A tutorial for Plotting in R

Version 1.0.0

Konstantinos I. Bougioukas

07/10/2021

Contents

Preface

In this tutorial we will learn about data visualization in R. The lessons we will cover will enable us to shift from simply showing data to storytelling with data. Being able to tell stories with data is a skill that's becoming ever more important in our world of increasing data and desire for data-driven decision making. An effective data visualization can mean the difference between success and failure when it comes to communicating the findings of our research.

The `ggplot2` package is generally the preferred tool of choice for constructing data visualisations in R. The main reason for this is because of its grounding in the grammar of graphics, the idea that any plot can be expressed from the same set of components: a **data** set, a **coordinate system**, and a set of **geoms**–the visual representation of data points. To display values, ggplot2 maps variables in the data to visual properties of the geom (aesthetics) like size, color, and x and y locations. Additionally, the key to understanding ggplot2 is thinking about a figure in **layers**.

There are two functions in ggplot2: the first is `qplot()` (which stands for "quick plot") and the other is `ggplot()`. In this tutorial we will cover `ggplot()`.

There are many extension packages for `ggplot2`, which make it easy to produce specialized types of graphs, such as survival plots, geographic maps and ROC curves. With `ggplot2` (and its extensions), users can produce elegant, professional-looking visualizations that communicate results powerfully to the desired audience. However, `ggplot2` runs into some limitations regarding user interactivity. This becomes increasingly problematic when creating interactive documents using R Markdown or dashboard apps in R Shiny, where interactivity is a crucial component of communicating information effectively.

Fortunately, the `plotly` package significantly enhances the design of interactive charts in R, allowing users to hover over data points, zoom into specific areas, pan back and forth through time, and much more.

Objectives of the lessons

- To be able to use `ggplot2` to generate publication quality graphics.
- To understand the basic grammar of graphics, including the aesthetics and geometry layers, adding statistics, transforming scales, and coloring or panelling by groups.
- To understand how to save plots in a variety of formats.
- To be able to find extensions for `ggplot2` to produce custom graphics. For example, animated graphs with the `gganimate` package which adds support for declaring animations using an API familiar to users of `ggplot2`, and linked data visualizations with the `ggiraph`.
- To be able to create interactive charts with `plotly` , `highcharter` and `c3` packages.

1 Preparation

1.1 Packages to download

We should first install the following packages to execute the full tutorial, if we don't have them installed on our computer:

- {here},
- {tidyverse},
- {tidycovid19},
- {plotly},
- {highcharter},
- {c3},
- {gghighlight},
- {ggforce},
- {paletteer},
- {scales},
- {ggrepel},
- {ggsci},
- {ggtext},
- {ggfx},
- {RColorBrewer},
- {patchwork},
- {gganimate},
- {gghighlight},
- {scatterD3},
- {ggpp},
- {ggdist},
- {ggprism},
- {ggnewscale},
- {albersusa}, from "https://git.rud.is/hrbrmstr/albersusa.git"
- {ggiraph},
- {htmlwidgets},
- {webshot}

1.2 Dataset description

In this tutorial we will work on Covid-19 data. The `download_merged_data()` from `tidycovid19` package downloads all data sources and creates a merged country-day panel.

```
library(tidyverse)

#library(remotes)
#remotes::install_github("joachim-gassen/tidycovid19")
# https://github.com/joachim-gassen/tidycovid19
library(tidycovid19)
```

```
# get the data
covid_data <- download_merged_data(cached = TRUE)
```

Following we can see the definitions for each variable included in the merged country-day data frame:

Table 1: Dataset variables

var_name	var_def
iso3c	ISO3c country code as defined by ISO 3166-1 alpha-3
country	Country name
date	Calendar date
confirmed	Confirmed Covid-19 cases as reported by JHU CSSE (accumulated)
deaths	Covid-19-related deaths as reported by JHU CSSE (accumulated)
recovered	Covid-19 recoveries as reported by JHU CSSE (accumulated)

var_name	var_def
ecdc_cases	Covid-19 cases as reported by ECDC (accumulated, weekly post 2020-12-14)
ecdc_deaths	Covid-19-related deaths as reported by ECDC (accumulated, weekly post 2020-12-14)
total_tests	Accumulated test counts as reported by Our World in Data
tests_units	Definition of what constitutes a 'test'
positive_rate	The share of COVID-19 tests that are positive, given as a rolling 7-day average
hosp_patients	Number of COVID-19 patients in hospital on a given day
icu_patients	Number of COVID-19 patients in intensive care units (ICUs) on a given day
total_vaccinations	Total number of COVID-19 vaccination doses administered
soc_dist	Number of social distancing measures reported up to date by ACAPS, net of lifted restrictions
mov_rest	Number of movement restrictions reported up to date by ACAPS, net of lifted restrictions
pub_health	Number of public health measures reported up to date by ACAPS, net of lifted restrictions
gov_soc_econ	Number of social and economic measures reported up to date by ACAPS, net of lifted restrictions
lockdown	Number of lockdown measures reported up to date by ACAPS, net of lifted restrictions
apple_mtr_driving	Apple Maps usage for driving directions, as percentage*100 relative to the baseline of Jan 13, 2020
apple_mtr_walking	Apple Maps usage for walking directions, as percentage*100 relative to the baseline of Jan 13, 2020

var_name	var_def
apple_mtr_transit	Apple Maps usage for public transit directions, as percentage*100 relative to the baseline of Jan 13, 2020
gcmr_retail_recreation	Google Community Mobility Reports data for the frequency that people visit retail and recreation places expressed as a percentage*100 change relative to the baseline period Jan 3 - Feb 6, 2020
gcmr_grocery_pharmacy	Google Community Mobility Reports data for the frequency that people visit grocery stores and pharmacies expressed as a percentage*100 change relative to the baseline period Jan 3 - Feb 6, 2020
gcmr_parks	Google Community Mobility Reports data for the frequency that people visit parks expressed as a percentage*100 change relative to the baseline period Jan 3 - Feb 6, 2020
gcmr_transit_stations	Google Community Mobility Reports data for the frequency that people visit transit stations expressed as a percentage*100 change relative to the baseline period Jan 3 - Feb 6, 2020
gcmr_workplaces	Google Community Mobility Reports data for the frequency that people visit workplaces expressed as a percentage*100 change relative to the baseline period Jan 3 - Feb 6, 2020
gcmr_residential	Google Community Mobility Reports data for the frequency that people visit residential places expressed as a percentage*100 change relative to the baseline period Jan 3 - Feb 6, 2020
gtrends_score	Google search volume for the term 'coronavirus', relative across time with the country maximum scaled to 100
gtrends_country_score	Country-level Google search volume for the term 'coronavirus' over a period starting Jan 1, 2020, relative across countries with the country having the highest search volume scaled to 100 (time-stable)
region	Country region as classified by the World Bank (time-stable)
income	Country income group as classified by the World Bank (time-stable)

var_name	var_def
population	Country population as reported by the World Bank (original identifier 'SP.POP.TOTL', time-stable)
land_area_skm	Country land mass in square kilometers as reported by the World Bank (original identifier 'AG.LND.TOTL.K2', time-stable)
pop_density	Country population density as reported by the World Bank (original identifier 'EN.POP.DNST', time-stable)
pop_largest_city	Population in the largest metropolian area of the country as reported by the World Bank (original identifier 'EN.URB.LCTY', time-stable)
life_expectancy	Average life expectancy at birth of country citizens in years as reported by the World Bank (original identifier 'SP.DYN.LE00.IN', time-stable)
gdp_capita	Country gross domestic product per capita, measured in 2010 US-$ as reported by the World Bank (original identifier 'NY.GDP.PCAP.KD', time-stable)
timestamp	Date and time where data has been collected from authoritative sources

Let's have a look at the types of variables:

```
glimpse(covid_data)
```

```
## Rows: 131,681
## Columns: 40
## $ iso3c                 <chr> "ABW", "ABW"~
## $ country               <chr> "Aruba", "Ar~
## $ date                  <date> 2020-03-13,~
## $ confirmed             <dbl> NA, NA, NA, ~
## $ deaths                <dbl> NA, NA, NA, ~
## $ recovered             <dbl> NA, NA, NA, ~
## $ ecdc_cases            <dbl> 2, 2, 2, 2, ~
```

```
## $ ecdc_deaths            <dbl> 0, 0, 0, 0, ~
## $ total_tests            <dbl> NA, NA, NA, ~
## $ tests_units            <chr> NA, NA, NA, ~
## $ positive_rate          <dbl> NA, NA, NA, ~
## $ hosp_patients          <dbl> NA, NA, NA, ~
## $ icu_patients           <dbl> NA, NA, NA, ~
## $ total_vaccinations     <dbl> NA, NA, NA, ~
## $ soc_dist               <dbl> NA, NA, NA, ~
## $ mov_rest               <dbl> NA, NA, NA, ~
## $ pub_health             <dbl> NA, NA, NA, ~
## $ gov_soc_econ           <dbl> NA, NA, NA, ~
## $ lockdown               <dbl> NA, NA, NA, ~
## $ apple_mtr_driving      <dbl> NA, NA, NA, ~
## $ apple_mtr_walking      <dbl> NA, NA, NA, ~
## $ apple_mtr_transit      <dbl> NA, NA, NA, ~
## $ gcmr_place_id          <chr> "ChIJ23da4s8~
## $ gcmr_retail_recreation <dbl> -10, -23, -2~
## $ gcmr_grocery_pharmacy  <dbl> 40, 15, -13,~
## $ gcmr_parks             <dbl> -4, -7, -6, ~
## $ gcmr_transit_stations  <dbl> -5, -19, -18~
## $ gcmr_workplaces        <dbl> 3, -3, -5, -~
## $ gcmr_residential       <dbl> 1, 7, 6, 12,~
## $ gtrends_score          <dbl> NA, NA, NA, ~
## $ gtrends_country_score  <int> NA, NA, NA, ~
## $ region                 <chr> "Latin Ameri~
## $ income                 <chr> "High income~
## $ population             <dbl> 106766, 1067~
## $ land_area_skm          <dbl> 180, 180, 18~
## $ pop_density            <dbl> 593.1444, 59~
## $ pop_largest_city       <dbl> NA, NA, NA, ~
## $ life_expectancy        <dbl> 76.293, 76.2~
## $ gdp_capita             <dbl> 26631.47, 26~
## $ timestamp              <dttm> 2021-10-07 ~
```

The data frame contains more than 131681 rows and 40 variables. There are 32 numeric variables, 6 variables of character type, and two variables with dates (one of Date type and the other of POSIXct type).

NOTE
Another informative function to inspect our dataset is `skim()` in the `skimr` package:

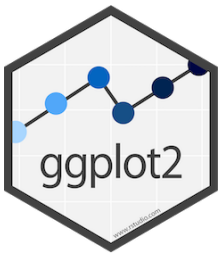
```
skimr::skim(covid_data)
```

In this example, we investigate graphically the association between a country's wealth and COVID-19 cases. This research question may provide valuable information on whether interventions have been successful, and whether people in countries with fragile health systems are at increased risk.

What other variables are associated to both wealth and COVID-19 cases? For example, wealthier countries may have more resources to test for the virus and a lack of reported cases in developing countries could indicate a scarcity of testing. Therefore, when assessing wealth and COVID-19 cases, we should be cautious when comparing countries with different testing rates. In addition, evidence suggests age is an important factor in COVID-19 infection and mortality. Therefore, we should be careful when comparing countries with higher proportions of older inhabitants to other countries. Using diagrams to depict multivariable associations may be helpful.

1.3 Data preparation for the plots

```
dat <- covid_data %>%
  filter(date == "2021-06-12", population > 1000000) %>%
  mutate(cases_per_100k = confirmed / population * 100000,
         tests_per_capita = total_tests / population)
```



2 Step-by-step anatomy of a ggplot

Because `ggplot2` is a core tidyverse package, it is installed for us when we installed the `tidyverse` meta-package.

> **ggplot2** is a system for declaratively creating graphics, based on The Grammar of Graphics. We provide the data, tell ggplot2 how to map variables to aesthetics, what graphical primitives to use, and it takes care of the details.

Consequently, a ggplot is built up from a few basic elements (Figure 1):

1. **Data**: The raw data that we want to plot.
2. **Geometries** `geom_`: The geometric shapes that will represent the data.
3. **Aesthetics** `aes()`: Aesthetics of the geometric and statistical objects, such as color, size, shape, transparency and position.
4. **Scales** `scale_`: Maps between the data and the aesthetic dimensions, such as data range to plot width or factor values to colors.
5. **Statistical transformations** `stat_`: Statistical summaries of the data, such as quantiles, fitted curves and sums.
6. **Coordinate system** `coord_`: The transformation used for mapping data coordinates into the plane of the data rectangle.
7. **Facets** `facet_`: The arrangement of the data into a grid of plots.
8. **Visual themes** `theme()`: The overall visual defaults of a plot, such as background, grids, axes, default typeface, sizes and colors.

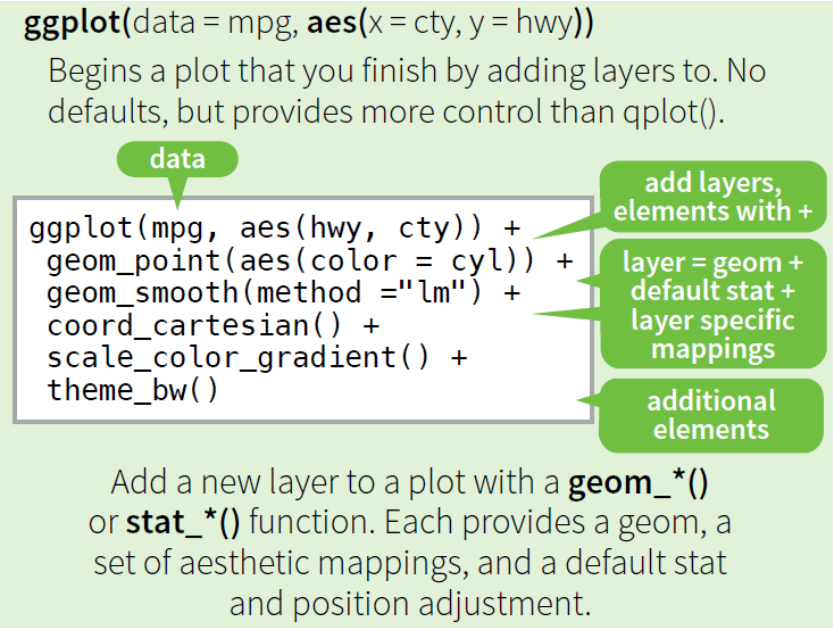


Figure 1: ggplot2 basic syntax

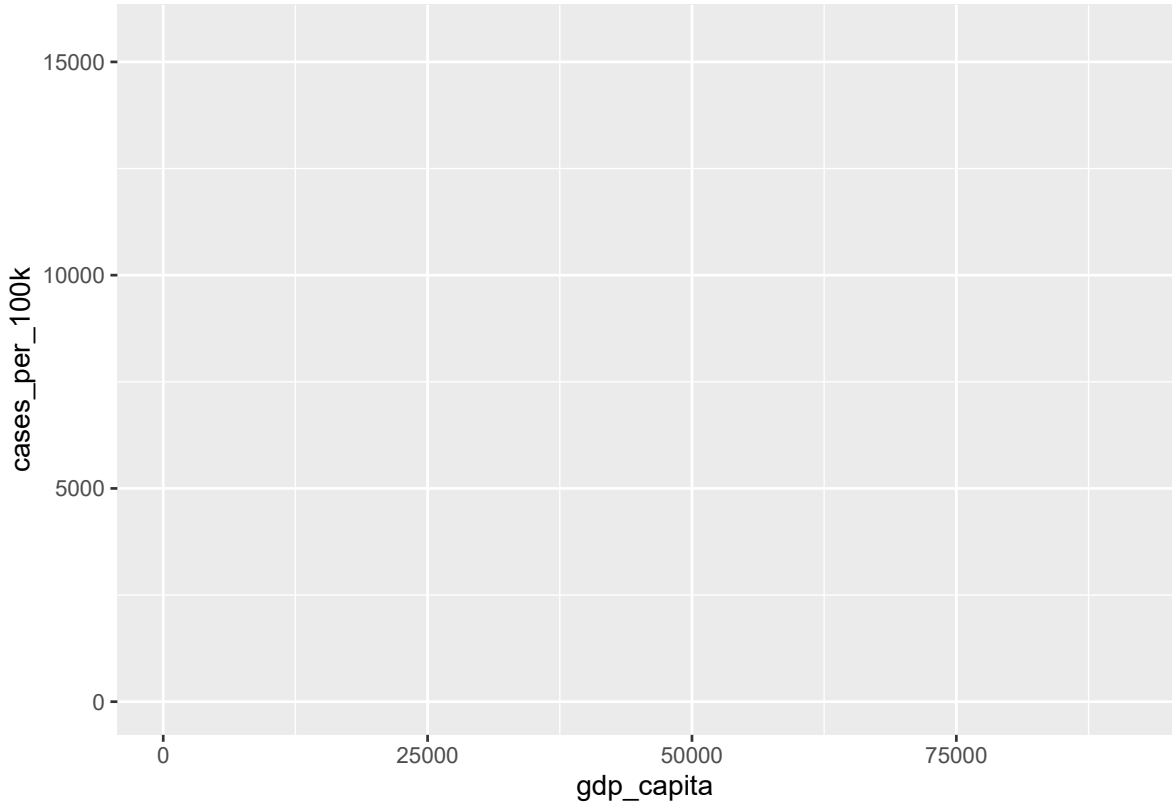
2.1 Start with a default blank ggplot

We always start to define a plotting element by calling:

```
ggplot(data = df, mapping = aes(x = variable1, y = variable2)).
```

For example, here, we map the variable `gdp_capita` to the x position and the variable `cases_per_100K` to the y position. Later, we will also map variables to all kind of other aesthetics such as color, size, and shape.

```
p <- ggplot(data = dat,
            mapping = aes(x = gdp_capita, y = cases_per_100k))
p
```



The `ggplot()` function has **two basic arguments**. The argument `data` defines which dataset to use for the plot; this allows us to refer to `gdp_capita` and `gdp_capita` without repeating from which dataset they come. The argument `mapping` defines which variables are mapped onto which aesthetics. `Aesthetics` are the visual properties of the elements that make up the plot. For example, `aes(x = gdp_capita, y = cases_per_100k)` means that the x and y aesthetics of all elements will be mapped onto the variables `gdp_capita` and `gdp_capita`, respectively.

As we can see, only a panel is created when running this. Why? This is because we have only told `ggplot` what dataset to use and what columns should be used for `x` and `y` axis. We haven't explicitly asked it to draw anything else —we still need to provide a geometry!

`ggplot2` allows us to store the current ggobject in an object of our choice by assigning it to a name, in our case called p. We can add other elements on top of the blank layer using the + operator.

NOTE We don't usually have to spell out data and mapping.

Next, suppose that we want to create a scaterplot. Scatterplots can be useful for showing the relationship between at least two variables, because they allow us to encode data simultaneously on a horizontal x-axis and vertical y-axis to see whether and what association exists.

2.2 Add a geometry

Geoms are the geometric shapes that make up ggplot2 visualisations (Figure 2). Each is called with a function that begins with geom_* and ends with the name of the geom (e.g., point or line). Each geom has a number of aesthetics that define its visual properties.

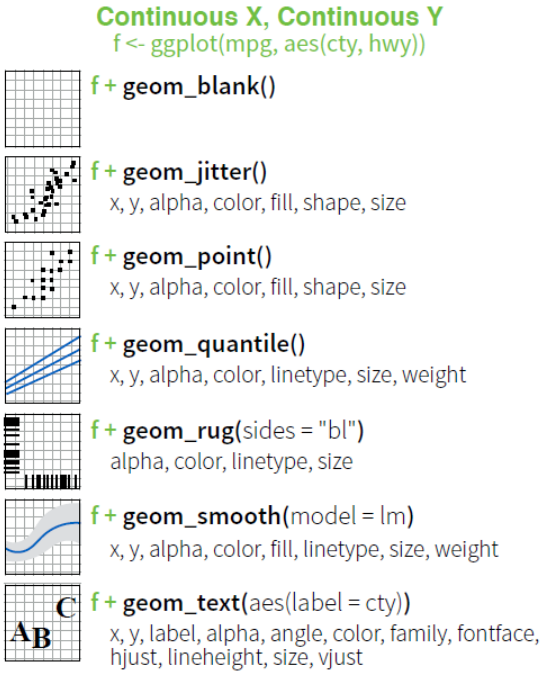
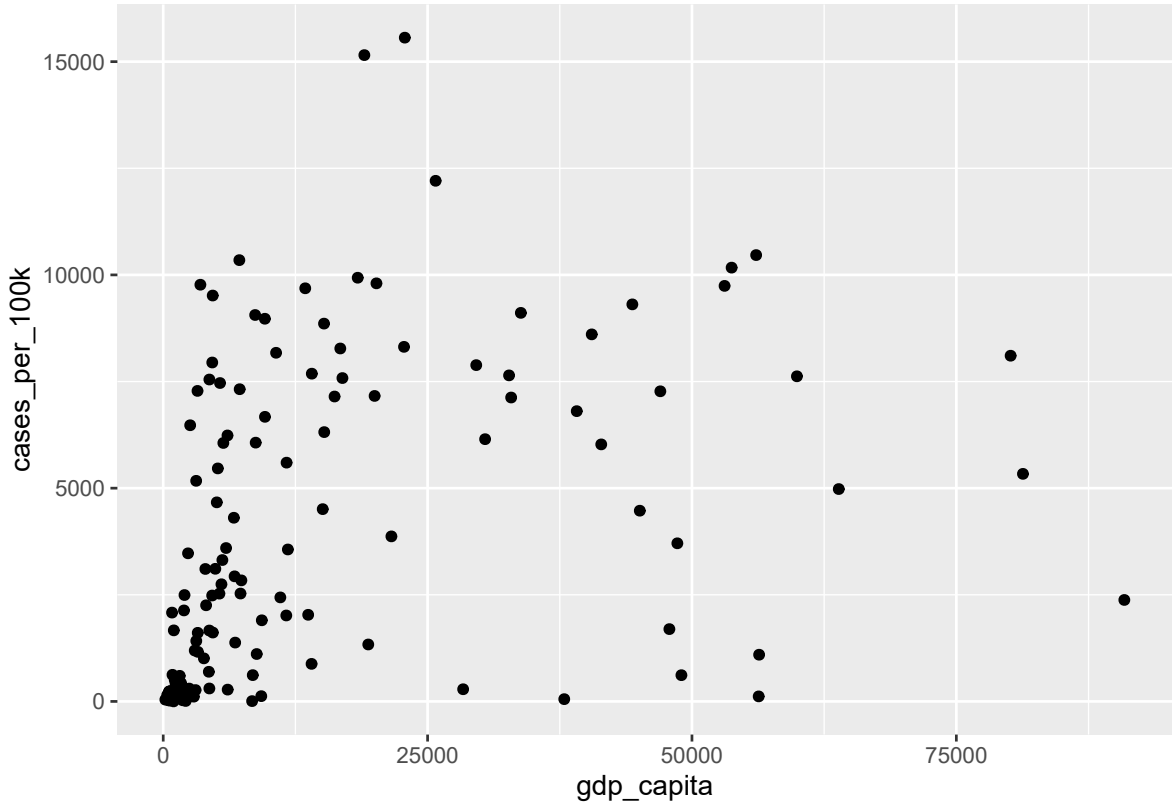


Figure 2: Geometries for two continuous variables

Let's tell `ggplot2` which style we want to use. For example, we will built a scatter plot by adding points using a geom layer called **`geom_point`**. In this case, `geom_point()` inherits the `x` and `y` aesthetics from the `ggplot()` function.

```
p + geom_point()
```

```
## Warning: Removed 3 rows containing missing values
## (geom_point).
```



2.3 Add Aesthtics to Geometries: `aes()`

Each `geom` comes with its own properties (called arguments) and the same argument may result in a different change depending on the geom we are using.

`Aesthetic attributes` describe every aspect of a given graphical element and refer to which variable is mapped onto it. We can map our data to anything that our geom supports.

To add additional variables to a plot, we can use aesthetics like color, shape, and size.

NOTE Each argument to aes() maps a **variable** in our data to a specific element in our 'geom'.

2.3.1 `color` aesthetics

Color is an important dimension in human vision and is consequently equally important in the design of a scientific figure. However, color can be either our greatest ally or our worst enemy if not used properly. If we decide to use color, we should consider which colors to use and where to use them.

Color palettes (or colormaps) are classified into three main categories in `ggplot2`:

1. **Sequential:** one variation of a unique color, used for **quantitative** data varying from dark to light (Figure 3). These scales may exist in discrete or continuous forms or both.

