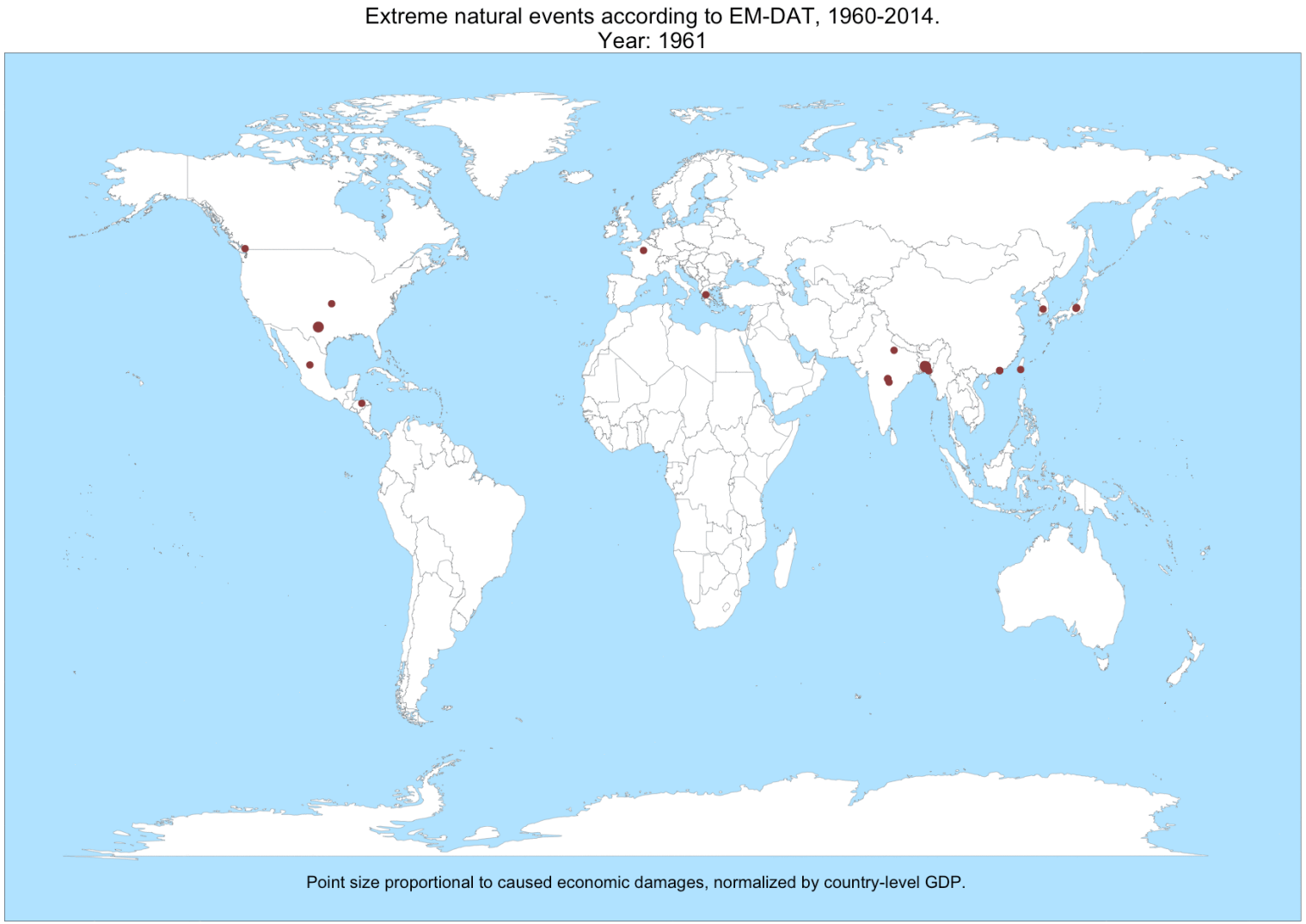
Advanced Charts V

Combine them! For instance, combine `geom_boxplot` (for boxplots), colour them by decade, add a reference line through `geom_smooth`, estimate kernel density with `geom_density`, add lines with `geom_line` and draw confidence bands with `geom_ribbon` in one, admittedly nice, single graph.



Advanced Charts VI

Use the `gganimate` package to animate your graphs (see the guide). Use the `ggmap` package to plot objects on a geographical map. For more complex maps, check out the `sf` package (here is an introduction).