Figure 3: Example sequential color scales

2. **Diverging:** variation from one color to another, used to highlight **deviation from a median value** (Figure 4) A diverging color scale creates a gradient between three different colors, allowing us to easily identify low, middle, and high values within our data.

Figure 4: Example diverging color scales

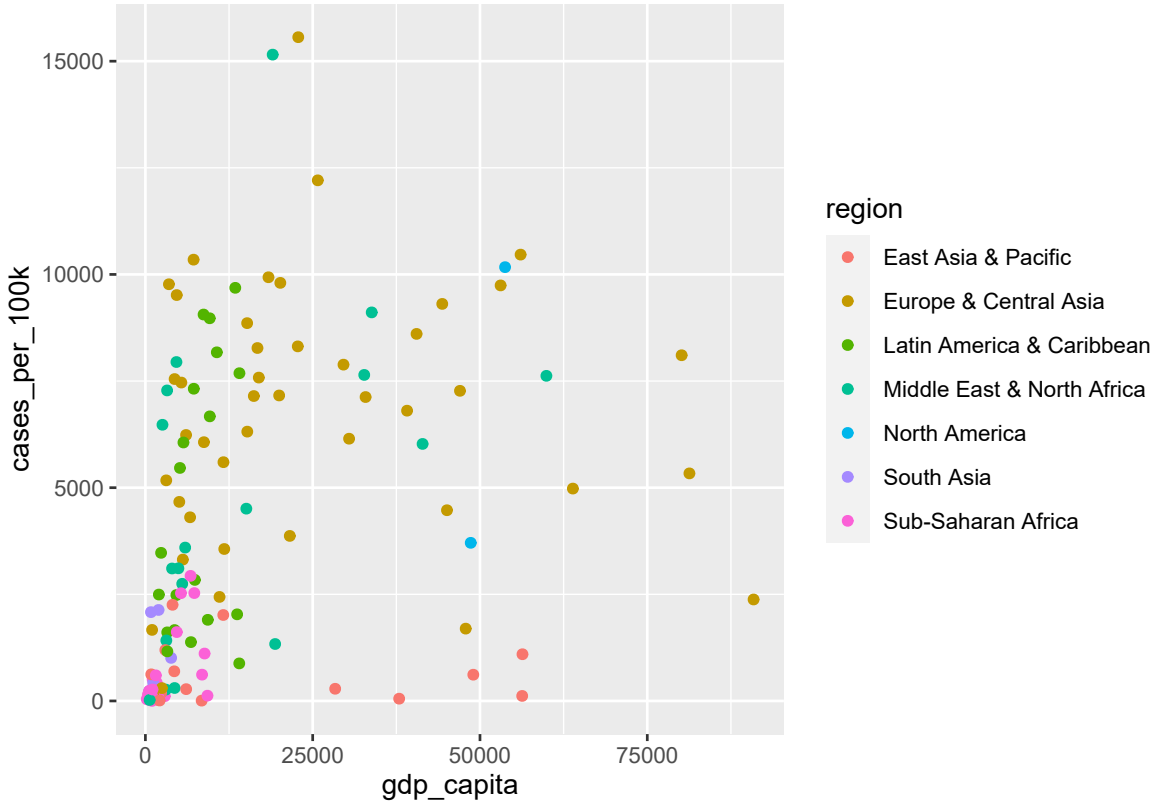
3. **Qualitative:** a discrete set of distinct colors with no implied order, used mainly for discrete or **categorical** data (Figure 5).

Figure 5: Example qualitative color scales

Now, suppose we want to group the points according to the categorical variable `region` using different colors, as follows:

```
p + geom_point(aes(color = region))
```

```
## Warning: Removed 3 rows containing missing values
## (geom_point).
```



Here, we added inside the `aes()` of `geom_point` the color aesthetic. The data of the categorical variable `region` mapped to color aesthetic of `geom_point`. Obviously, the qualitative scale was applied automatically by `ggplot2`. In addition, ggplot automatically created a legend to show the correspondence between the regions and colors.

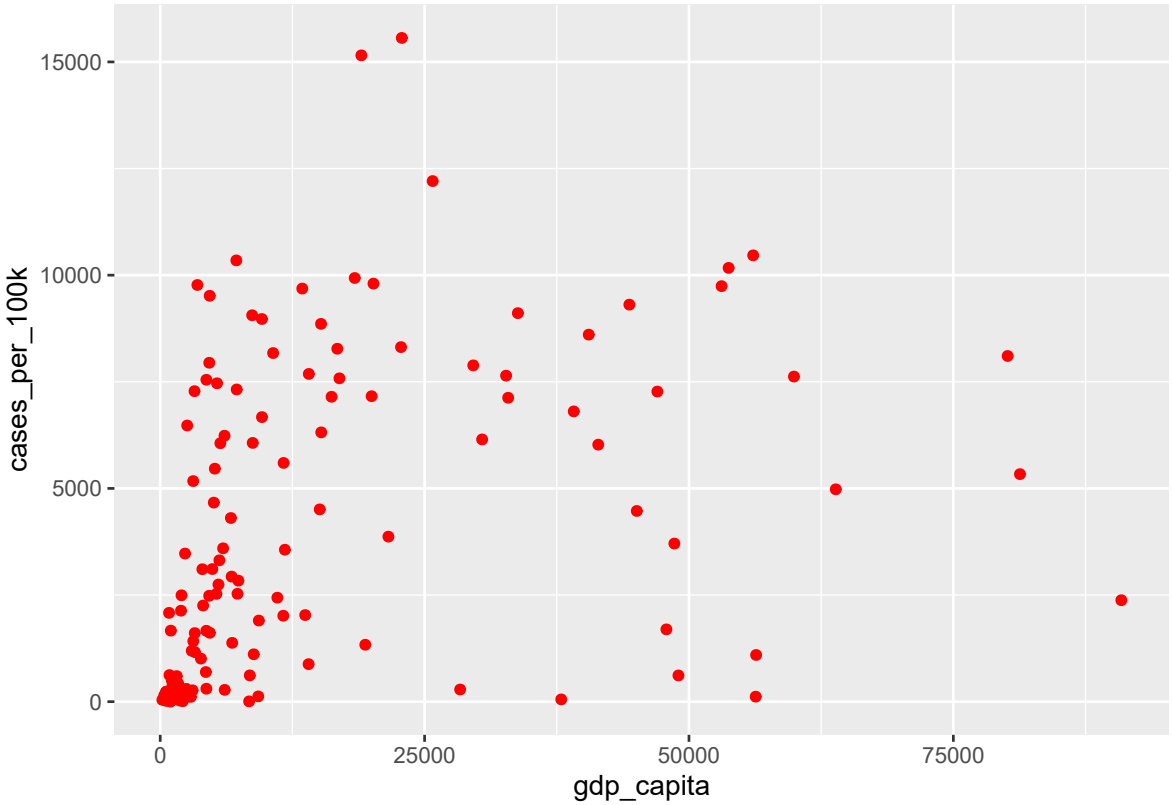
To find all aesthetics that apply to the `geom_point()` function, run `?geom_point`. We can find a useful list of all aesthetic specifications there.

> **NOTE** The grouping variable must be categorical—in other words, a factor or character vector. If it is stored as a vector of numeric values, it should be converted to a factor before it is used as a grouping variable.

It is very important to understand the difference between including `ggplot` arguments **inside** or **outside** of the `aes()` function. For example, let's see the difference if we add the color outside the `aes()` function:

```
p + geom_point(color = "red")
```

```
## Warning: Removed 3 rows containing missing values
## (geom_point).
```



In this case, we set an aesthetic to a **fixed value** (red) in the individual layer `geom_point` outside of `aes()`. In fact, we set the red color for the entire graph (constant aesthetics), so it changed the color of the plot globally.

> **NOTE** ggplot2 understands both color and colour as well as the short version col.

In R, colors can be specified either by name (e.g., `"blue"`) or as a hexadecimal RGB triplet (such as `#0072B2`). The hexadecimal system defines each color (red, green, or blue) in terms of two values that range from 0–9, then from A–F, giving a total of 16 different values for each character.

We can also use palettes of colors such as `viridis` palette which we will use in the next graphs. In following Table we present a color blind-friendly palette.

In R, the hex color code goes in quotes and is preceded by a #. For example:

Name	Hex code
orange	#E69F00
sky blue	#56B4E9
bluish green	#009E73
yellow	#F0E442
blue	#0072B2
vermilion	#D55E00
reddish purple	#CC79A7
black	#000000

Although the code for hexadecimal may not be intuitive, it is easy to look up online exactly what the code is for any color we want to use (just do an internet search for something like "RGB to Hex color"). The main advantage of the Hex system over RGB is that it is very compact to specify whatever color we want.

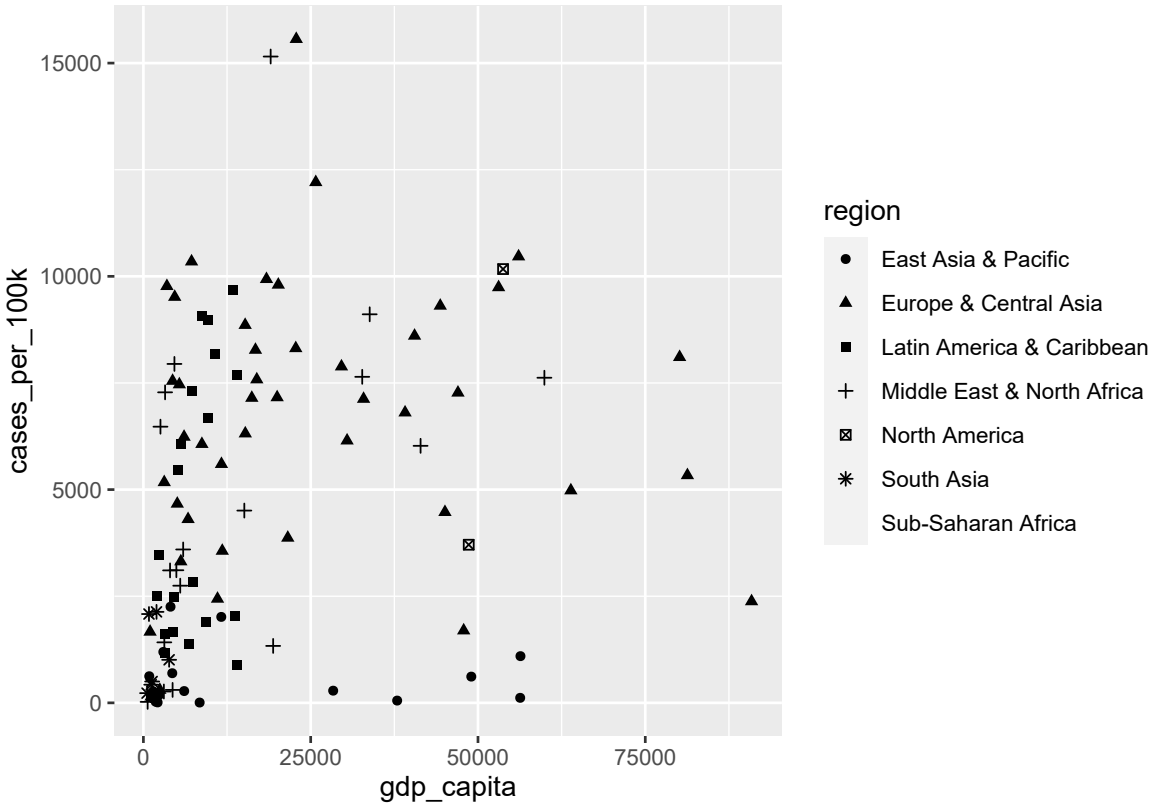
2.3.2 `shape` aesthetics

Alternatively, we can group the points according to the `region` variable using different point shapes, as follows:

```
p + geom_point(aes(shape = region))
```

```
## Warning: The shape palette can deal with a
## maximum of 6 discrete values because
## more than 6 becomes difficult to
## discriminate; you have 7. Consider
## specifying shapes manually if you must
## have them.
```

```
## Warning: Removed 46 rows containing missing values
## (geom_point).
```



The different points shapes symbols commonly used in R are shown in the **Figure 6** below:

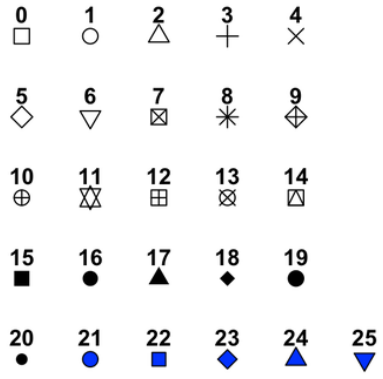


Figure 6: The different points symbols commonly used in R

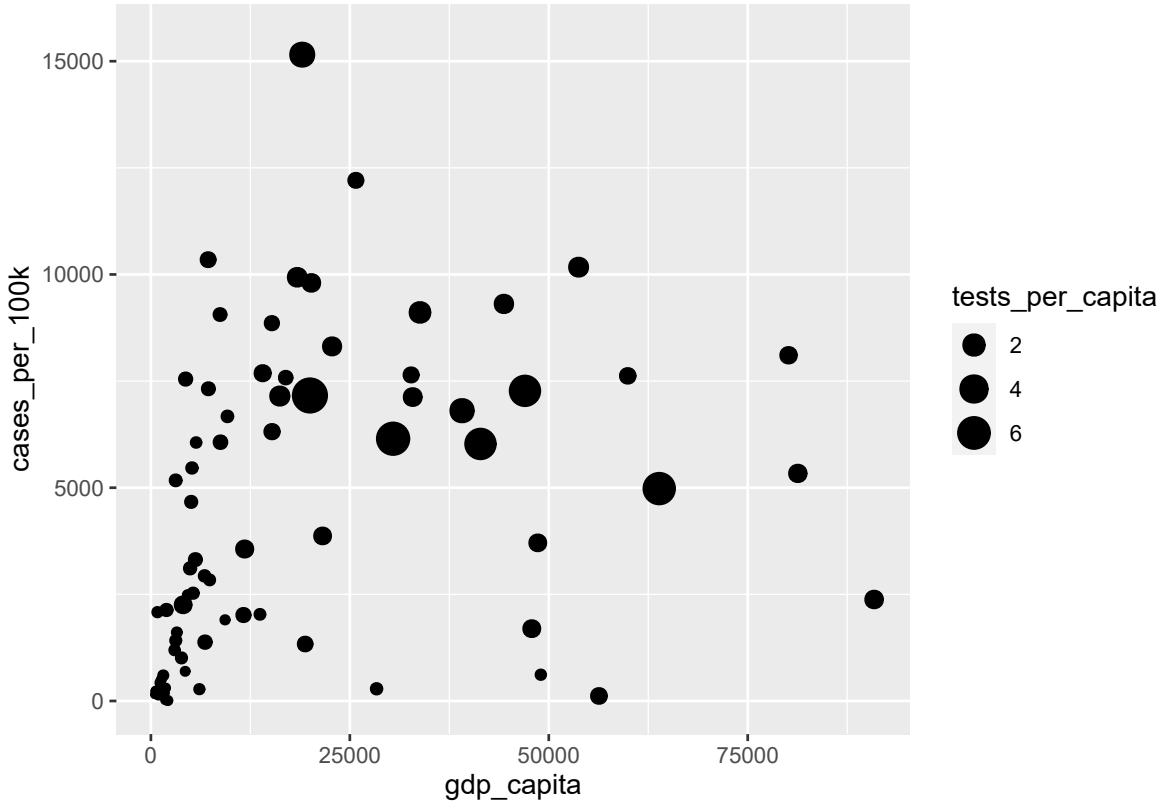
As we can observe in the previous graph, `ggplot2` by default allows only six different point shapes to be displayed. However, we will see how to change this using appropriate `scales`.

2.3.3 `size` aesthetics

Next, we can add a third variable `tests_per_capita` using the size aesthetic:

```
p + geom_point(aes(size = tests_per_capita))
```

```
## Warning: Removed 76 rows containing missing values
## (geom_point).
```



2.4 Add a new geom

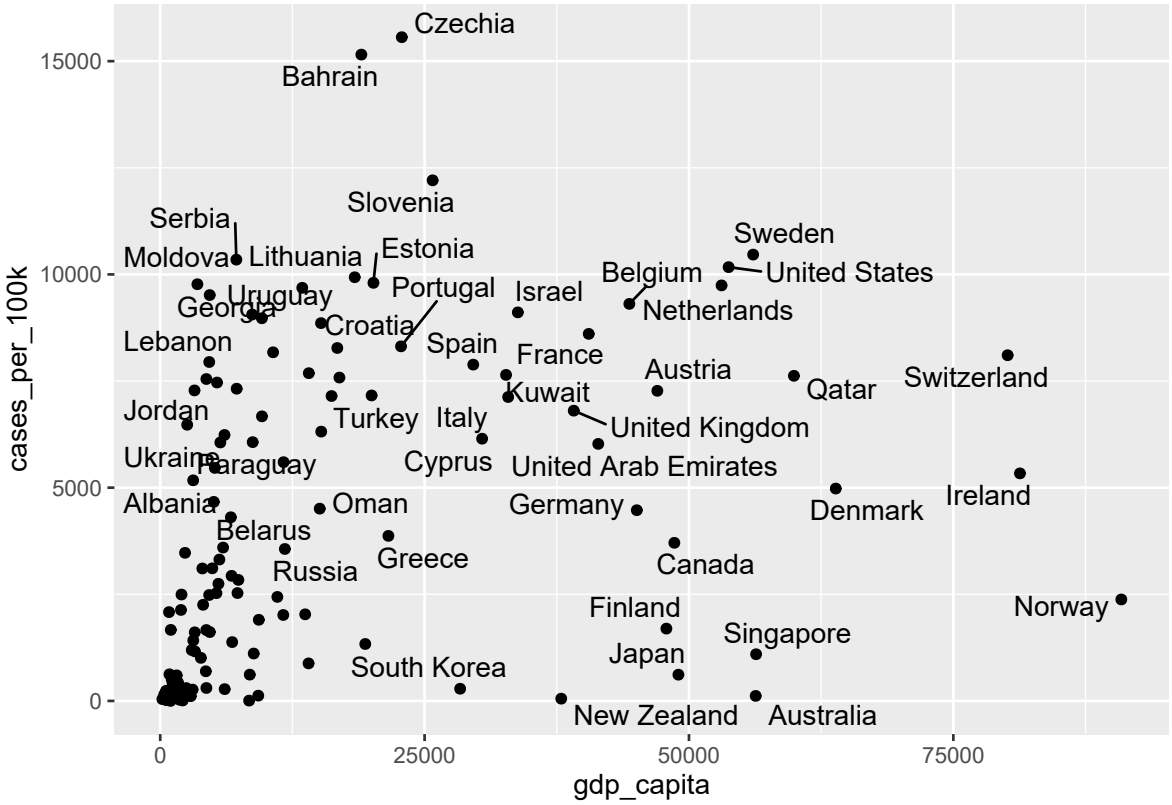
We can add the name of the country for each data point:

```
ggplot(dat, aes(x = gdp_capita, y = cases_per_100k)) +
  geom_point() +
  geom_text_repel(aes(label = country))
```

```
## Warning: Removed 3 rows containing missing values
## (geom_point).
```

```
## Warning: Removed 3 rows containing missing values
## (geom_text_repel).
```

```
## Warning: ggrepel: 105 unlabeled data points
## (too many overlaps). Consider increasing
## max.overlaps
```



2.5 Change the default properties of the plot: `scales`

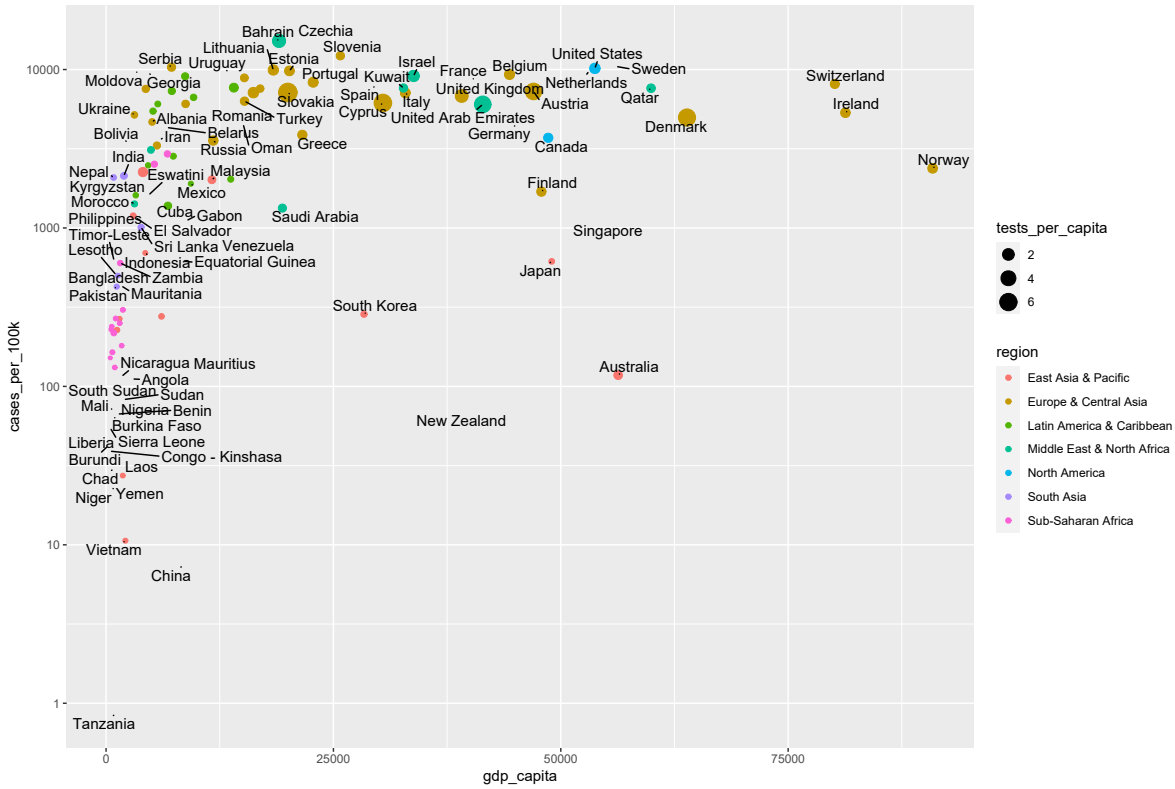
2.5.1 Change the scale of the axis

```
ggplot(dat, aes(x = gdp_capita, y = cases_per_100k)) +
  geom_point(aes(size = tests_per_capita, color = region)) +
  geom_text_repel(aes(label = country),
                  min.segment.length = 0, seed = 42,
                  box.padding = 0.1) +
  scale_y_continuous(trans = "log10") # or scale_y_log10()
```

```
## Warning: Removed 76 rows containing missing values
## (geom_point).
```

```
## Warning: Removed 3 rows containing missing values
## (geom_text_repel).
```

```
## Warning: ggrepel: 63 unlabeled data points
## (too many overlaps). Consider increasing
## max.overlaps
```



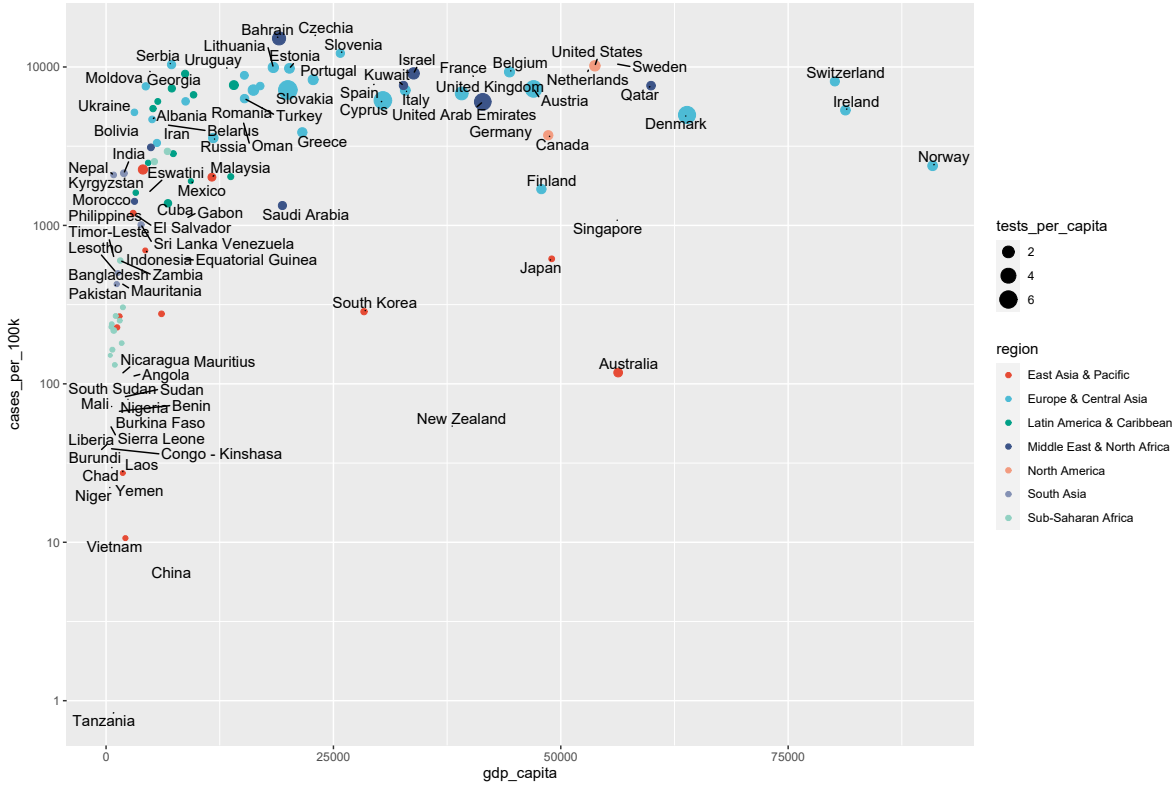
Continuous variable `tests_per_capital` mapped to size and categorical variable `region` mapped to color.

2.5.2 Change the default colors

```
ggplot(dat, aes(x = gdp_capita, y = cases_per_100k)) +
  geom_point(aes(size = tests_per_capita, color = region)) +
  geom_text_repel(aes(label = country),
                  min.segment.length = 0, seed = 42,
                  box.padding = 0.1) +
  scale_y_continuous(trans = "log10") +
  scale_color_npg()
```

```
## Warning: Removed 76 rows containing missing values
## (geom_point).
```

```
## Warning: Removed 3 rows containing missing values
## (geom_text_repel).

## Warning: ggrepel: 63 unlabeled data points
## (too many overlaps). Consider increasing
## max.overlaps
```



NOTE: More for color scales

[HCL-Based Color Scales for ggplot2]

[Scientific Journal and Sci-Fi Themed]

We can also create our own color palettes. For example:

```
scale_fill_mine <- function(...){
  ggplot2::discrete_scale(
    "fill", "mine",
```

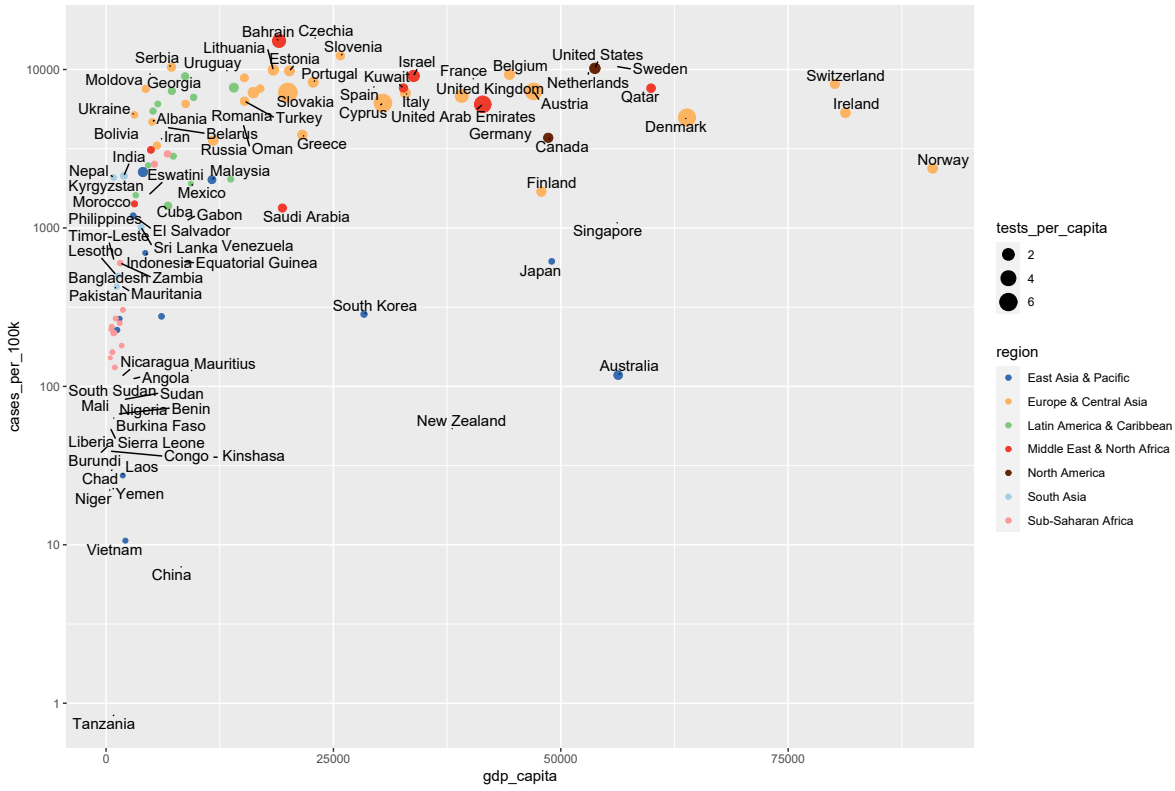
```
    scales::manual_pal(
      values = c(
        "#386cb0","#fdb462","#7fc97f",
        "#ef3b2c","#662506","#a6cee3",
        "#fb9a99","#984ea3","#ffff33", "#000099")),
    ...)
}


scale_color_mine <- function(...){
  ggplot2::discrete_scale(
    "color", "mine",
    scales::manual_pal(
      values = c(
        "#386cb0","#fdb462","#7fc97f",
        "#ef3b2c","#662506","#a6cee3",
        "#fb9a99","#984ea3","#ffff33","#000099")),
    ...)
}
```

```
ggplot(dat, aes(x = gdp_capita, y = cases_per_100k)) +
  geom_point(aes(size = tests_per_capita, color = region)) +
  geom_text_repel(aes(label = country),
                  min.segment.length = 0, seed = 42,
                  box.padding = 0.1) +
  scale_y_continuous(trans = "log10") +
  scale_color_mine()
```

```
## Warning: Removed 76 rows containing missing values
## (geom_point).
```

```
## Warning: Removed 3 rows containing missing values
## (geom_text_repel).
```

```
## Warning: ggrepel: 63 unlabeled data points
## (too many overlaps). Consider increasing
## max.overlaps
```

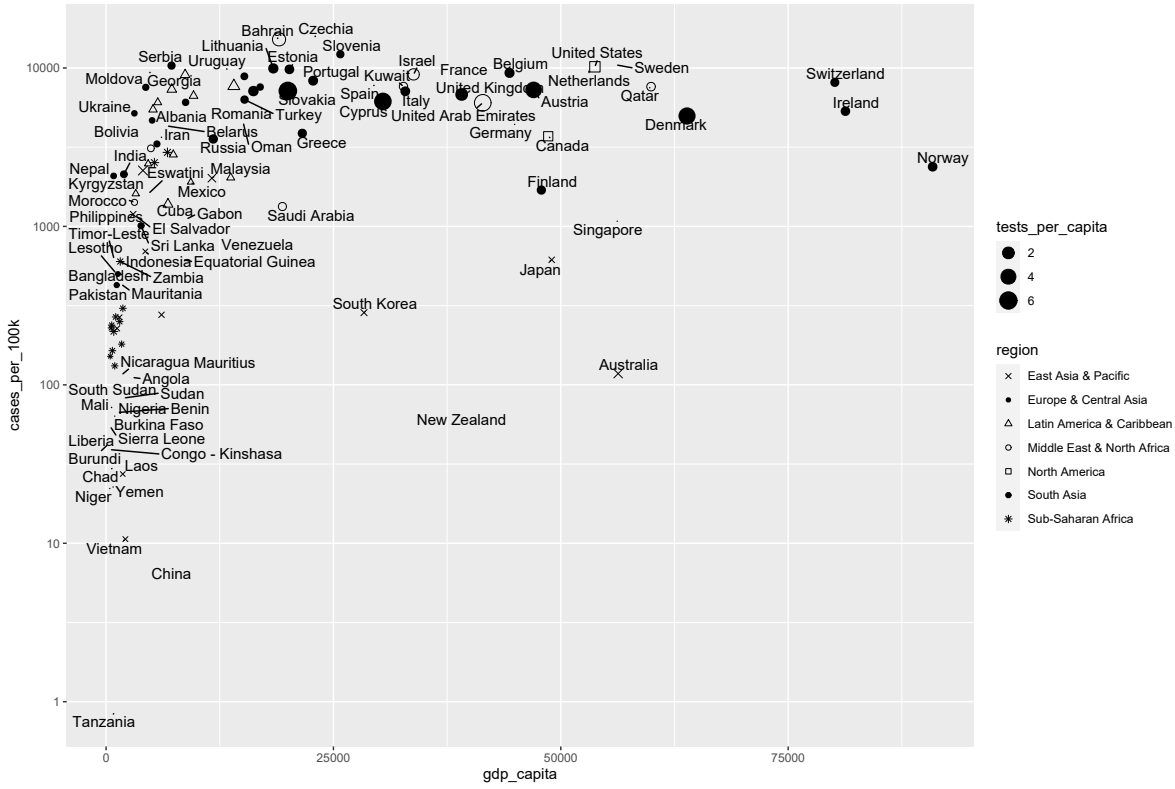


2.5.3 Change the default shape points

```
ggplot(dat, aes(x = gdp_capita, y = cases_per_100k)) +
  geom_point(aes(size = tests_per_capita, shape= region)) +
  geom_text_repel(aes(label = country),
                  min.segment.length = 0, seed = 42,
                  box.padding = 0.1) +
  scale_y_continuous(trans = "log10") +
  scale_shape_manual(values = c(4, 16, 2, 1, 0, 19, 8))
```

```
## Warning: Removed 76 rows containing missing values
```

```
## (geom_point).

## Warning: Removed 3 rows containing missing values
## (geom_text_repel).

## Warning: ggrepel: 63 unlabeled data points
## (too many overlaps). Consider increasing
## max.overlaps
```



However, when a variable is mapped to size (here, `test_per_capital`), it's a good idea to not map a variable to shape (here, `region`). This is because it is difficult to compare the sizes of different shapes (e.g., a size 4 square with a size 4 triangle). Also, some of the shapes really are different sizes: shapes 16 and 19 are both circles, but at any given numeric size, shape 19 circles are visually larger than shape 16 circles.

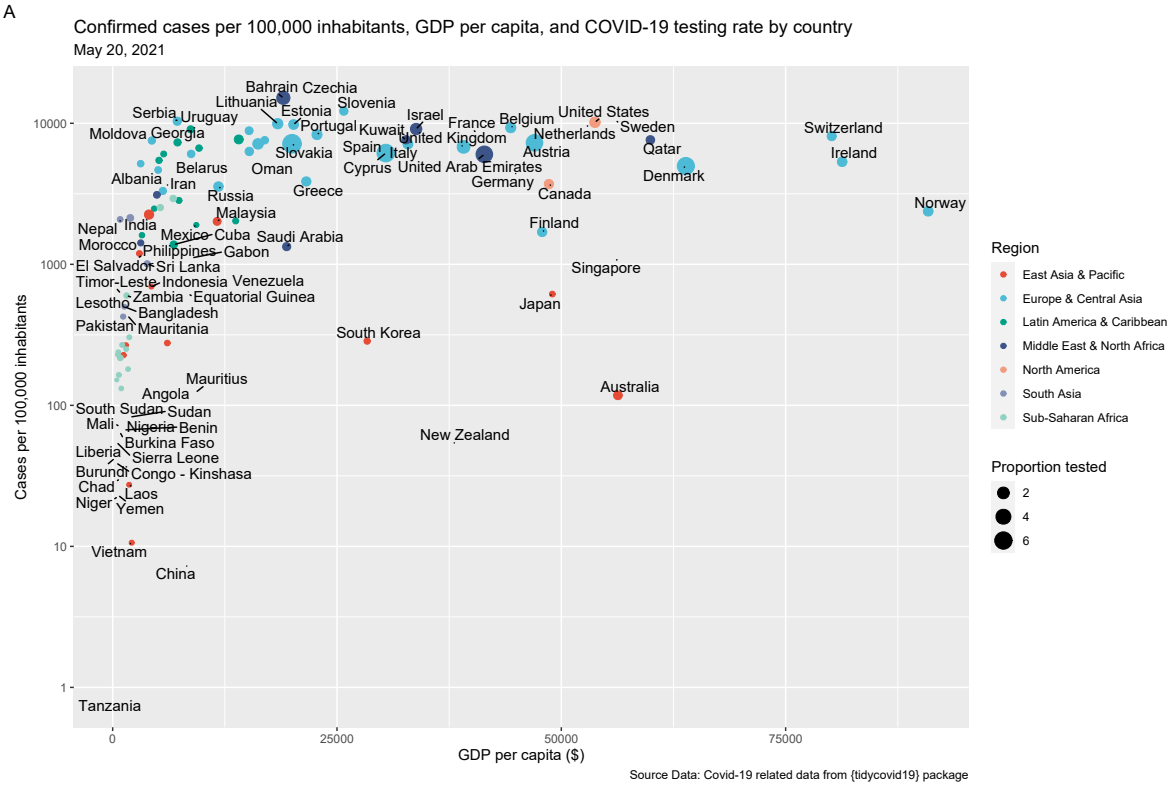
3 Modify axis, legend, and plot labels: `labs()`

```
ggplot(dat, aes(x = gdp_capita, y = cases_per_100k)) +
  geom_point(aes(size = tests_per_capita, color = region)) +
  geom_text_repel(aes(label = country),
                  min.segment.length = 0, seed = 42,
                  box.padding = 0.1
  labs(x = "GDP per capita ($)",
       y = "Cases per 100,000 inhabitants",
       color = "Region",
       size = "Proportion tested",
       title = "Confirmed cases per 100,000 inhabitants,
       GDP per capita, and COVID-19 testing rate by country",
       subtitle = "May 20, 2021",
       caption = "Source Data: Covid-19 related data from
       {tidycovid19} package",
       tag = 'A')
```

```
## Warning: Removed 76 rows containing missing values
## (geom_point).
```

```
## Warning: Removed 3 rows containing missing values
## (geom_text_repel).
```

```
## Warning: ggrepel: 70 unlabeled data points
## (too many overlaps). Consider increasing
## max.overlaps
```



4 Modify theme components: `theme()`

We can fine tune the appearance of the graph using themes. Theme functions (which start with `element_`) control background colors, fonts, grid-lines, legend placement, and other non-data related features of the graph.

This function is very complex since it allows you to specify all the different details contributing to the plot appearance as well as generating your own format and style.

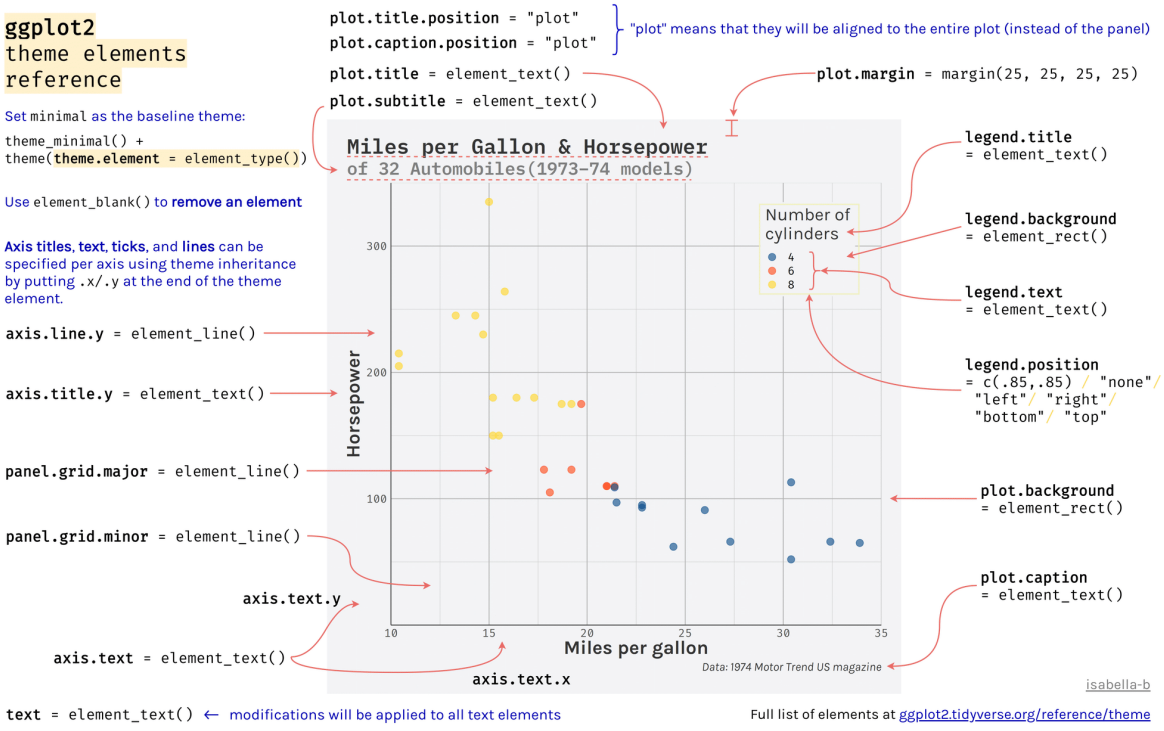


Figure 7: ggplot2 basic theme elements

`ggplot2` theme system comes with multiple `element_` functions:

- `element_text()`: specify the display of text elements
- `element_line()`: specify the display of lines (i.e. axis lines)
- `element_rect()`: specify the display of borders and backgrounds
- `element_blank()`draw nothing

4.1 `element_text()`

While making a plot with `ggplot2`, it automatically chooses appropriate values for various aspects of **element text**. In total there are 10 different aspects/elements of texts in a plot made with `ggplot2`. The figure below shows the anatomy of text elements and the key word in ggplot2 describing the element.

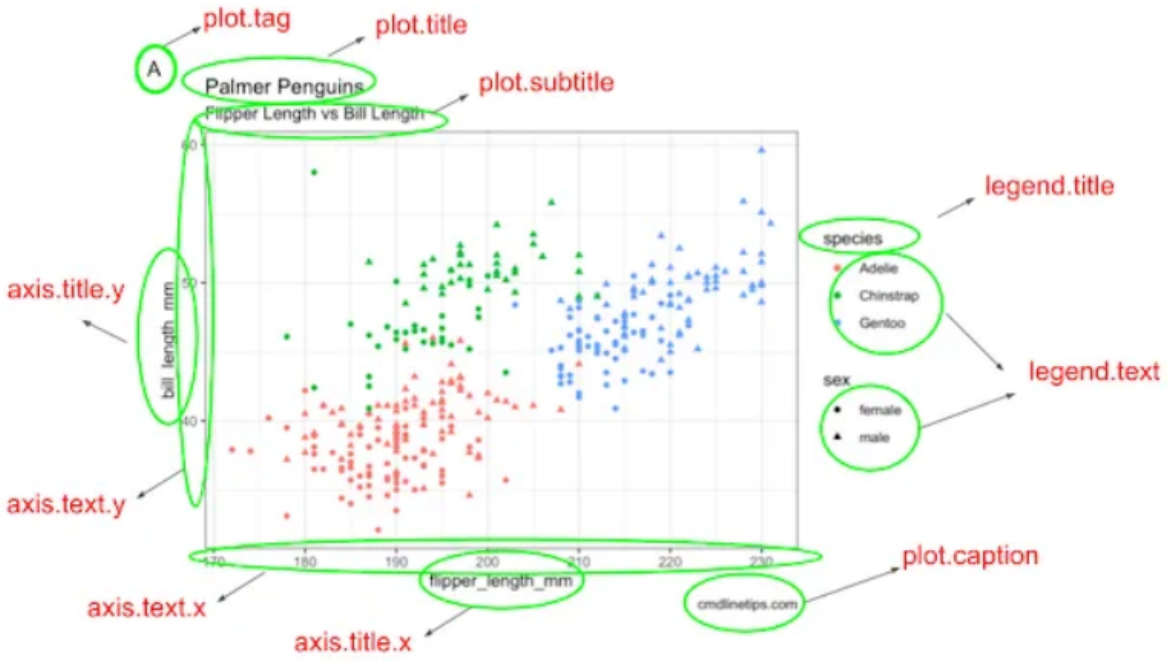


Figure 8: Anatomy of Text Elements in ggplot2

With `element_text()`, we can customize the looks of the the texts. For example, to control the look of title of a plot, we will use `plot.title` element as argument to `theme()` function and use `element_text()` to specify the color, font and size of the plot title.

4.2 `element_line()`

With `element_line()`, we can customize all the lines that are not part of data. For example, we can customize the color of `x` and `y` axis lines, we can make the axis lines as arrows, and we can add second x-axis on top and so on.

Broadly, with `element_line()` we can customize three groups of lines in a plot. First, are `X` and `Y` axis lines. Second, are the lines associated with **tick** on `X` and `Y` axis. And the third, are the **major** and **minor** grid lines along both `X` and `Y` axis. The figure below shows the anatomy of line elements and the key word in ggplot2 describing the element.

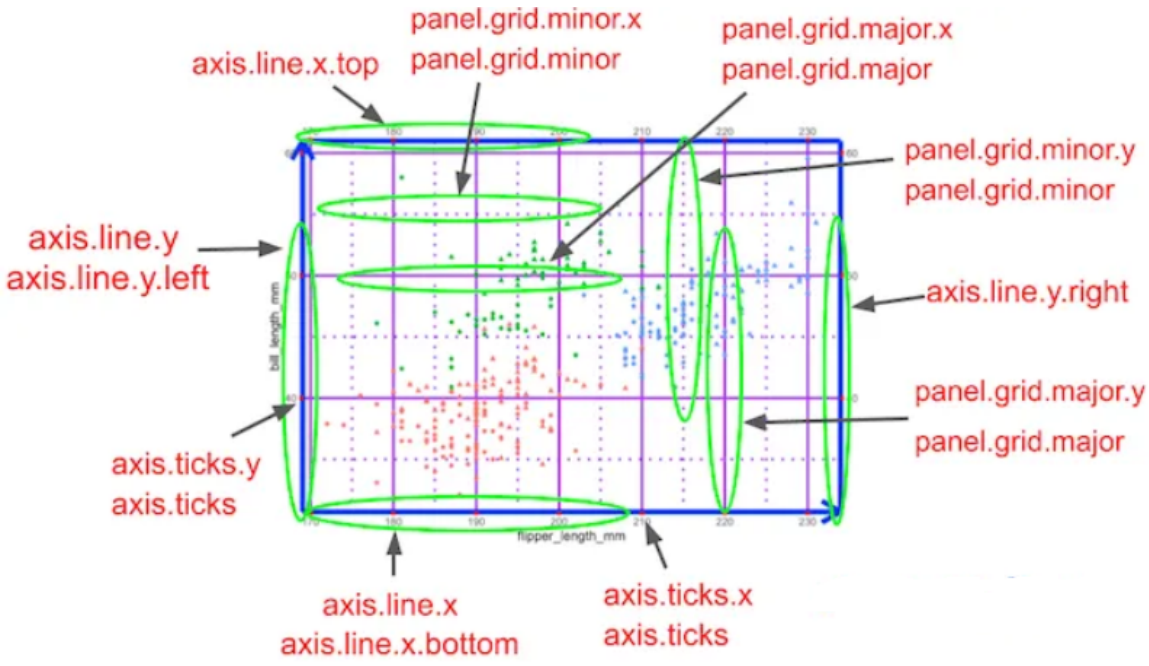


Figure 9: Anatomy of Line Elements in ggplot2

4.3 `element_rect()`

With `element_rect()`, we can customize all things that are rectangular in a plot. For example, we can customize the line and fill colors of rectangles that define borders and backgrounds in a plot.

In total there are 7 aspects of rectangular elements we can control using using `element_rect()`.