Reporting: rmarkdown

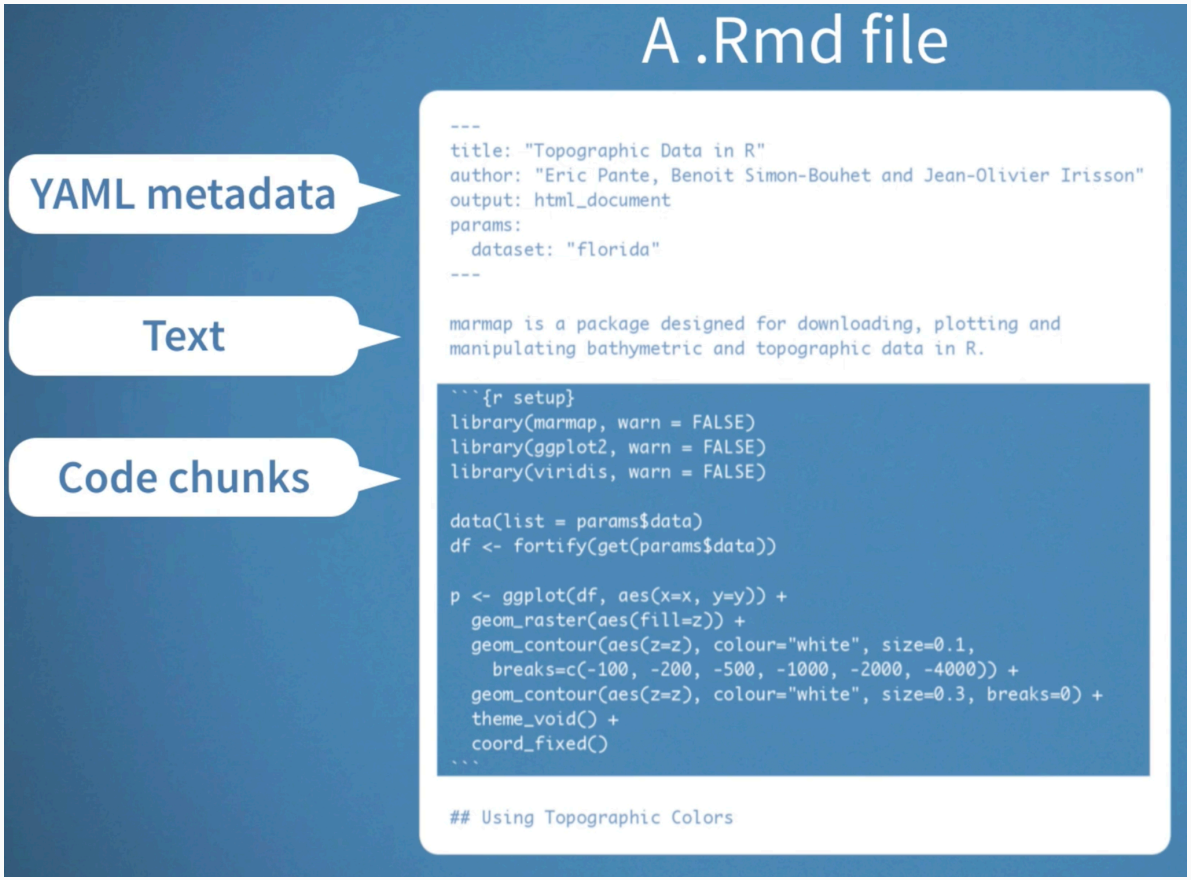
Rmarkdown

Rmarkdown is a format for writing reproducible, dynamic reports with R.

You can use it to embed R code and results into slideshows, pdfs, html documents, Word files and more.

Technically, it is just a plain text file with extension `.Rmd`, which you visualize on your RStudio, composed by three parts:

- YAML metadata: brief instructions on the structure of the report.
- Markdown Text: the text you want in your report, formatted using **markdown** language.
- Code Chucks: pieces of code you use to generate your results.



Getting Started

Markdown *per se* is simply a set of markup annotations for plain text files. Through very simple annotations, you can render your plain text into an elegant and well formatted text file.

With **Rmarkdown** you can combine markdown text with pieces of R code (or other languages, e.g. python) to produce nice reports.

- Install it through `install.packages("rmarkdown")`.
- See the attached file rmarkdown.rmd for a simple tutorial on markdown syntax and code chunks management.
- Check out Rmarkdown guide for advanced tasks.
- Check out GitHub Markdown cheatsheet for further info on markdown syntax.
- And you are good to go!