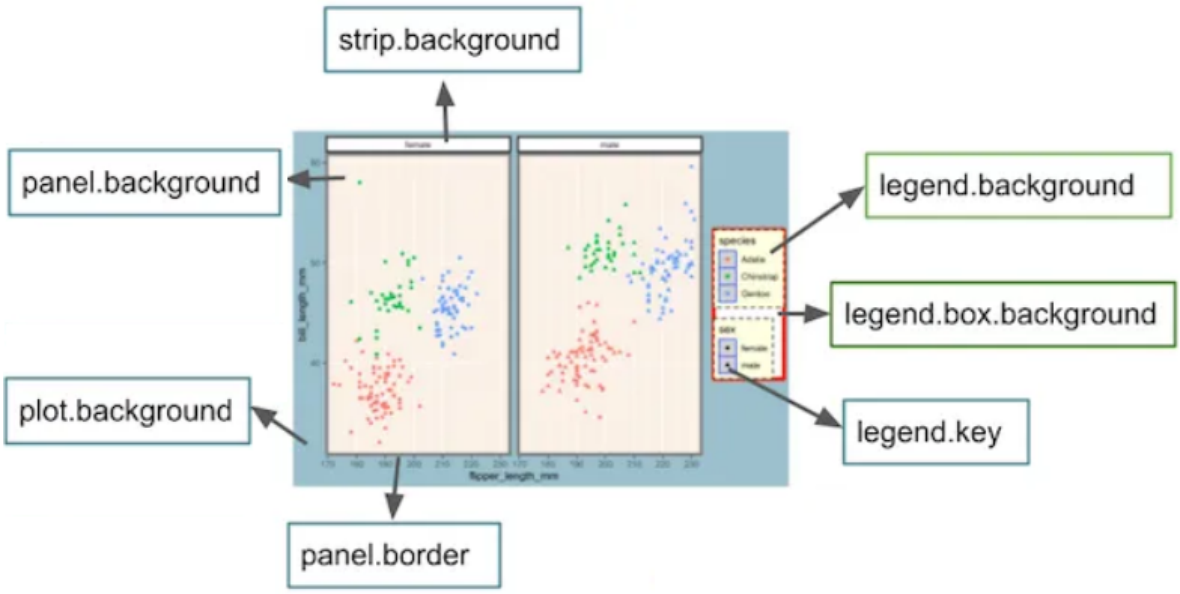


Figure 10: Anatomy of Rectangle Elements in ggplot2

4.4 `element_blank()`

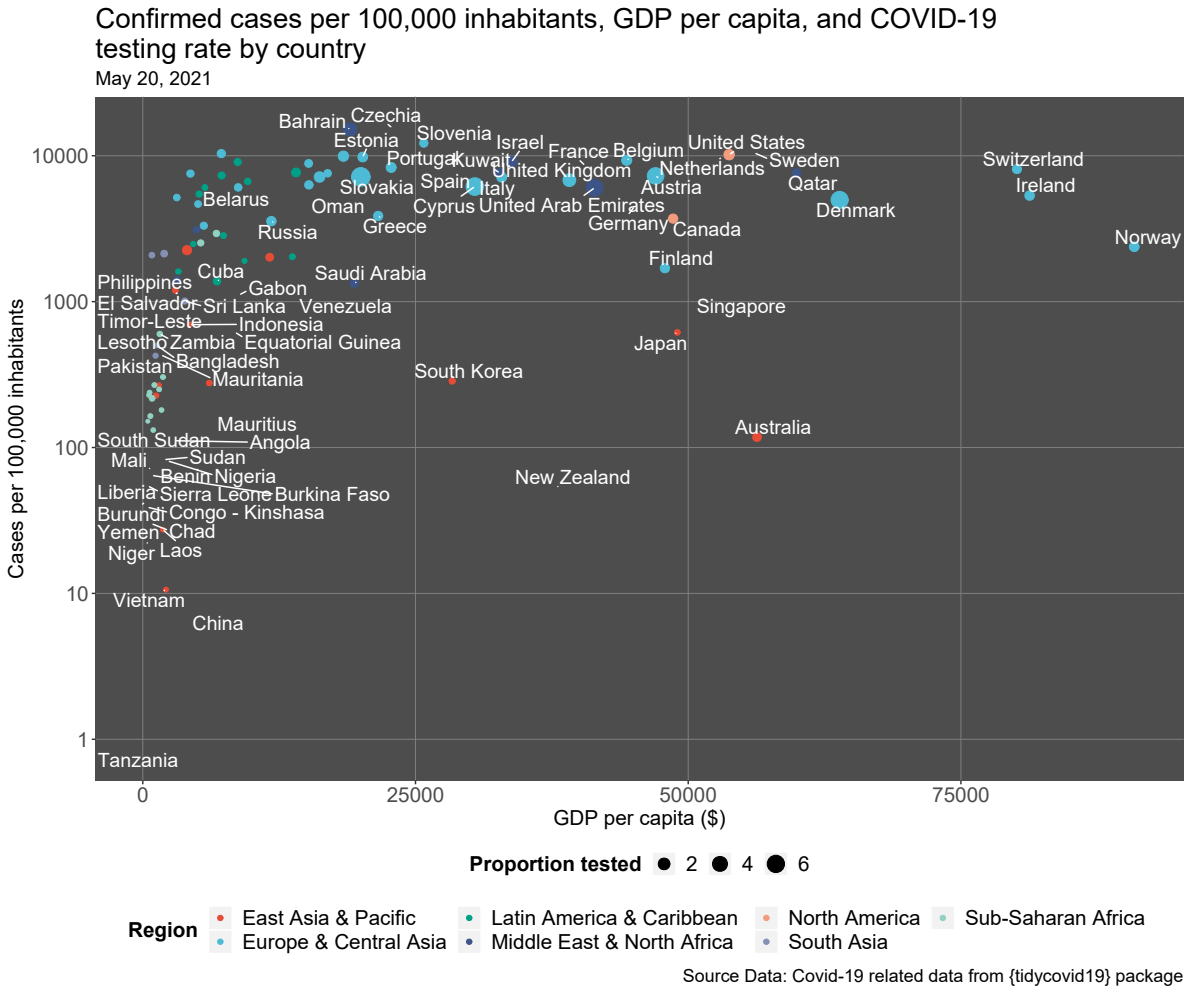
If we look at the `ggplot2` theme documentation page, we can see that `element_blank()` does not take any arguments. So what does it do? As it says `element_blank()`: draws nothing, and assigns no space. We can use `element_blank()` if we don't want any specific non-data plot elements drawn on the plot (we disable them). With `element_blank()` we can suppress or remove change for each of the theme elements. For example, using `panel.grid.minor = element_blank()` in the `theme()` we disable the minor grid lines of the panel.

4.5 Modify theme elements in practice

```
ggplot(dat, aes(x = gdp_capita, y = cases_per_100k)) +
  geom_point(aes(size = tests_per_capita, color = region)) +
  geom_text_repel(aes(label = country),
                  min.segment.length = 0, seed = 42,
```

```
                box.padding = 0.1, color = "white", size = 5) +
scale_y_continuous(trans = "log10") +
scale_color_npg() +
labs(x = "GDP per capita ($)",
     y = "Cases per 100,000 inhabitants",
     color = "Region",
     size = "Proportion tested",
     title = "Confirmed cases per 100,000 inhabitants,
     GDP per capita, and COVID-19 testing rate by country",
     subtitle = "May 20, 2021",
     caption = "Source Data: Covid-19 related data from {tidycovid19}
     package") +
theme(
      # background, panel and grid lines
      plot.background = element_blank(),
      panel.background = element_rect(fill = "grey30"),
      panel.grid.major = element_line(size = 0.1, color = "grey50"),
      panel.grid.minor = element_blank(),
      # title, subtitle and caption
      plot.title = element_text(size = 20),
      plot.subtitle = element_text(size = 14),
      plot.caption = element_text(size = 13),
      # axis
      axis.title = element_text(size = 15),
      axis.text = element_text(size = 15),
      # legend
      legend.background = element_blank(),
      legend.title = element_text(size = 15, face = "bold"),
      legend.text = element_text(size = 15),
      legend.key = element_rect(color = "white"),
      legend.position="bottom",
      legend.box = 'vertical'
      )
```

```
## Warning: Removed 76 rows containing missing values
## (geom_point).

## Warning: Removed 3 rows containing missing values
## (geom_text_repel).

## Warning: ggrepel: 82 unlabeled data points
## (too many overlaps). Consider increasing
## max.overlaps
```



The figure depicts strong, positive relationships between the following pairs of variables: wealth/COVID-19 cases, wealth/COVID-19 testing, and COVID-19 cases/testing.

Note the use of two plot attributes (size and color) to depict additional variables.

4.6 Adding an in-build theme from `ggplot2`

4.6.1 Add a minimal in-build theme: `theme_minimal()`

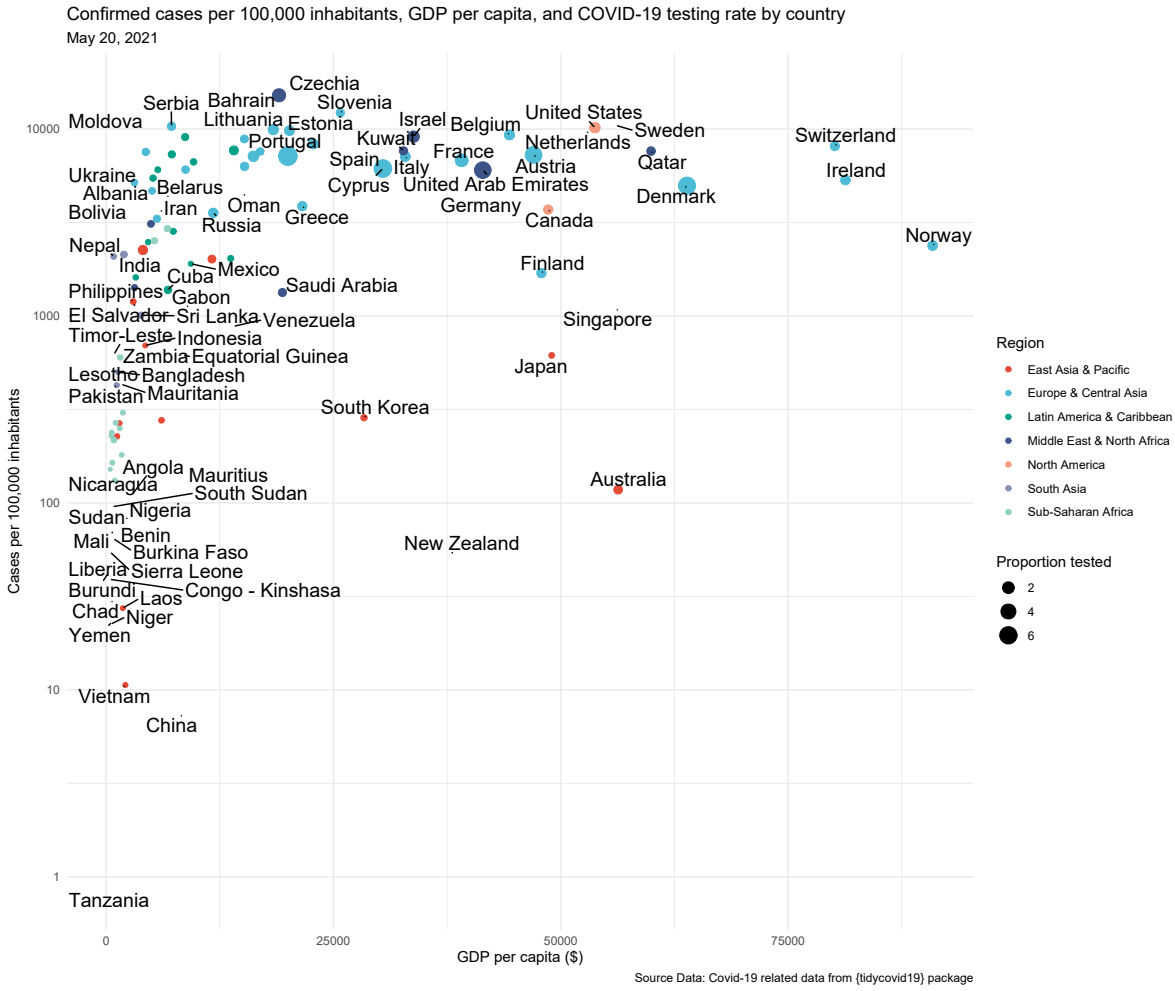
While it is possible to build up a new theme from the ground it is usually easier and less error-prone to modify an existing theme. This is done in `ggplot2` as well as can be seen by looking at e.g., `theme_minimal()`:

```
ggplot(dat, aes(x = gdp_capita, y = cases_per_100k)) +
  geom_point(aes(size = tests_per_capita, color = region)) +
  geom_text_repel(aes(label = country),
                  min.segment.length = 0, seed = 42,
                  box.padding = 0.1, color = "black", size = 5) +
  scale_y_continuous(trans = "log10") +
  scale_color_npg() +
  labs(x = "GDP per capita ($)",
       y = "Cases per 100,000 inhabitants",
       color = "Region",
       size = "Proportion tested",
       title = "Confirmed cases per 100,000 inhabitants, GDP per
       capita, and COVID-19 testing rate by country",
       subtitle = "May 20, 2021",
       caption = "Source Data: Covid-19 related data from
       {tidycovid19} package") +
  theme_minimal()
```

```
## Warning: Removed 76 rows containing missing values
## (geom_point).
```

```
## Warning: Removed 3 rows containing missing values
## (geom_text_repel).
```

```
## Warning: ggrepel: 73 unlabeled data points
## (too many overlaps). Consider increasing
## max.overlaps
```



In-build theme from ggplot2:

- theme_gray() – signature ggplot2 theme
- theme_bw() – dark on light ggplot2 theme
- theme_linedraw() – uses black lines on white backgrounds only
- theme_light() – similar to linedraw() but with grey lines aswell
- theme_dark() – lines on a dark background instead of light

- theme_minimal() – no background annotations, minimal feel.
- theme_classic() – theme with no grid lines.
- theme_void() – empty theme with no elements

4.6.2 In-build theme and order of the theme elements

In-build themes like `theme_grey()` or `theme_minimal()` are really just collections of changes to `theme()`, so the order is important when using a complete theme.

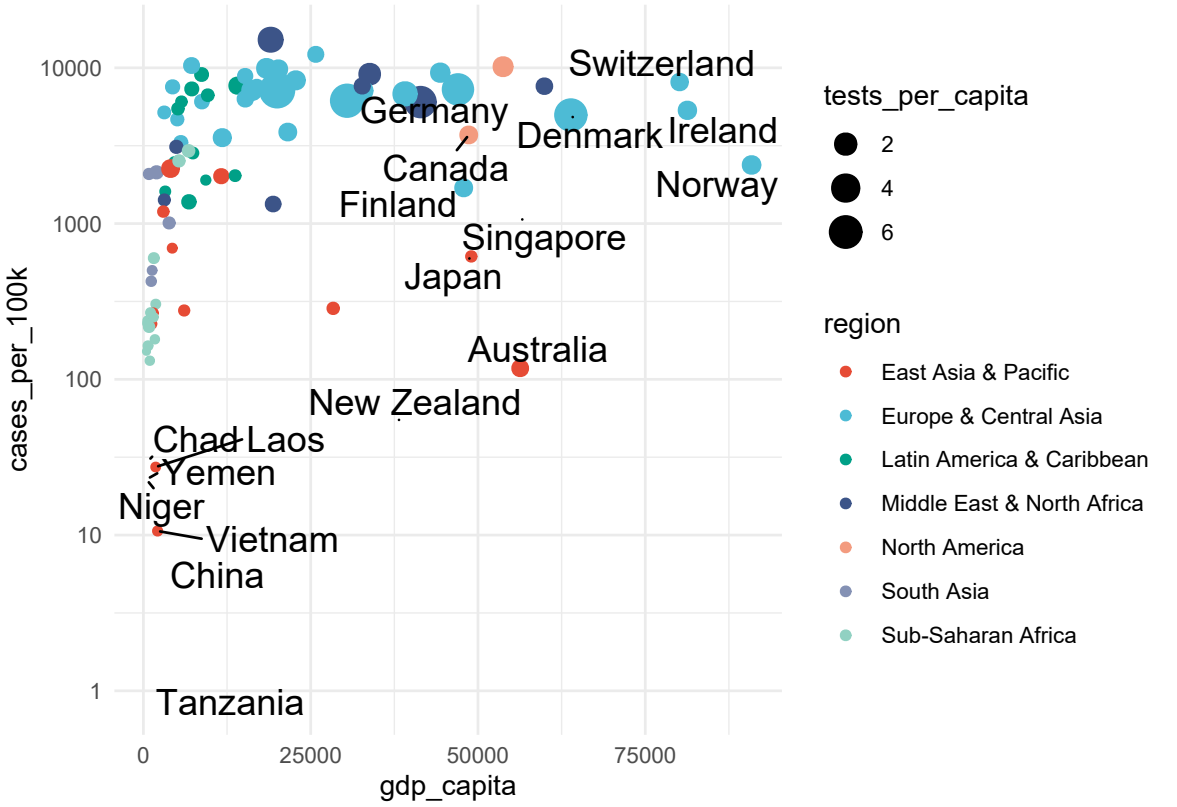
For example, if we want to turn off the gridlines in the plot panel (using `element_blank`) and the in-build theme is at the bottom of the code, we 'll still have panel gridlines! That's because `theme_minimal()` turns them on, as we typed it after we turned it off.

```
ggplot(dat, aes(x = gdp_capita, y = cases_per_100k)) +
  geom_point(aes(size = tests_per_capita, color = region)) +
  geom_text_repel(aes(label = country),
                  min.segment.length = 0, seed = 42,
                  box.padding = 0.1, color = "black", size = 5) +
  scale_y_continuous(trans = "log10") +
  scale_color_npg() +
  theme(panel.grid = element_blank()) +
  theme_minimal()
```

```
## Warning: Removed 76 rows containing missing values
## (geom_point).
```

```
## Warning: Removed 3 rows containing missing values
## (geom_text_repel).
```

```
## Warning: ggrepel: 134 unlabeled data points
## (too many overlaps). Consider increasing
## max.overlaps
```



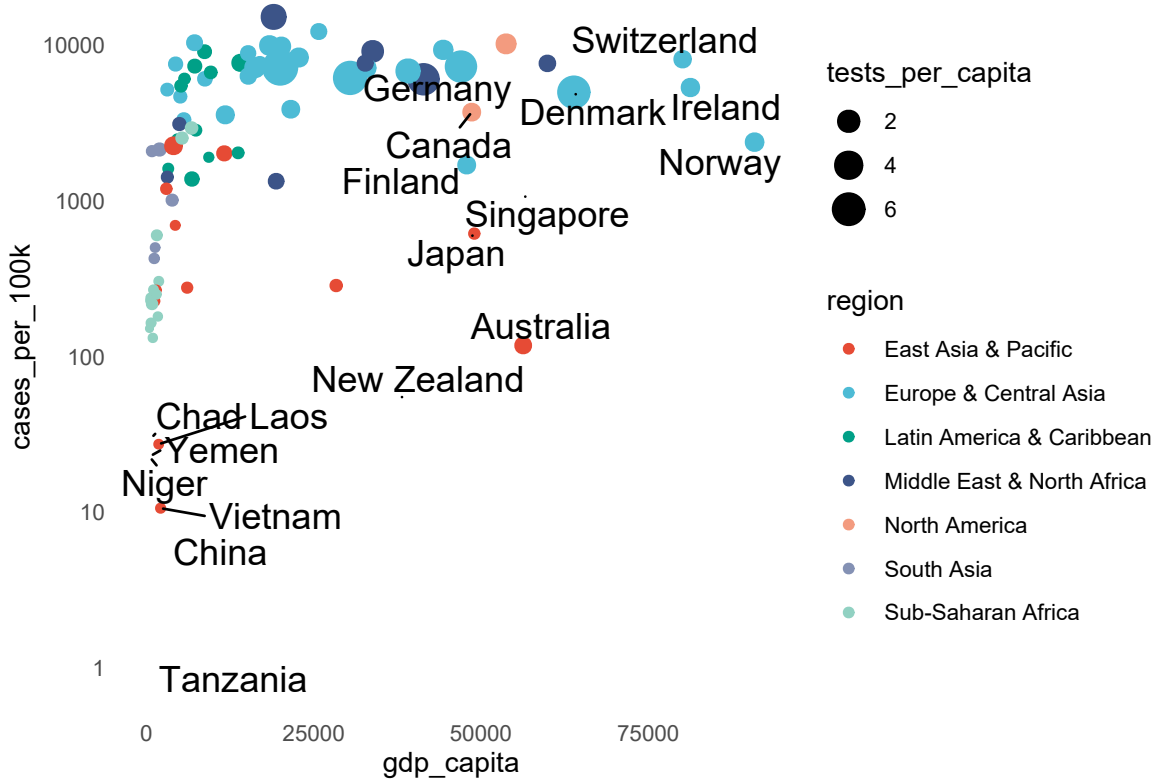
In conclusion, if we want to use both `theme_minimal()` and remove the gridlines, we need to make sure any theme adjustments come after `theme_minimal()`:

```
ggplot(dat, aes(x = gdp_capita, y = cases_per_100k)) +
  geom_point(aes(size = tests_per_capita, color = region)) +
  geom_text_repel(aes(label = country),
                  min.segment.length = 0, seed = 42,
                  box.padding = 0.1, color = "black", size = 5) +
  scale_y_continuous(trans = "log10") +
  scale_color_npg() +
  theme_minimal() +
  theme(panel.grid = element_blank())
```

```
## Warning: Removed 76 rows containing missing values
## (geom_point).
```

```
## Warning: Removed 3 rows containing missing values
```

```
## (geom_text_repel).

## Warning: ggrepel: 134 unlabeled data points
## (too many overlaps). Consider increasing
## max.overlaps
```



4.6.3 In-build theme and specificication of the theme elements

Suppose we want to change the angle of axis title to 45 degrees:

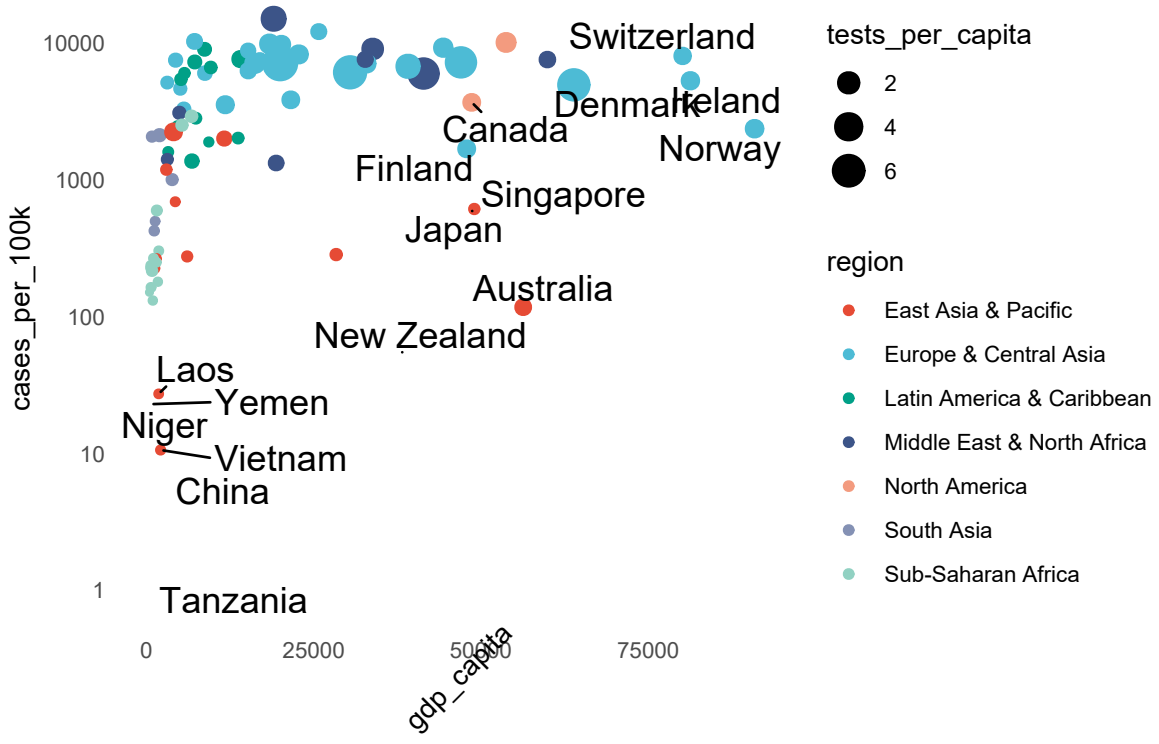
```
ggplot(dat, aes(x = gdp_capita, y = cases_per_100k)) +
  geom_point(aes(size = tests_per_capita, color = region)) +
  geom_text_repel(aes(label = country),
                  min.segment.length = 0, seed = 42,
                  box.padding = 0.1, color = "black", size = 5) +
  scale_y_continuous(trans = "log10") +
```

```
  scale_color_npg() +
  theme_minimal() +
  theme(panel.grid = element_blank(),
        axis.title = element_text(angle = 45))
```

```
## Warning: Removed 76 rows containing missing values
## (geom_point).
```

```
## Warning: Removed 3 rows containing missing values
## (geom_text_repel).
```

```
## Warning: ggrepel: 136 unlabeled data points
## (too many overlaps). Consider increasing
## max.overlaps
```



As we observe the change has been applied only to x-axis title. So, why the change takes effect only in one axis? The reason for this has to do with the `theme_minimal` we are using.

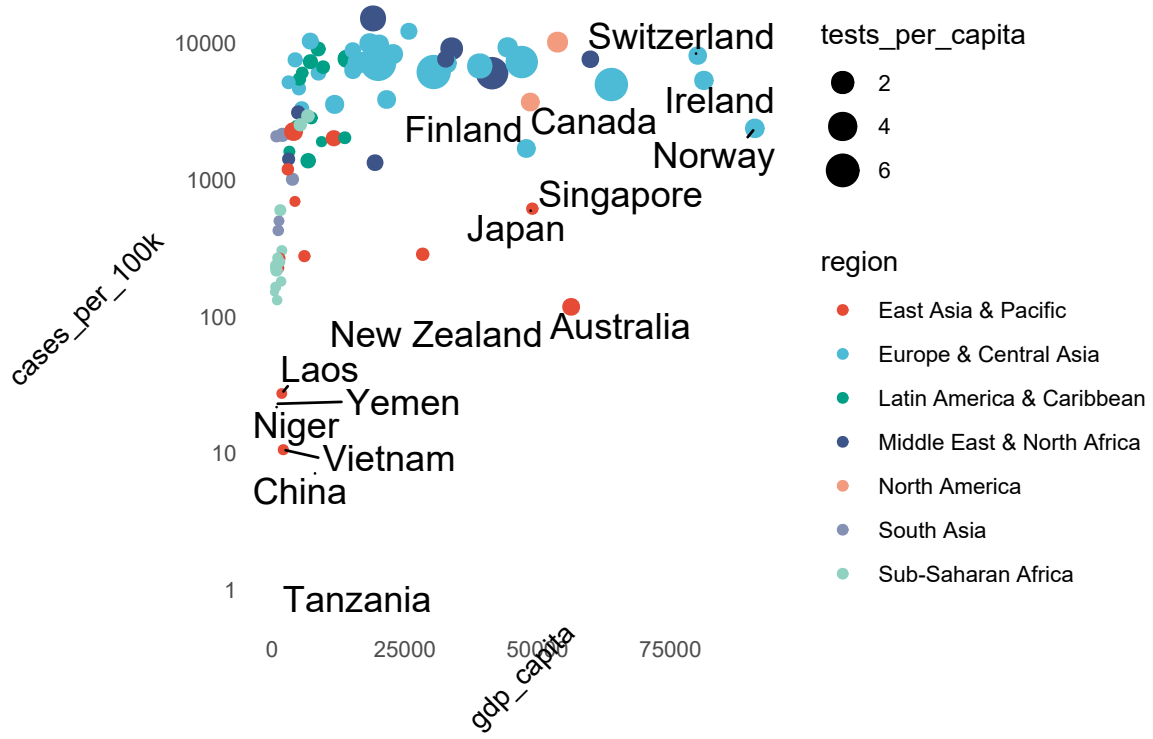
The `theme_minimal` sets the axis.title.y to 90 degrees, so if we want to change it, we need to edit that very specific element and not any one above it (such as `axis.title`).

```
ggplot(dat, aes(x = gdp_capita, y = cases_per_100k)) +
  geom_point(aes(size = tests_per_capita, color = region)) +
  geom_text_repel(aes(label = country),
                  min.segment.length = 0, seed = 42,
                  box.padding = 0.1, color = "black", size = 5) +
  scale_y_continuous(trans = "log10") +
  scale_color_npg() +
  theme_minimal() +
  theme(panel.grid = element_blank(),
        axis.title = element_text(angle = 45),
        axis.title.y = element_text(angle = 45, vjust = 0.5))
```

```
## Warning: Removed 76 rows containing missing values
## (geom_point).
```

```
## Warning: Removed 3 rows containing missing values
## (geom_text_repel).
```

```
## Warning: ggrepel: 137 unlabeled data points
## (too many overlaps). Consider increasing
## max.overlaps
```



NOTE A crucial piece of the theme system is that when we are customizing theme elements the most specific wins.

5 Focus the attention on specific data or plot area

5.1 Blur points

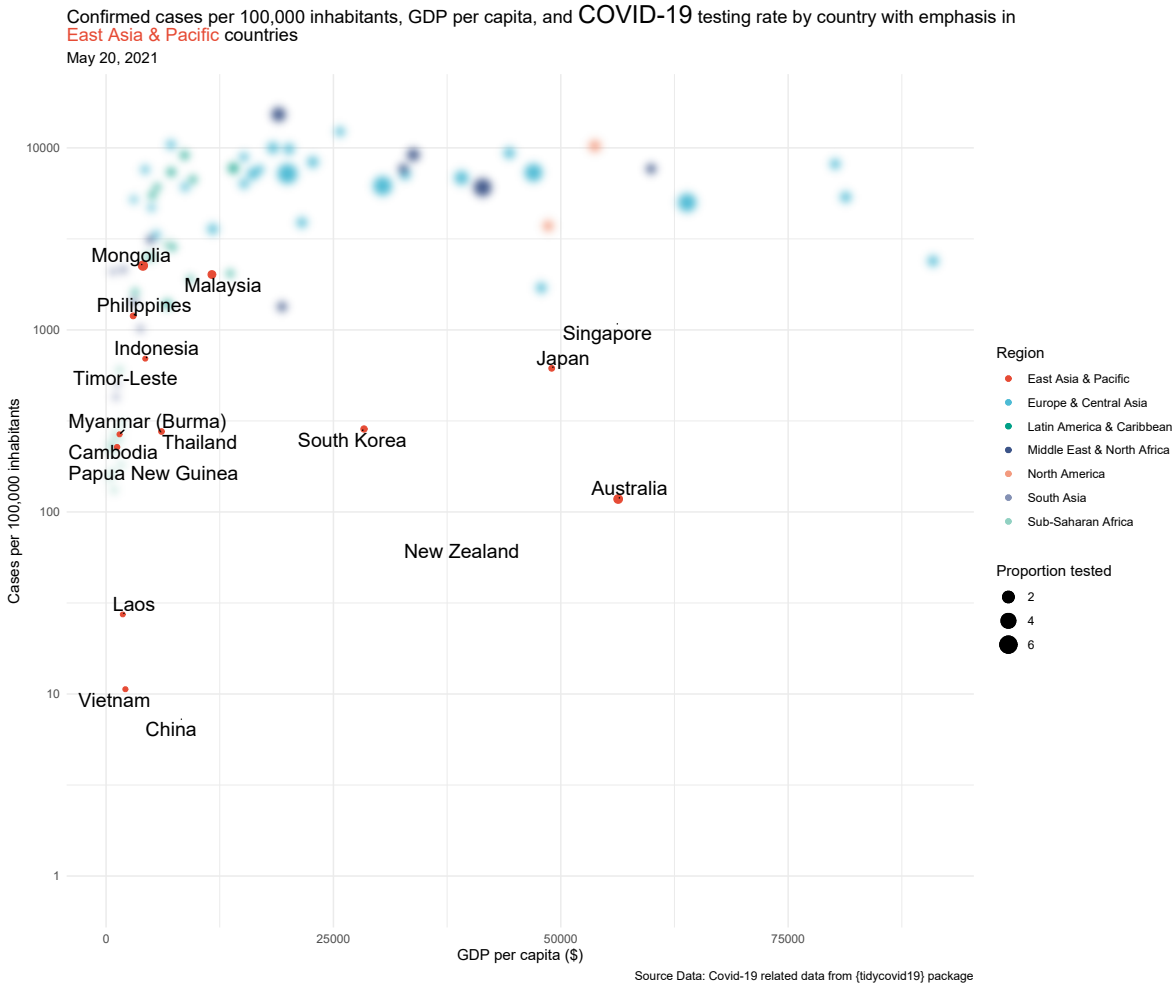
We can focus the attention on specific region e.g. `East Asia and Pacific` countries. One solution is to blur all the points that we do not want to highlight. We will use the `with_blur()` function, which blurs the layers to which it is applied.

```r
ggplot(dat, aes(x = gdp_capita, y = cases_per_100k,
                size = tests_per_capita, color = region)) +
  with_blur(
  geom_point(data=dat %>% filter(region != "East Asia & Pacific")),
  sigma = unit(0.95, 'mm') # you can choose the amount of blur
  ) +
  geom_point(data=dat %>% filter(region == "East Asia & Pacific")) +
  geom_text_repel(data=dat %>%
                    filter(region == "East Asia & Pacific"),
                  aes(label = country),
                  min.segment.length = 0, seed = 42,
                  box.padding = 0.1, color = "black", size = 5) +
  scale_y_continuous(trans = "log10") +
  scale_color_npg() +
  labs(x = "GDP per capita ($)",
       y = "Cases per 100,000 inhabitants",
       color = "Region",
       size = "Proportion tested",
       title = "Confirmed cases per 100,000 inhabitants, GDP per
       capita, and <span style='font-size:18pt'>COVID-19</span>
       testing rate by country with emphasis in  <br> <span
       style='color:#E64B35FF'>East Asia & Pacific</span>
       countries",
       subtitle = "May 20, 2021",
       caption = "Source Data: Covid-19 related data from
       {tidycovid19} package") +
  theme_minimal() +
  theme(plot.title = element_markdown())
```

```
## Warning: Removed 71 rows containing missing values
## (blurred_geom).
```

```
## Warning: Removed 5 rows containing missing values
```

```
## (geom_point).
```



In the above graph, additional styling was applied via inline CSS (Cascading Style Sheets) to the title. The CSS properties color, font-size, and font-family are currently supported.

> The `ggtext` package defines a new theme element, `element_markdown()`. It behaves similarly to element_text() but render the provided text as `markdown/html`. `element_markdown()` is meant as a direct replacement for `element_text()`, and it renders text without word wrapping. To start a new line, use the `<br>` tag or add two spaces before the end of a line.

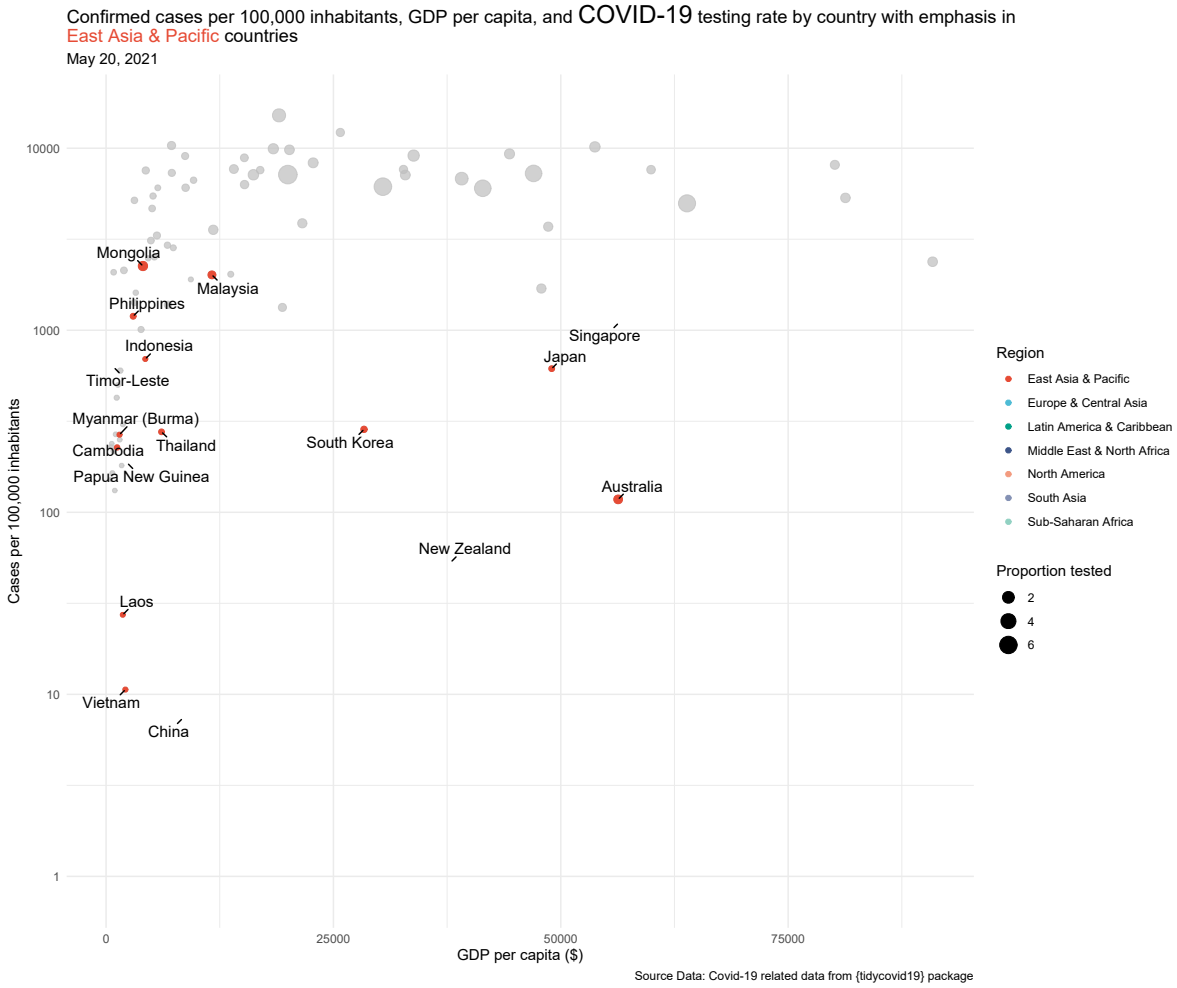
5.2 Highlight data points

```
ggplot(dat, aes(x = gdp_capita, y = cases_per_100k,
                color = region)) +
  geom_point(aes(size = tests_per_capita)) +
  scale_y_continuous(trans = "log10") +
  scale_color_npg() +
  gghighlight(region == "East Asia & Pacific", keep_scales = TRUE,
              use_direct_label = FALSE) +
  geom_text_repel(aes(label = country),
                  min.segment.length = 0, seed = 42,
                  box.padding = 0.3,
                  color = "black", size = 4) +
  labs(x = "GDP per capita ($)",
       y = "Cases per 100,000 inhabitants",
       color = "Region",
       size = "Proportion tested",
       title = "Confirmed cases per 100,000 inhabitants, GDP per
       capita, and <span style='font-size:18pt'>COVID-19</span>
       testing rate by country with emphasis in  <br> <span
       style='color:#E64B35FF'>East Asia & Pacific</span>
       countries",
       subtitle = "May 20, 2021",
       caption = "Source Data: Covid-19 related data from {tidycovid19}
       package") +
  theme_minimal() +
  theme(plot.title = element_markdown())
```

```
## Warning: Tried to calculate with group_by(), but the calculation failed.
## Falling back to ungrouped filter operation...
```

```
## Warning: Removed 76 rows containing missing values
## (geom_point).
```

```
## Warning: Removed 5 rows containing missing values
## (geom_point).
```



5.3 Limit Axis Range (Zoom)

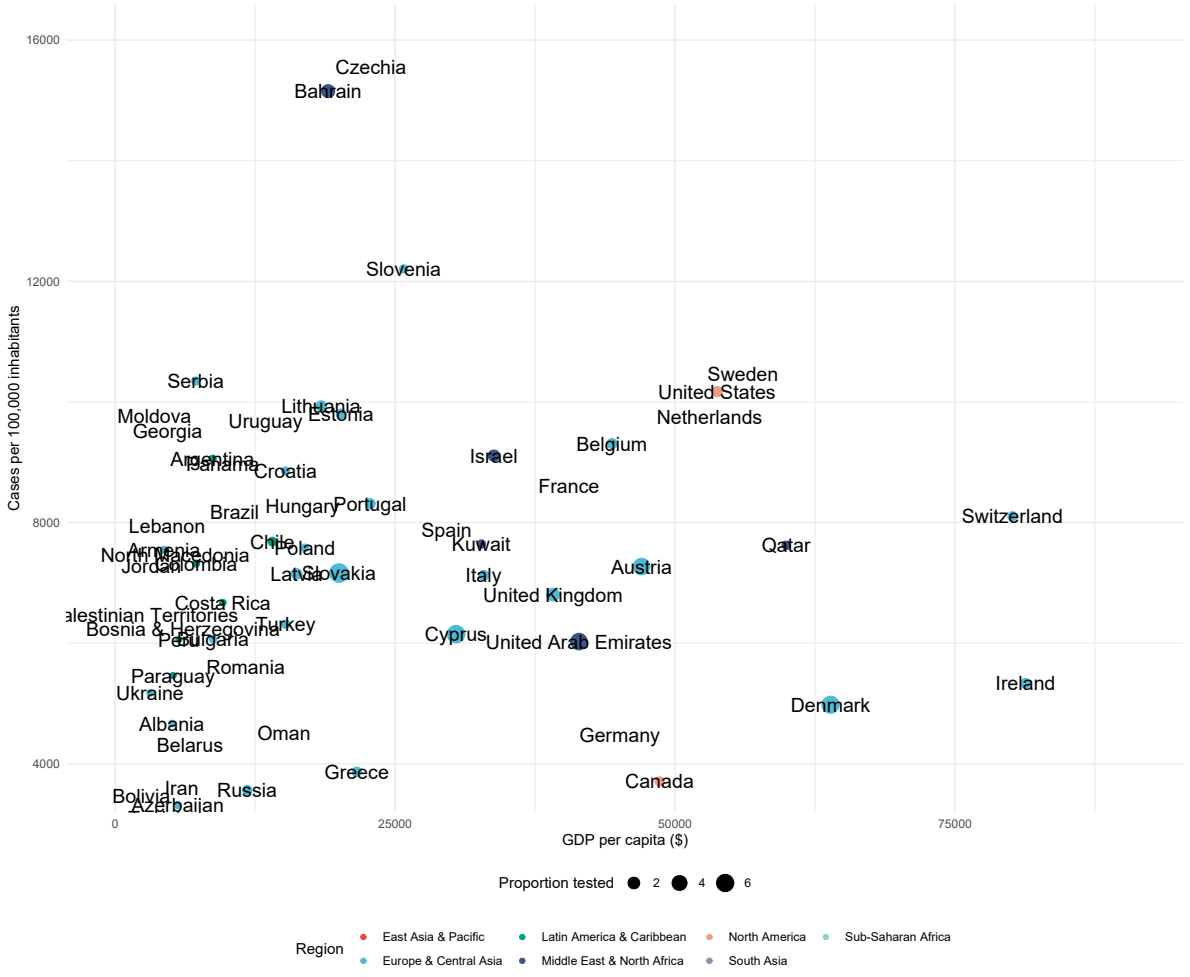
Sometimes we want to zoom into our data. We can do this without subsettting our data. We can change the `X` and `Y` axis limits by zooming in to the region of interest without deleting points. This is done using `coord_cartesian()`.

```
ggplot(dat, aes(x = gdp_capita, y = cases_per_100k)) +
  geom_point(aes(size = tests_per_capita, color = region)) +
```

```
  geom_text(aes(label = country), size = 5) +
  scale_color_npg() +
  labs(x = "GDP per capita ($)",
       y = "Cases per 100,000 inhabitants",
       color = "Region",
       size = "Proportion tested") +
  coord_cartesian(ylim=c(3.8*10^3, 1.6*10^4)) +
  theme_minimal() +
  theme(legend.position= "bottom",
        legend.box = 'vertical')
```

```
## Warning: Removed 76 rows containing missing values
## (geom_point).
```

```
## Warning: Removed 3 rows containing missing values
## (geom_text).
```



Note that the text is above the points, because it is the 'top layer' of the plot. The `geom_*` layers that appear early in the command are drawn first, and can be obscured by the `geom_*` layers that come after them.

What happens if we switch the order of the `geom_point()` and `geom_text()` functions above? What do we notice?

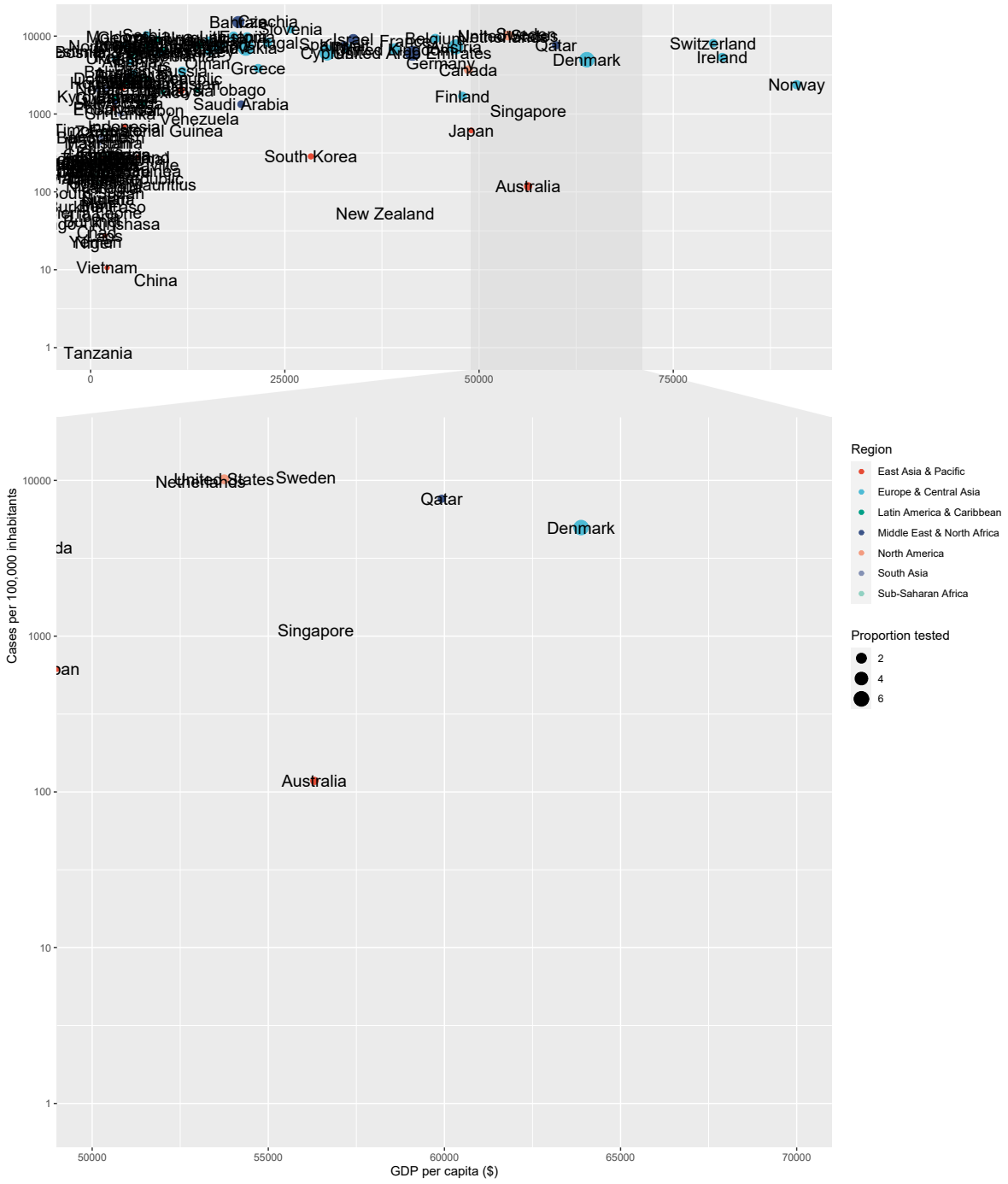
5.4 Highlight a certain area of the figure

The `facet_zoom()` function from the package `ggforce` provides the means to zoom in on a subset of the data, while keeping the view of the full dataset as a separate panel.

```
ggplot(dat, aes(x = gdp_capita, y = cases_per_100k)) +
  geom_point(aes(size = tests_per_capita, color = region)) +
  geom_text(aes(label = country), size = 5) +
  scale_y_log10() +
  scale_color_npg() +
  labs(x = "GDP per capita ($)",
       y = "Cases per 100,000 inhabitants",
       color = "Region",
       size = "Proportion tested"
       ) +
  facet_zoom(xlim = c(50000, 70000))
```

```
## Warning: Removed 152 rows containing missing values
## (geom_point).
```

```
## Warning: Removed 6 rows containing missing values
## (geom_text).
```



Another way to zoom-in detail is using the package `ggpp`. The `geom_plot()` geometry plots ggplot objects, nested in a tibble passed as data argument, using aesthetics `x` and `y` for positioning, and label for the ggplot object containing the definition of the plot to be nested. As an example we produce a plot where the inset plot is a zoomed-in detail from the main plot. In this case the main and inset plots start as the same plot.

```r
# the inset plot
gp1 <- ggplot(dat, aes(x = gdp_capita, y = cases_per_100k)) +
  geom_point(aes(size = tests_per_capita, color = region)) +
  geom_text(aes(label = country), size = 5) +
  scale_color_npg() +
  theme(legend.position = "none")


data.tb <-
  tibble(x = 110000, y = 1,
         plot = list(gp1 +
                       coord_cartesian(xlim = c(30000, 80000),
                                       ylim = c(4000, 11000)) +
                       labs(x = NULL, y = NULL) ))

# the main plot
ggplot(dat, aes(x = gdp_capita, y = cases_per_100k)) +
  geom_point(aes(size = tests_per_capita, color = region)) +
  geom_text(aes(label = country), size = 4) +
  geom_plot(data = data.tb, aes(x, y, label = plot)) +
  annotate(geom = "rect",
            xmin = 30000, xmax = 80000,
           ymin = 4000, ymax = 11000,
           linetype = "dotted",
           fill = NA, colour = "black") +
  scale_y_log10() +
  scale_color_npg() +
  labs(x = "GDP per capita ($)",
       y = "Cases per 100,000 inhabitants",
       color = "Region",
       size = "Proportion tested")
```

6 Split a plot into a matrix of panels: `facet()`

Another technique for displaying categorical variables on a plot is faceting. The facet approach partitions a plot into a matrix of panels. Each panel shows a different subset of the data. Coordinate systems and faceting control the position of elements of the plot.

There are two types of faceting provided by `ggplot2`: `facet_wrap` and `facet_grid`. **Facet wrap** produces a 1d ribbon of panels that is wrapped into 2d to save space, while **facet grid** produces a `2d` grid of panels defined by variables which form the rows and columns. These differences are illustrated in Figure 11.

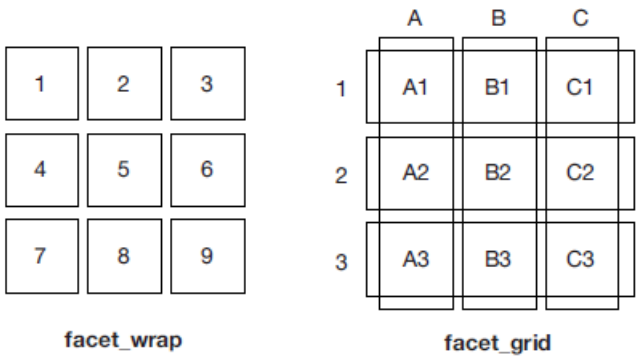


Figure 11: A sketch illustrating the difference between the two faceting systems.

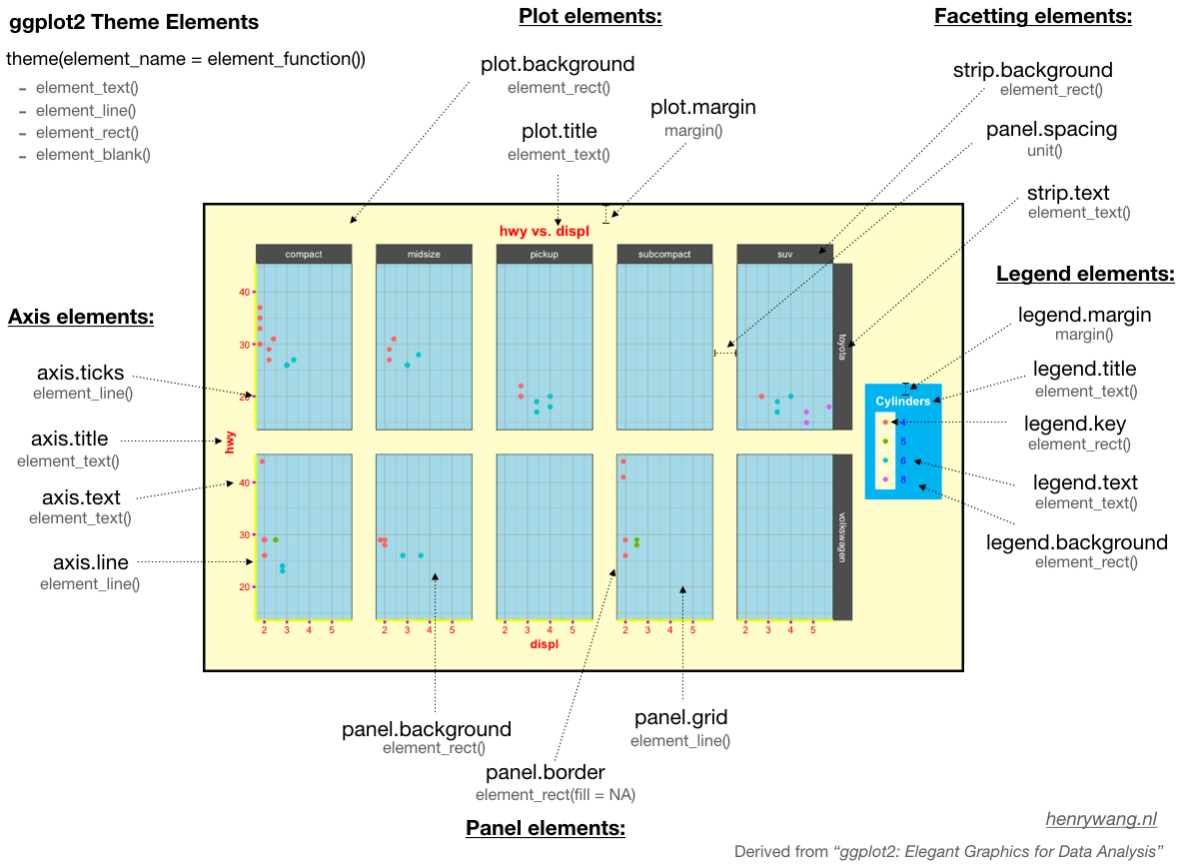


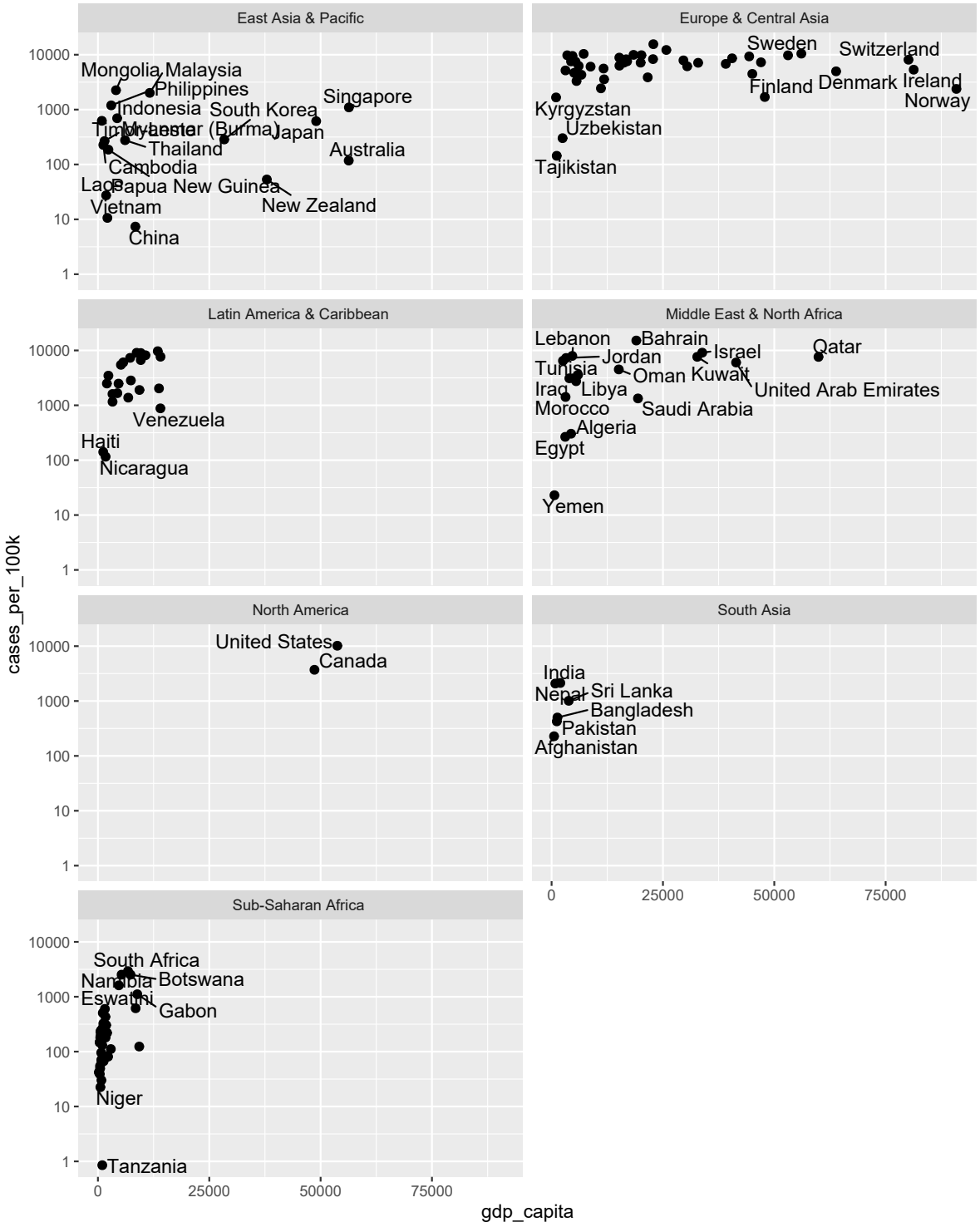
Figure 12: ggplot2 basic theme and facet elements

6.1 `facet_wrap` multiple-panel plots based on one variable

Facets can be placed next to each other, wrapping with a certain number of columns or rows. The label for each plot will be at the top of the plot.

```
ggplot(dat, aes(x = gdp_capita, y = cases_per_100k)) +
  geom_point(size = 2.0) +
  geom_text_repel(aes(label = country),
                  min.segment.length = 0, seed = 42,
                  box.padding = 0.1,  color = "black", size = 4) +
  scale_y_continuous(trans = "log10") +
  facet_wrap(~region,  ncol=2)
```

```
## Warning: Removed 3 rows containing missing values
## (geom_point).

## Warning: Removed 3 rows containing missing values
## (geom_text_repel).

## Warning: ggrepel: 19 unlabeled data points
## (too many overlaps). Consider increasing
## max.overlaps

## Warning: ggrepel: 36 unlabeled data points
## (too many overlaps). Consider increasing
## max.overlaps

## Warning: ggrepel: 35 unlabeled data points
## (too many overlaps). Consider increasing
## max.overlaps

## Warning: ggrepel: 2 unlabeled data points
## (too many overlaps). Consider increasing
## max.overlaps
```



The points in a scatter plot may obscure each other and prevent the viewer from accurately assessing the distribution of the data. This is called overplotting. If the amount of overplotting is low, you may be able to alleviate it by using smaller points, by us-

ing a different shape through which other points can be seen, or making the points semitransparent using `alpha`.

```
ggplot(dat, aes(x = gdp_capita, y = cases_per_100k)) +
  geom_point(size = 1.5, alpha = 0.4) +
  geom_text_repel(aes(label = country),
                  min.segment.length = 0, seed = 42,
                  box.padding = 0.1, color = "black", size = 4) +
  scale_y_continuous(trans = "log10") +
  facet_wrap(~region,  ncol=2)
```

```
## Warning: Removed 3 rows containing missing values
## (geom_point).
```

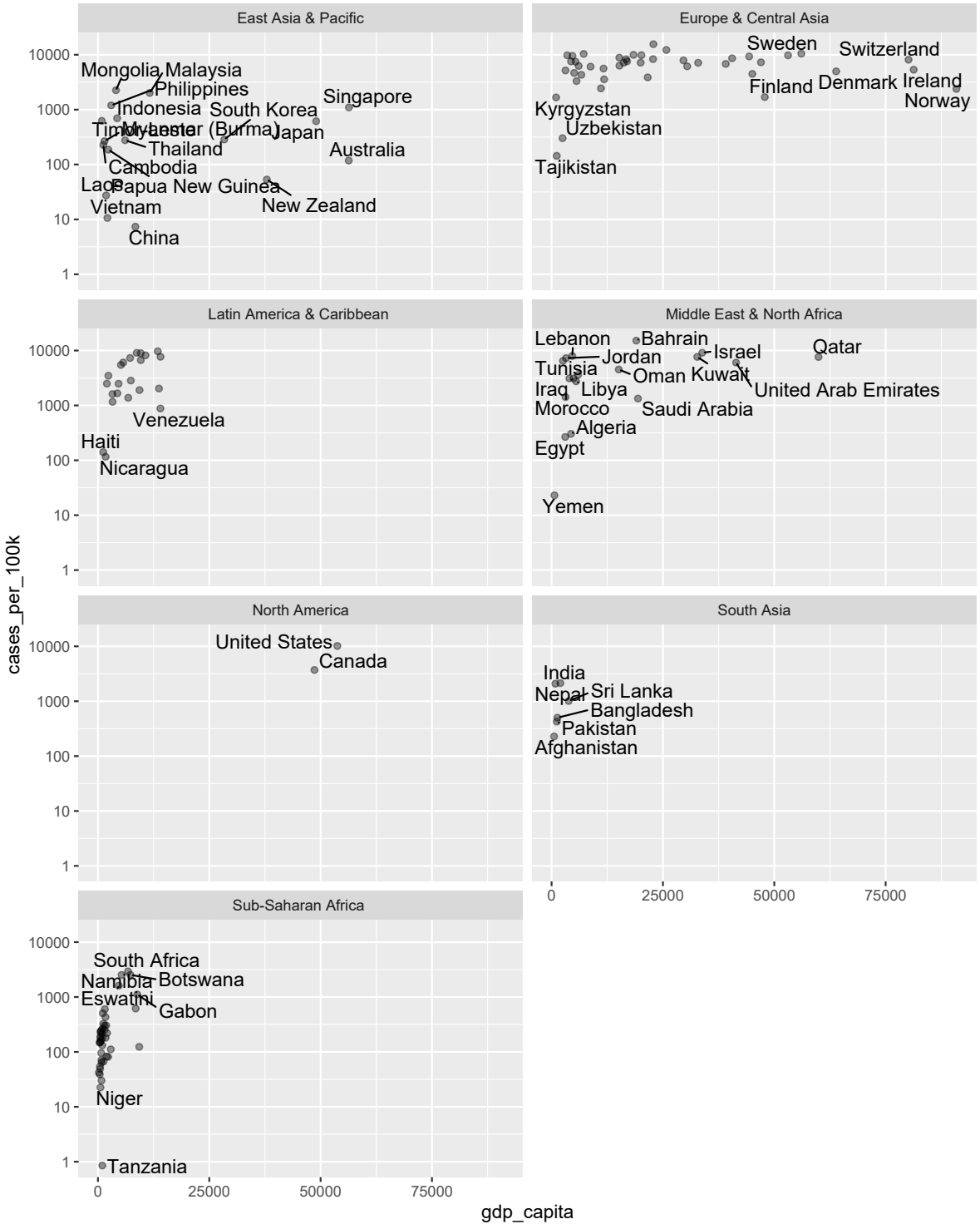
```
## Warning: Removed 3 rows containing missing values
## (geom_text_repel).
```

```
## Warning: ggrepel: 19 unlabeled data points
## (too many overlaps). Consider increasing
## max.overlaps
```

```
## Warning: ggrepel: 36 unlabeled data points
## (too many overlaps). Consider increasing
## max.overlaps
```

```
## Warning: ggrepel: 35 unlabeled data points
## (too many overlaps). Consider increasing
## max.overlaps
```

```
## Warning: ggrepel: 2 unlabeled data points
## (too many overlaps). Consider increasing
## max.overlaps
```


East Asia & Pacific
Europe & Central Asia
Latin America & Caribbean
Middle East & North Africa
North America
South Asia
Sub-Saharan Africa
cases_per_100k
gdp_capita
10000
1000
100
10
1
0
25000
50000
75000
Mongolia
Malaysia
Philippines
Indonesia
South Korea
Singapore
Timor-Leste
Myanmar (Burma)
Japan
Thailand
Australia
Cambodia
Lao
Papua New Guinea
Vietnam
New Zealand
China
Sweden
Switzerland
Finland
Denmark
Ireland
Norway
Kyrgyzstan
Uzbekistan
Tajikistan
Venezuela
Haiti
Nicaragua
Lebanon
Bahrain
Israel
Qatar
Jordan
Tunisia
Oman
Kuwait
Iraq
Libya
United Arab Emirates
Morocco
Saudi Arabia
Algeria
Egypt
Yemen
United States
Canada
India
Nepal
Sri Lanka
Bangladesh
Pakistan
Afghanistan
South Africa
Namibia
Botswana
Eswatini
Gabon
Niger
Tanzania

6.2 `facet_grid` multiple-panel plots based on two variables

The `facet_grid()` function does a similar thing but instead of creating different plots it creates different grids and then plots each plot in the grids. The data can be split up by one or two variables that vary on the horizontal and/or vertical direction. This is done by giving a formula to `facet_grid()`, of the form vertical ~ horizontal.

```
dat %>%
  mutate(income = factor(income, levels = c("Low income",
                                            "Lower middle income",
                                            "Upper middle income",
                                            "High income"))) %>%
ggplot(aes(x = tests_per_capita, y = cases_per_100k)) +
  geom_point(size = 1.5, color = "red", alpha = 0.4) +
  geom_text_repel(aes(label = country),
                  min.segment.length = 0, seed = 42,
                  box.padding = 0.1, color = "gray30",
                   segment.size = 0.2, size = 3) +
  scale_y_continuous(trans = "log10") +
  facet_grid(region~income) +
  theme(strip.background = element_blank(),
        strip.text.y = element_text(angle = 0))
```

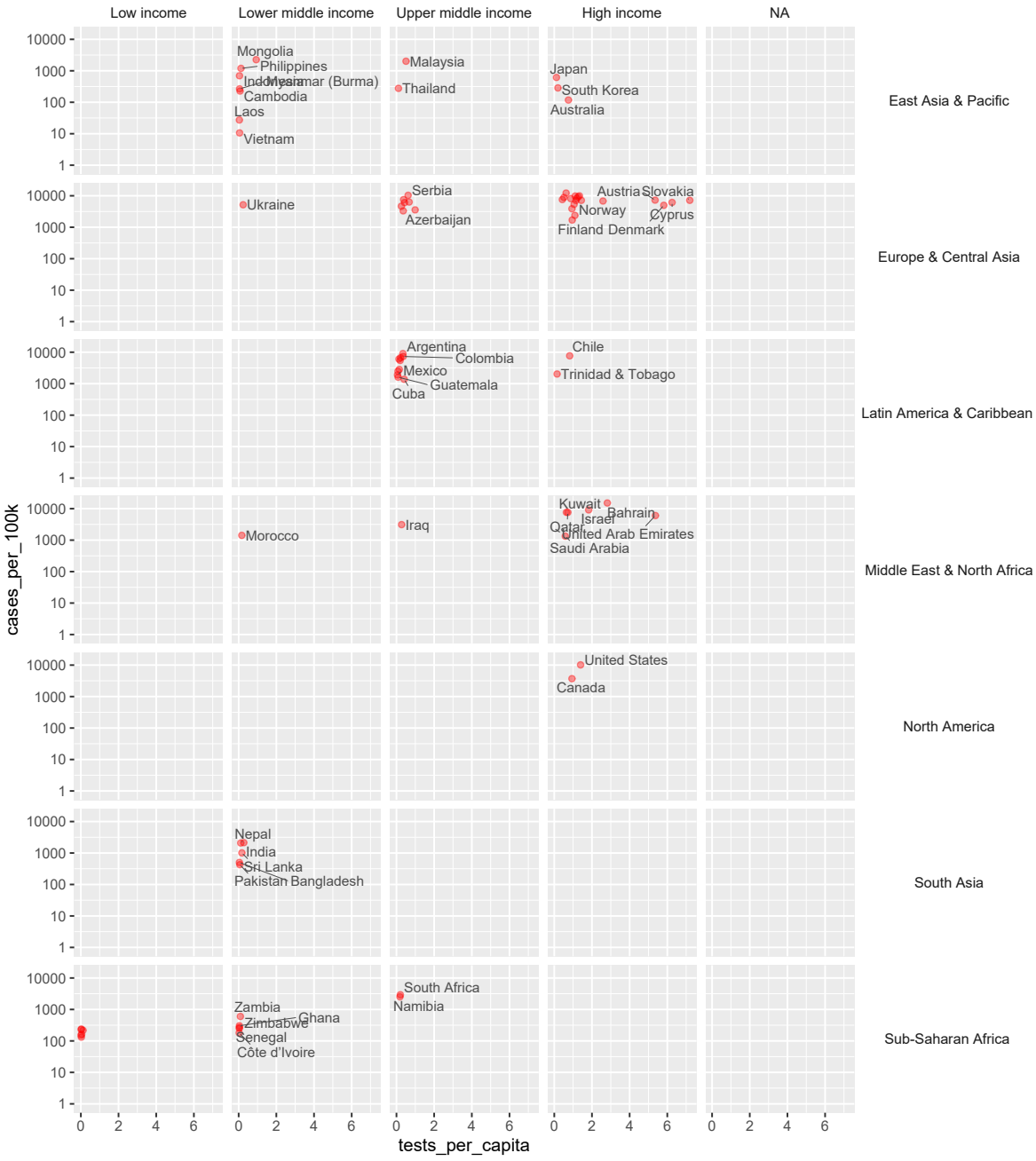
```
## Warning: Removed 76 rows containing missing values
## (geom_point).
```

```
## Warning: Removed 76 rows containing missing values
## (geom_text_repel).
```

```
## Warning: ggrepel: 6 unlabeled data points
## (too many overlaps). Consider increasing
## max.overlaps
```

```
## Warning: ggrepel: 5 unlabeled data points
```

```
## (too many overlaps). Consider increasing
## max.overlaps

## Warning: ggrepel: 5 unlabeled data points
## (too many overlaps). Consider increasing
## max.overlaps

## Warning: ggrepel: 13 unlabeled data points
## (too many overlaps). Consider increasing
## max.overlaps
```



Note that the lines which come before the `ggplot()` function are piped %>% whereas from `ggplot()` onwards we have to use +. This is because `ggplot2` package was written before the pipe %>% was introduced. The + sign in `ggplot2` functions (similar to the pipe %>% in other functions in the tidyverse) allows code to be written from left to right adding different layers and customisations to the same plot.

Moreover, as commands get longer, we suggest to add carriage returns (new lines),

which must be inserted after the %>% or + symbols. In most cases, R is blind to white space and new lines, so this is a simple way to make our code more readable.

7 Add custom fonts in `ggplot2` plots

When creating graphics in R you can specify a font family. The integrated fonts in R are sans (`Arial`), serif (`Times New Roman`), mono (`Courier`) and symbol (`Standard Symbols L`). However, it is possible to add custom fonts in R with the `extrafont` package.

The `extrafont` package allows adding True Font Type (.ttf) fonts (the major type of font found in both Mac and Microsoft Windows operating systemsto) to R easily. Before loading the package, it is recommended to install all the fonts we want in our system. For that purpose we will need to download them from any source and install them.

7.1 Import the system custom fonts

Once we have all the desired fonts installed, the first time after installing the package we will need to run the `font_import` function, which will import all the `.ttf` fonts of our system.

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

# Import all the .ttf files from your system
# We will only need to run this once, but it will take
# a few minutes to finish
font_import()
```

After running the previous function, we can see the full list of fonts available in alphabetical order running `fonts()`.

```
# Show the full list
# fonts()

# Show only the first six fonts
head(fonts())
```

```
## [1] "Agency FB"  "Algerian"   "AR BERKLEY"
## [4] "AR BLANCA"  "AR BONNIE"  "AR CARTER"
```

7.2 Using the custom fonts

After importing all the fonts, we will need to register them with the `loadfonts()` function. We will need to run the following code once on each R session:

```
loadfonts()
```

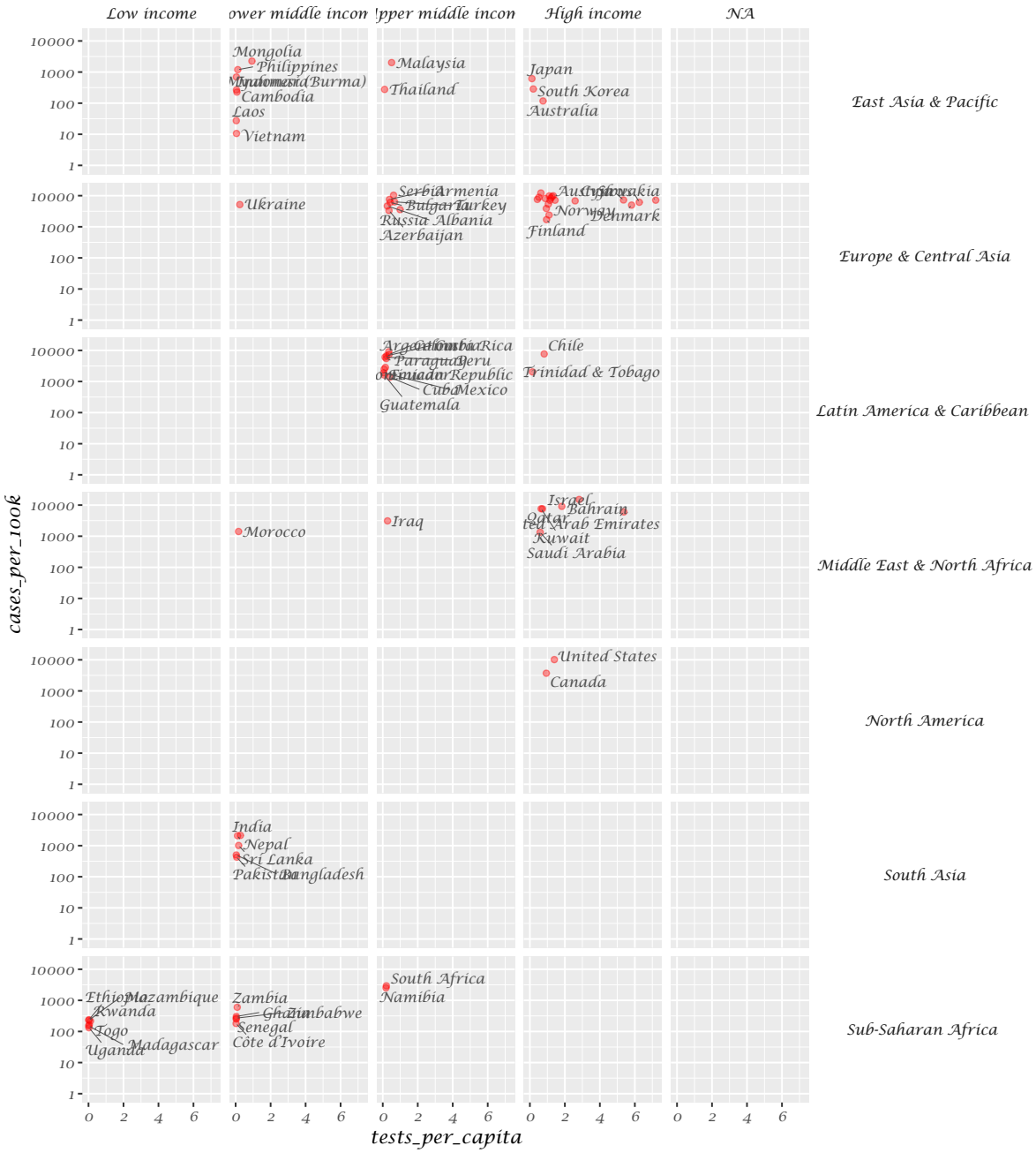
Now, we will be able to specify the fonts on the corresponding argument of the graphical function we are going to use with the name appearing on the `fonts()` list that corresponds to the font we want to set.

In `ggplot2` text functions (e.g., geom_text_repel) or/and the text option in the `theme()` we can specify the `family` argument:

```
dat %>%
  mutate(income = factor(income, levels = c("Low income",
                                            "Lower middle income",
                                            "Upper middle income",
                                            "High income"))) %>%
ggplot(aes(x = tests_per_capita, y = cases_per_100k)) +
  geom_point(size = 1.5, color = "red", alpha = 0.4) +
  geom_text_repel(aes(label = country),
                  min.segment.length = 0, seed = 42,
                  box.padding = 0.1, color = "gray30",
                  segment.size = 0.2, size = 3,
```

```
                  max.overlaps = 17,
                  family = "Lucida Calligraphy") +
  scale_y_continuous(trans = "log10") +
  facet_grid(region~income) +
  theme(strip.background = element_blank(),
        strip.text.y = element_text(angle = 0),
        text = element_text(family = "Lucida Calligraphy"))
```

```
## Warning: Removed 76 rows containing missing values
## (geom_point).

## Warning: Removed 76 rows containing missing values
## (geom_text_repel).

## Warning: ggrepel: 13 unlabeled data points
## (too many overlaps). Consider increasing
## max.overlaps
```


Low income
ower middle incon
pper middle incon
High income
NA
East Asia & Pacific
Europe & Central Asia
Latin America & Caribbean
Middle East & North Africa
North America
South Asia
Sub-Saharan Africa
cases_per_100k
tests_per_capita
Mongolia
Philippines
Malaysia
Japan
Thailand
South Korea
Cambodia
Australia
Laos
Vietnam
Ukraine
Armenia
Turkey
Russia
Albania
Azerbaijan
Finland
Denmark
Chile
Trinidad & Tobago
Cuba
Mexico
Guatemala
Iraq
Morocco
Israel
Bahrain
Qatar
Kuwait
Saudi Arabia
United States
Canada
India
Nepal
Sri Lanka
Pakistan
Bangladesh
South Africa
Namibia
Zambia
Zimbabwe
Senegal
Côte d'Ivoire
Rwanda
Togo
Uganda
Madagascar

8 Practical example: the graph everyone wants to draw

8.1 Cumulative Number of Deaths from COVID-19

Seeing the total number of deaths over time, on a country-by-country basis, can illustrate how the pandemic is expanding. Because the epidemic began at different times in different countries, viewing each country's curve from the same starting point can allow us to more easily compare countries. The starting point for this chart is the day on which the 10th death was confirmed in each country, with the trend lines following the number of days since that event.

Let's say we want to focus on data of specific countries:

```
focus_cn <- c("CHN", "GBR", "USA", "JPN", "GRC", "MEX",
              "KOR", "ITA", "ESP", "BRA", "IND")
```

First, we have to prepare the data:

```
covid_deaths <- covid_data %>%
  select(date, iso3c, deaths) %>%
  group_by(iso3c) %>%
  arrange(date) %>%
  filter(deaths > 10) %>%
  mutate(days_elapsed = date - min(date),
         end_label = ifelse(date == max(date), iso3c, NA),
         end_label = case_when(iso3c %in% focus_cn ~ end_label,
                               TRUE ~ NA_character_),
         cgroup = case_when(iso3c %in% focus_cn ~ iso3c,
                            TRUE ~ "OTHER")) %>%
  ungroup()

covid_deaths
```

```
## # A tibble: 87,478 x 6
```

```
##    date       iso3c deaths days_elapsed
##    <date>     <chr>  <dbl> <drtn>
##  1 2020-01-22 CHN       17 0 days
##  2 2020-01-23 CHN       18 1 days
##  3 2020-01-24 CHN       26 2 days
##  4 2020-01-25 CHN       42 3 days
##  5 2020-01-26 CHN       56 4 days
##  6 2020-01-27 CHN       82 5 days
##  7 2020-01-28 CHN      131 6 days
##  8 2020-01-29 CHN      133 7 days
##  9 2020-01-30 CHN      171 8 days
## 10 2020-01-31 CHN      213 9 days
## # ... with 87,468 more rows, and 2 more
## #   variables: end_label <chr>, cgroup <chr>
```

We also set particular colors for the selected countries:

```
## Colors
cgroup_cols <- c(prismatic::clr_darken(
  paletteer_d("ggsci::category20_d3"), 0.2)[1:length(focus_cn)],
  "gray70")
```

Now we are ready to create the chart with the cumulative number of confirmed deaths:

```
death_curves <- covid_deaths %>% filter(cgroup != "OTHER") %>%
    ggplot(mapping = aes(x = days_elapsed, y = deaths,
                         color = cgroup, label = end_label,
                         group = iso3c)) +
    geom_line(size = 0.8) +
    geom_text_repel(nudge_x = 0.2,
                    nudge_y = 0.1, size = 3,
                    segment.color = NA) +
    guides(color = FALSE) +
    scale_color_manual(values = cgroup_cols) +
```

```
    scale_y_continuous(labels = scales::comma_format(accuracy = 1),
                       limits = c(10, 6.1*10^5),
                       breaks = seq(0, 6.*10^5, 10^5)) +
    labs(x = "Days Since 10th Confirmed Death",
         y = "Cumulative Number of Deaths",
         title = "Cumulative Number of Deaths from COVID-19",
         subtitle = paste("Data as of", format(max(covid_deaths$date),
                                               "%d/%m/%y")),
         caption = "Data: https://coronavirus.jhu.edu/map.html")
```

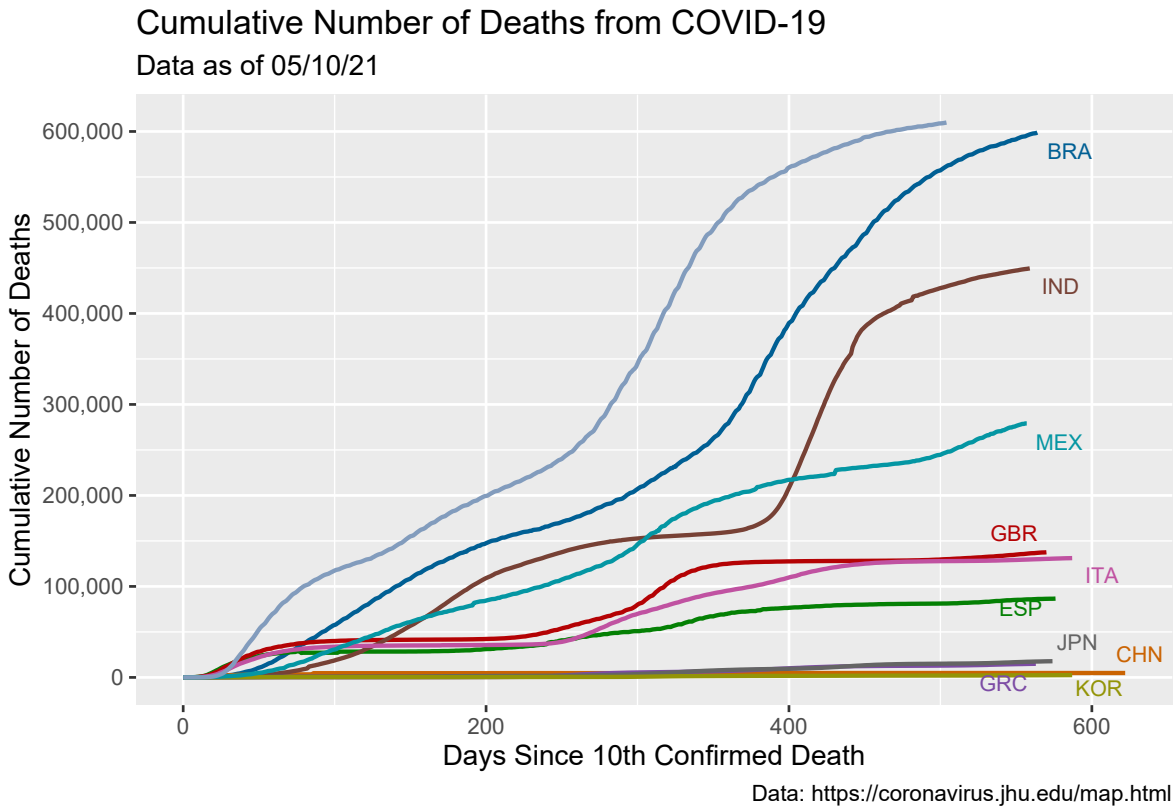
```
## Warning: `guides(<scale> = FALSE)` is
## deprecated. Please use `guides(<scale> =
## "none")` instead.
```

```
death_curves
```

```
## Don't know how to automatically pick scale for object of type difftime. Defaulting
```

```
## Warning: Removed 76 row(s) containing missing
## values (geom_path).
```

```
## Warning: Removed 6340 rows containing missing
## values (geom_text_repel).
```



8.2 Cumulative Number of Deaths from COVID-19 in logarithmic scale

Next, we present the chart with the cumulative number of confirmed deaths in logarithmic scale:

```
death_log_curves <- covid_deaths %>% filter(cgroup != "OTHER") %>%
    ggplot(mapping = aes(x = days_elapsed, y = deaths,
                         color = cgroup, label = end_label,
                         group = iso3c)) +
    geom_line(size = 0.8) +
    geom_text_repel(nudge_x = 0.2,
                    nudge_y = 0.1, size = 3,
                    segment.color = NA) +
    guides(color = FALSE) +
```

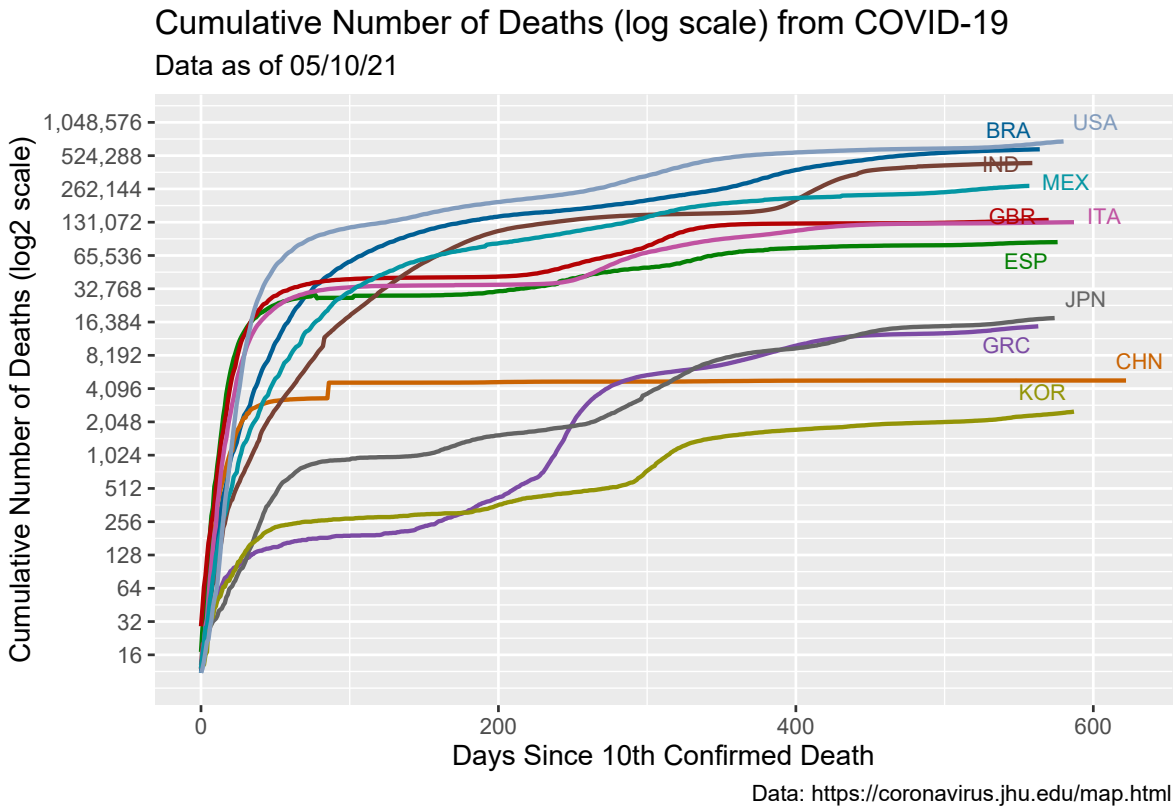
```
    scale_color_manual(values = cgroup_cols) +
    scale_y_continuous(labels = scales::comma_format(accuracy = 1),
                       limits = c(10, 2^20),
                       breaks = 2^seq(4, 20),
                       trans = "log2") +
    labs(x = "Days Since 10th Confirmed Death",
         y = "Cumulative Number of Deaths (log2 scale)",
         title = "Cumulative Number of Deaths (log scale) from COVID-19",
         subtitle = paste("Data as of", format(max(covid_deaths$date),
                                               "%d/%m/%y")),
         caption = "Data: https://coronavirus.jhu.edu/map.html")
```

```
## Warning: `guides(<scale> = FALSE)` is
## deprecated. Please use `guides(<scale> =
## "none")` instead.
```

```
death_log_curves
```

```
## Don't know how to automatically pick scale for object of type difftime. Defaulting
```

```
## Warning: Removed 6339 rows containing missing
## values (geom_text_repel).
```



This chart shows the cumulative number of confirmed deaths since the 10th confirmed death for each country. An upward bend in a curve can indicate either a time of explosive growth of coronavirus cases in a given country or a change in how deaths are defined or counted. Comparing across countries can also show where the pandemic is growing most rapidly at any point in time.

Limited testing and challenges in the attribution of the cause of death means that the number of confirmed deaths may not be an accurate count of the true number of deaths from COVID-19.

9 Combine several plots into one figure

9.1 Assemble a number of plots

Sometimes, we would like to combine several plots into one figure. This can be easily done with the `patchwork` package. For example, if we want to combine two

`ggplot2` objects, say `death_curves` and `death_log_curves`, then we can directly call `death_curves + death_log_curves` to combine the two objects. Since the two plots share the same elements, patchwork allows us to use e.g. one top title, subtitle and tags by calling function `plot_annotation`.

```
death_curves + death_log_curves +
  plot_annotation(title = 'Cumulative Number of Deaths',
                  subtitle = 'A: Linear scale; B: Log scale',
                  tag_levels = 'A')
```

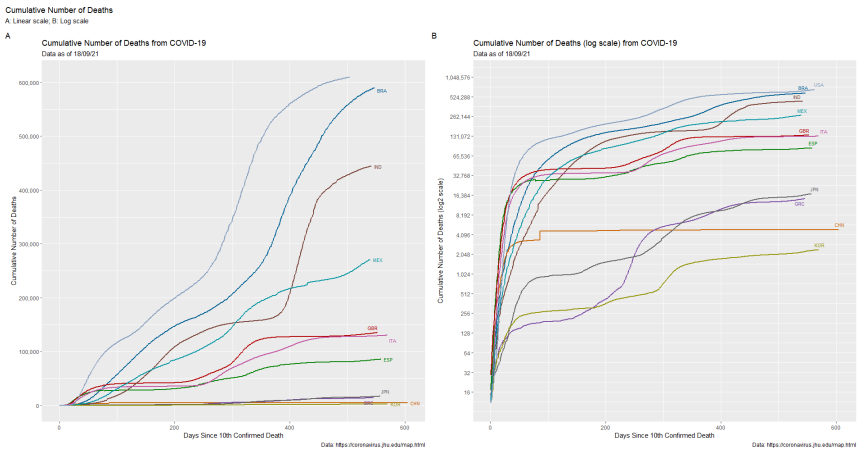


Figure 13: Cumulative Number of Deaths

9.2 Insets

Another approach is to use the `inset_element()` function which marks a plot or graphic object to be placed as an inset on the previous plot. It will thus not take up a slot in the provided layout, but share the slot with the previous plot. `inset_element()` allows us to freely position our inset relative to either the panel, plot, or full area of the previous plot, by specifying the location of the left, bottom, right, and top edge of the inset.

```
death_curves + inset_element(death_log_curves,
                             left =0.02, right =0.4,
                             bottom = 0.45, top =0.98)
```

We can save our plot in a folder, for example, named `figures`:

```
ggsave(
  here::here("figures",
             paste0("figure_name", format(Sys.time(), "%Y%m%d_%H%M%S"),
                    ".tiff")),
       type = "cairo",
       width = 13, height = 9, dpi = 320,
       compression = "lzw")
```

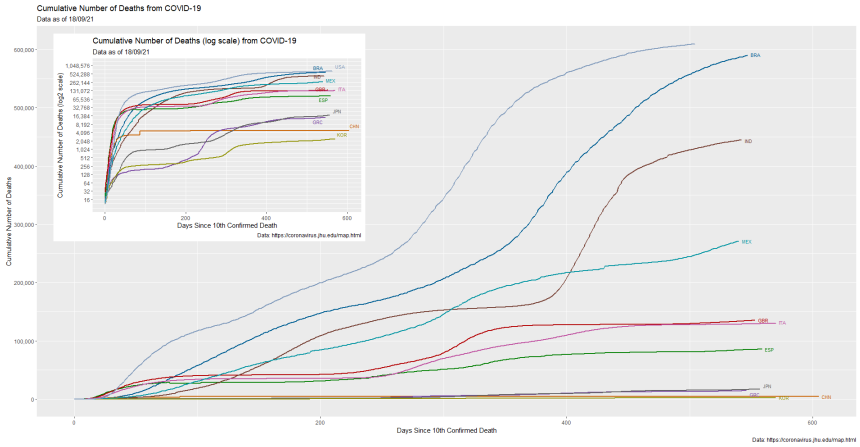


Figure 14: Cumulative Number of Deaths

10 Animated plots

```
animate1 <- covid_deaths %>%
  filter(iso3c %in% focus_cn) %>%
  ggplot(mapping = aes(x = date, y = deaths, group= iso3c,
```

```
                        color = iso3c)) +
  geom_path(size = 1) +
  geom_point() +
  geom_text(aes(label = iso3c), size = 5) +
  scale_y_continuous(labels = scales::comma_format(accuracy = 1),
                        limits = c(10, 2^20),
                        breaks = 2^seq(4, 20),
                        trans = "log2") +
  scale_color_manual(values = cgroup_cols) +
  labs(title = "Date: {round(frame_along, 0)}",
       x = 'Calendar time',
       y = 'Cumulative Number of Deaths (log2 scale)') +
  theme_minimal() +
  theme(legend.position = "none") +
  transition_reveal(date)

animate1
```

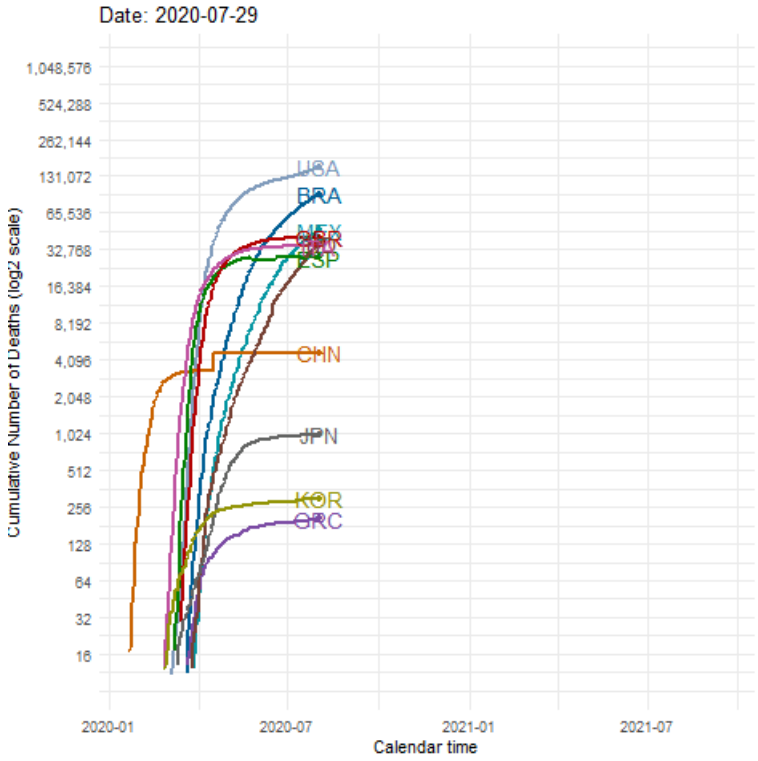


Figure 15: Animated cumulative Number of Deaths

We can export the animation as a giff:

```
anim_save(here("figures", "animate_covid.gif"), animate1)
```

11 Interactive plots

11.1 Plotly

Any graph made with the plotly R package is powered by the JavaScript library `plotly.js1`. The `plot_ly()` function provides a 'direct' interface to `plotly.js` with some additional abstractions to help reduce typing. These abstractions, inspired by the Grammar of Graphics and `ggplot2`, make it much faster to iterate from one graphic to another, making it easier to discover interesting features in the data.

`Plotly` allows the user to create high quality, interactive graphs. This includes scatter plots, histograms, heatmaps and many more! To demonstrate, we'll use it to explore the covid_data and learn a bit how `plotly` works along the way.

`Plotly` is an extensive package and we will merely scratch the surface of its capabilities in this course. More information about the package can be found at the website: Plotly

A Plotly chart is created by using tree set of functions:

- `plot_ly()`, can be thought of as the base which allows R objects to be mapped to the Plotly library. It is similar to `ggplot()` function.
- `layout()`, is used to control the chart title, axis labels, legends, canvas, range and scales.
- `add_trace()` (`add_*`), creates a geometry layer called a trace which is added to the chart. It defines the type of plot similar to geometries(geom_*) in ggplot.

Note that multiple traces can be added on one plot. There are many add_* functions. Running the following code provides a list of these functions:

```
stringr::str_subset(objects("package:plotly"), pattern ="^add_")
```

```
##  [1] "add_annotations"
##  [2] "add_area"
##  [3] "add_bars"
##  [4] "add_boxplot"
##  [5] "add_choropleth"
##  [6] "add_contour"
##  [7] "add_data"
##  [8] "add_fun"
##  [9] "add_heatmap"
## [10] "add_histogram"
## [11] "add_histogram2d"
## [12] "add_histogram2dcontour"
## [13] "add_image"
```

```
## [14] "add_lines"
## [15] "add_markers"
## [16] "add_mesh"
## [17] "add_paths"
## [18] "add_pie"
## [19] "add_polygons"
## [20] "add_ribbons"
## [21] "add_scattergeo"
## [22] "add_segments"
## [23] "add_sf"
## [24] "add_surface"
## [25] "add_table"
## [26] "add_text"
## [27] "add_trace"
```

Let's see an example of a simple interactive plot:

```
plotly1 <- dat %>%
  plot_ly(x = ~gdp_capita, y = ~life_expectancy,
          color = ~region)

plotly1
```

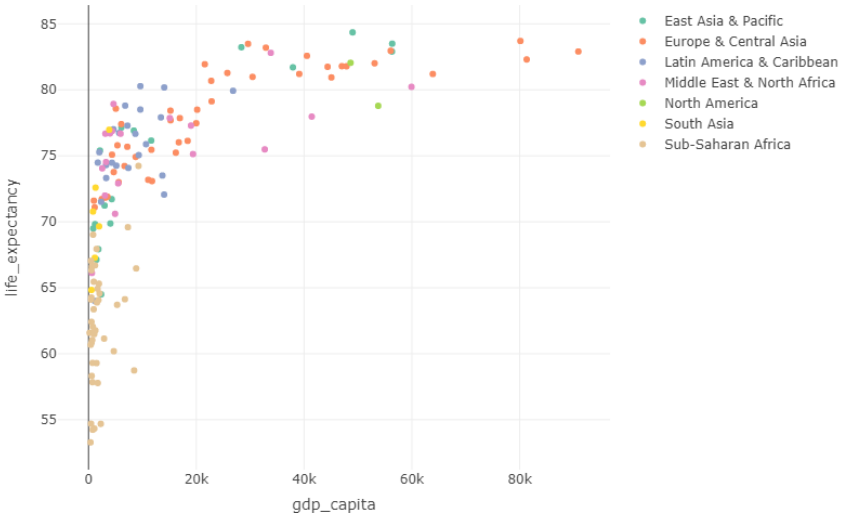


Figure 16: Plotly 1

Although we did not specify the plot type, the produced chart is a scatter plot. The `plot_ly()` function is the base plotly command to initialize a plot from a dataframe, similar to `ggplot()` from ggplot2. Therefore, we observe that if we assign variable names (e.g., gdp_capita, life_expectancy, region, etc.) to visual properties (e.g., x, y, color, etc.) within `plot_ly()`, it tries to find a sensible geometric representation of that information for us.

Analytically, the type of plot is specified by setting the **trace** type. The scatter trace type is the foundation for many low-level geometries (e.g., points, lines, and text), thus we must also specify a mode. To create a scatter plot with points the mode is set to markers, but additional scatter modes include lines, paths, segments, ribbons, polygons, and text.

The plot's interactivity can be accessed via its tool bar. The functionality provided by this toolbar from left to right are as follows:

- Downloading the plot as a png file.
- Zooming in specific areas of the plot
- Panning across the map.
- Selecting all points using a box.
- Selecting all points using a lasso.

- Zooming in and out on

Moreover, hovering over individual points displays their coordinates.

Plotly functions take a plotly object as an input and return a modified plotly object, making it work perfectly with the **pipe operator** (%>%). Note that `plot_ly()` uses a `~` syntax for mapping aesthetics that is a bit different from the `ggplot()` syntax we've presented earlier.

We can explicitly define the type and mode of the plot using the `add_trace()`. Additionally, we can change the palette of colors and use the layout function to set title and axis labels. For the axis labels we use the `xaxis` and `yaxis` arguments which require lists:

```
plotly2 <- dat %>%
  plot_ly(x = ~gdp_capita, y = ~life_expectancy,
          color=~region) %>%
  add_trace(type = "scatter", mode = "markers", colors = "Set1") %>%
  layout(title = "Life expectancy vs. GDP per capita",
         xaxis = list(
           title = "GDP per capita ($)"),
         yaxis = list(
           title = "Life expectancy (years)")
         )

plotly2
```

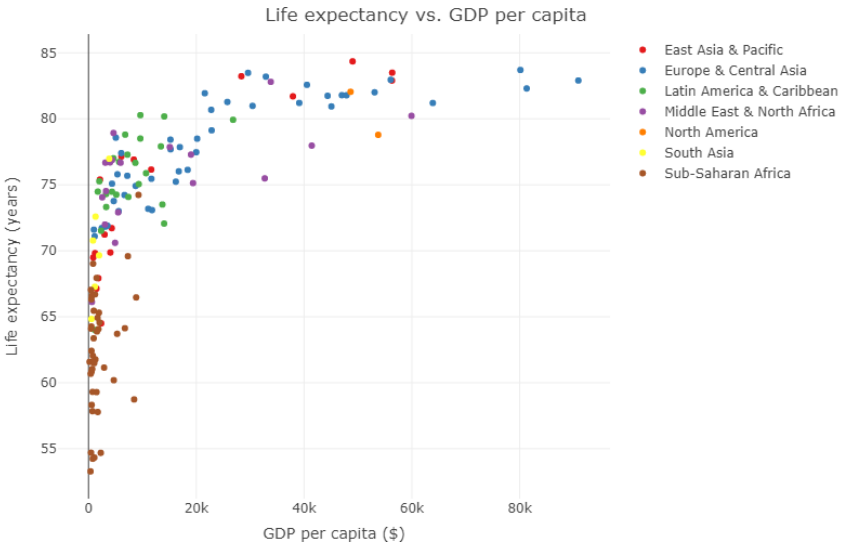


Figure 17: Plotly 2

Rather than using add_trace() and specifying the type and mode, we can use the convenience function add_markers() and change the "filled" circles with "open" circles. Note that if we want to use a constant value for an aesthetic (e.g., symbol), we must specify that the argument should be used "as-is," using the `I()` function. Another important attribute that can be added is the name of the country using the `text` argument in the `plot_ly()`.

```
plotly3 <- dat %>%
  plot_ly(x = ~gdp_capita, y = ~life_expectancy,
          color=~region,
          text = ~country) %>%
  add_markers(symbol = I("circle-open"), colors = "Set1") %>%
  layout(title = "Life expectancy vs. GDP per capita",
         xaxis = list(
           title = "GDP per capita ($)"),
         yaxis = list(
           title = "Life expectancy (years)")
         )

plotly3
```

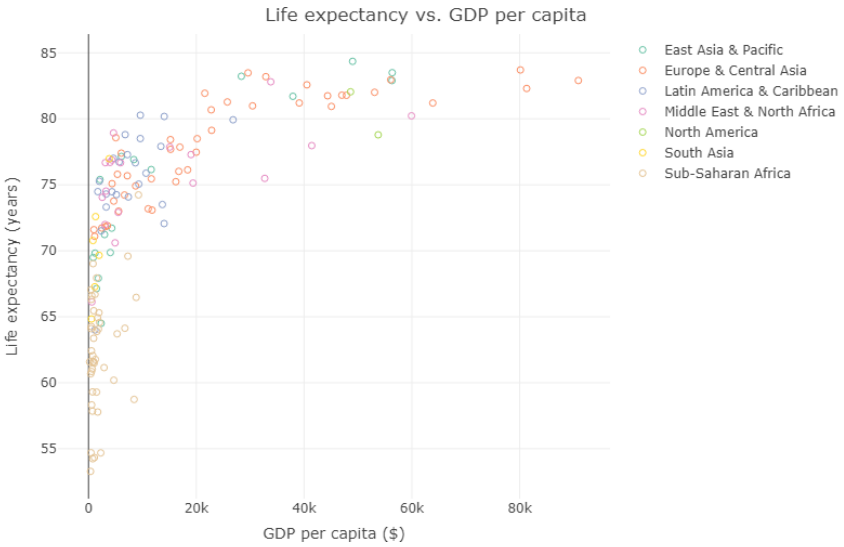


Figure 18: Plotly 3

Making other plot types is similarly easy by using the corresponding add_*() function.

We can also create 3D interactive plots using the `scatter3d` type as follows:

```
plotly4 <- dat %>%
  plot_ly(x = ~gdp_capita, y = ~life_expectancy,
          z= ~pub_health, color=~region,
          text = ~ country, size = I(200)) %>%
  add_trace(type = "scatter3d", mode = "markers",
            symbol = I("circle-open"), colors = "Set1") %>%
  layout(scene = list(
    xaxis = list(
      title = "x:GDP per capita ($)"),
    yaxis = list(
      title = "y:Life expectancy (years)"),
    zaxis = list(
      title = "z:No. health measures")
  ))

plotly4
```

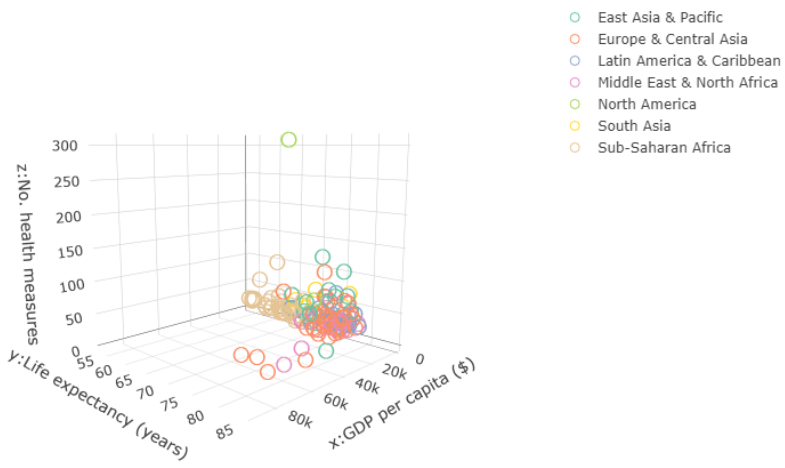


Figure 19: Plotly 4

We can save our widget 3D plot to an HTML file (e.g., for sharing with others):

```
htmlwidgets::saveWidget(widget = plotly4 , here("figures", "hc.html"))
```

We can use the `paste()` function to create a more customized text label. Use HTML tags for any formatting. For example, to show both the variables in a more attractive format, we could run:

```
plotly5 <- dat %>%
  plot_ly(x = ~gdp_capita, y = ~life_expectancy,
          z= ~pub_health, color=~region,
          size = I(200)) %>%
  add_trace(type = "scatter3d", mode = "markers",
            symbol = I("circle-open"), colors = "Set1",
            text = ~ paste("<b>Country:</b> ", country, "<br />",
"<b>GDP per capita ($):</b> ", round(gdp_capita, digits=1),"<br />",
"<b>Life expectancy (yrs):</b> ", round(life_expectancy, digits=1),"<br />",
"<b>No. health measures:</b> ", pub_health
                          ), hoverinfo = "text") %>%
  layout(scene = list(
    xaxis = list(
```

```
      title = "x:GDP per capita ($)"),
    yaxis = list(
      title = "y:Life expectancy (years)"),
    zaxis = list(
      title = "z:No. health measures")
  ))

plotly5
```

11.2 Integration with ggplot2: `ggplotly()`

A ggplot object can be transformed into an interactive plot by calling the function `ggplotly()`.

```
g_plot <- ggplot(dat, aes(x = gdp_capita, y = life_expectancy)) +
  geom_point(aes(color = region)) +
  geom_text(aes(label = country)) +
labs(title = "Life expectancy vs. GDP per capita",
         xaxis = list(
           title = "GDP per capita ($)"),
         yaxis = list(
           title = "Life expectancy (years)")
         )

ggplotly(g_plot)
```

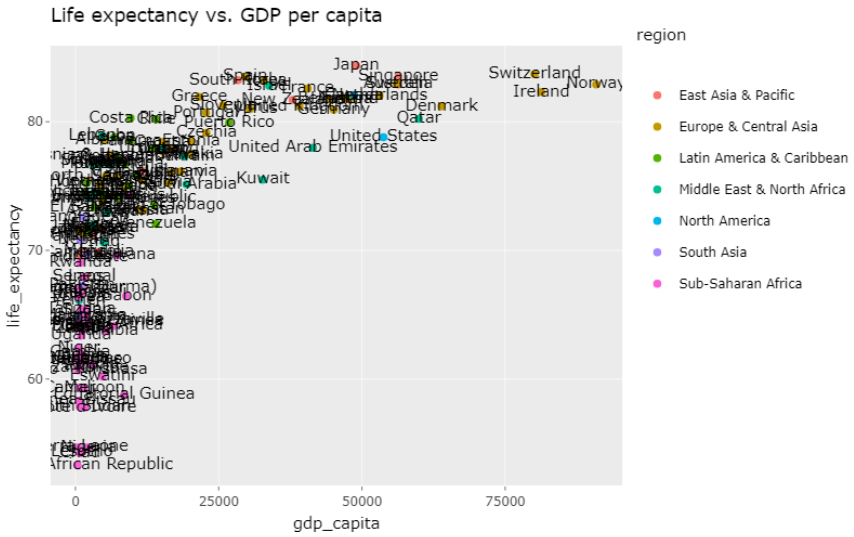


Figure 20: ggplotly

11.3 Highchartrers

```
highchart1 <- dat %>%
hchart("scatter", hcaes(x = gdp_capita, y = life_expectancy,
                        group = region)) %>%
  hc_xAxis(title = list(text="GDP per capita ($)")) %>%
  hc_yAxis(title = list(text="Life expectancy (years)")) %>%
  hc_add_theme(hc_theme_economist())

highchart1
```

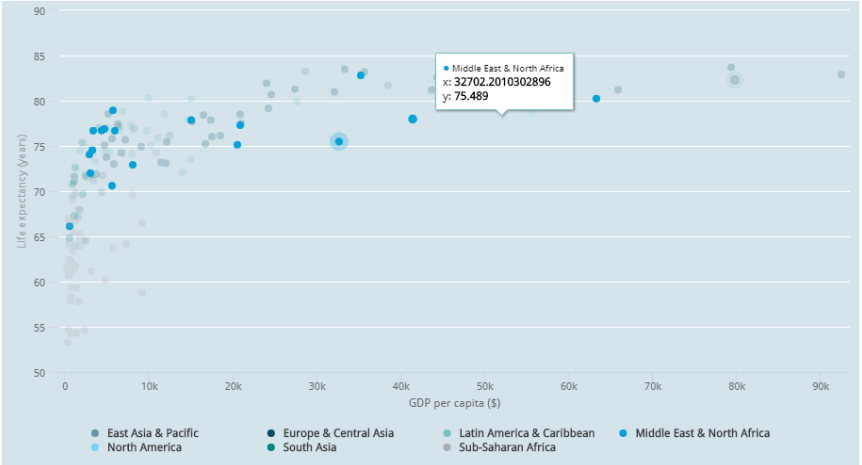


Figure 21: highchart1

11.4 c3

```
c3 <- dat %>%
  drop_na(gdp_capita, life_expectancy, region) %>%
  mutate(across(
    .cols = c(gdp_capita, life_expectancy),
    round, digits =0)) %>%
  c3(x = "gdp_capita",
     y = "life_expectancy",
     group = "region") %>%
  c3_scatter()

c3
```

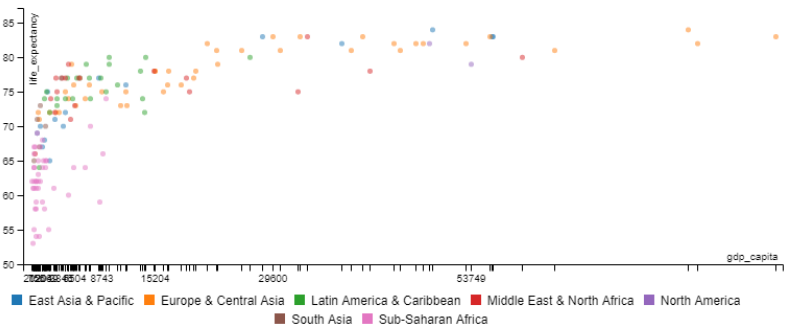


Figure 22: c3

11.5 ScatterD3

The `scatterD3` package provides an HTML widget based on the htmlwidgets package and allows to produce interactive scatterplots by using the `d3` javascript visualization library.

```
scat3D <- scatterD3(data = data.frame(dat),
                    x = gdp_capita, y = life_expectancy,
                    col_var = region,
                    lab = country)

scat3D
```

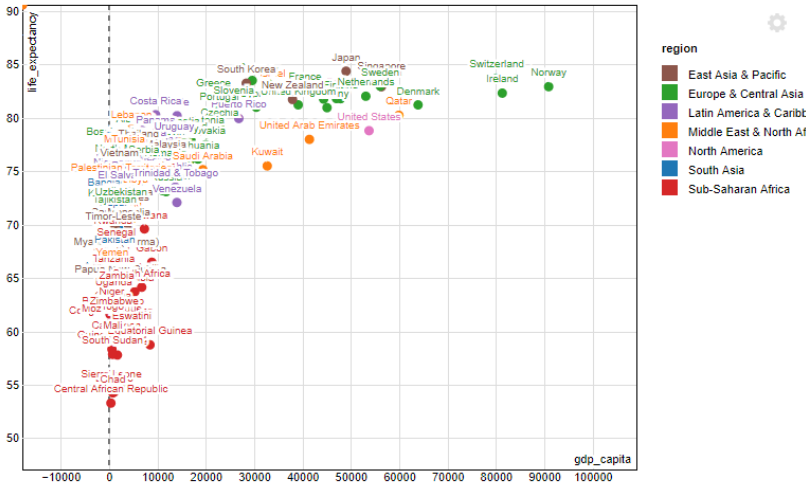


Figure 23: scat3D

This will display a simple visualization with the given variables. There are several interactive features directly available :

- we can **zoom** in and out with the mouse wheel while the mouse cursor is on the plot
- we can ***pan** the plot by dragging with your mouse
- **hovering** over a point displays a small tooltip window giving the variable values

11.6 Linked data visualizations in R with ggiraph

11.6.1 Prepare the data

For data, we are going to use recent US Covid vaccination data by state. In the code below, we are reading in the vaccination data, and changing “New York State” to just “New York” in the data frame.

```
data_url <- "https://github.com/owid/covid-19-data/raw/master/public/
data/vaccinations/us_state_vaccinations.csv"
all_data <- read.csv(data_url)
all_data$location[all_data$location == "New York State"] <- "New York"
```

Note that we can also use the `case_when()` which is a `dplyr` function (how?).

Next, we create a vector of entries that aren't US states or DC (District of Columbia). We will use it to filter out that data so our chart doesn't have too many rows.

```
not_states_or_dc <- c("American Samoa", "Bureau of Prisons",
                      "Dept of Defense", "Federated States of Micronesia",
                      "Guam", "Indian Health Svc", "Long Term Care",
                      "Marshall Islands", "Northern Mariana Islands",
                      "Puerto Rico", "Republic of Palau", "United States",
                      "Veterans Health", "Virgin Islands")
```

This next code filters out the `non_states_or_dc` rows, chooses only the most recent data, rounds the percent vaccinated to one decimal point, selects only the state and percent vaccinated columns, and renames our selected columns to `State` and `PctFullyVaccinated`.

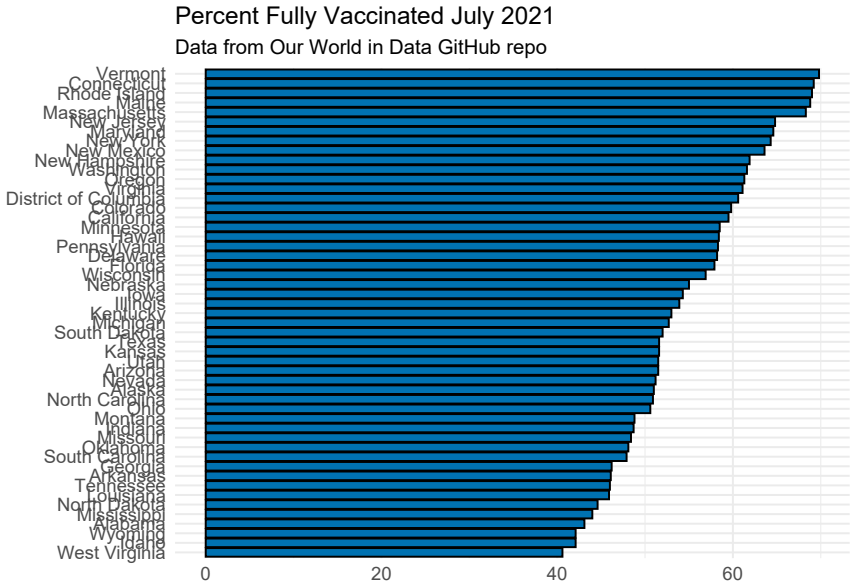
```
bar_graph_data_recent <- all_data %>%
  filter(date == max(date), !(location %in% not_states_or_dc)) %>%
  mutate(PctFullyVaccinated = round(people_fully_vaccinated_per_hundred, 1)) %>%
  select(State = location, PctFullyVaccinated)
```

11.6.2 Create a basic bar graph with ggplot2

Next we willl create a basic (static) ggplot bar chart of the data. We use `geom_col()` for a bar chart, add our own customary blue bars outlined in black and minimal theme, set the axis text size to 10 points, and flip the x and y coordinates so it's easier to read the state names.

```
bar_graph <- bar_graph_data_recent %>%
  ggplot(aes(x = PctFullyVaccinated,
             y = reorder(State, PctFullyVaccinated))) +
  geom_col(color = "black", fill="#0072B2", size = 0.5) +
  theme_minimal() +
```

```
  theme(axis.text=element_text(size = 10),
        axis.title = element_blank()) +
  labs(title = "Percent Fully Vaccinated July 2021",
       subtitle = "Data from Our World in Data GitHub repo"
  )

bar_graph
```



11.6.3 Create a tooltip column in R

`ggiraph` only lets us use one column for the tooltip display, but we want both state and rate in our tooltip. There's an easy solution: Add a tooltip column to the data frame with both state and rate in one text string:

```
bar_graph_data_recent <- bar_graph_data_recent %>%
  mutate(tooltip_text = paste0(toupper(State), "\n",
                    PctFullyVaccinated, "%"))
```

11.6.4 Make the bar chart interactive with ggiraph

To create a `ggiraph` interactive bar chart, we changed `geom_col()` to `geom_col_interactive()` and added tooltip and `data_id` to the `aes()` mapping. We also reduced the size of the axis text, because the `ggplot` size ended up being too large.

Then we displayed the interactive graph object with the `girafe()` function. We can set the graph width and height with `width_svg` and `height_svg` arguments within `girafe()`.

```
latest_vax_graph <- bar_graph_data_recent %>%
  ggplot(aes(x = PctFullyVaccinated,
             y = reorder(State, PctFullyVaccinated),
                    tooltip = tooltip_text, data_id = State
                    )) +
  geom_col_interactive(color = "black", fill="#0072B2", size = 0.5) +
  theme_minimal() +
  theme(axis.text=element_text(size = 6),
        axis.title = element_blank()) +
  labs(title = "Percent Fully Vaccinated July 2021",
       subtitle = "Data from Our World in Data GitHub repo")

girafe(ggobj = latest_vax_graph, width_svg = 5, height_svg = 4)
```

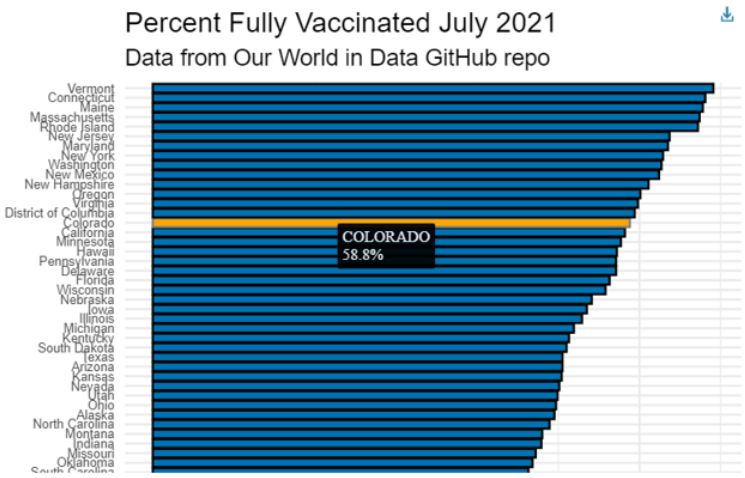


Figure 24: Bars

One thing that really makes `ggiraph` shine is how easy it is to link up multiple graphs. To demonstrate that, of course, we will need a second visualization to link to our bar chart.

11.6.5 Link a map and bar chart with ggiraph

We will create a USA map linked with the previous chart. Below is the code for the map.

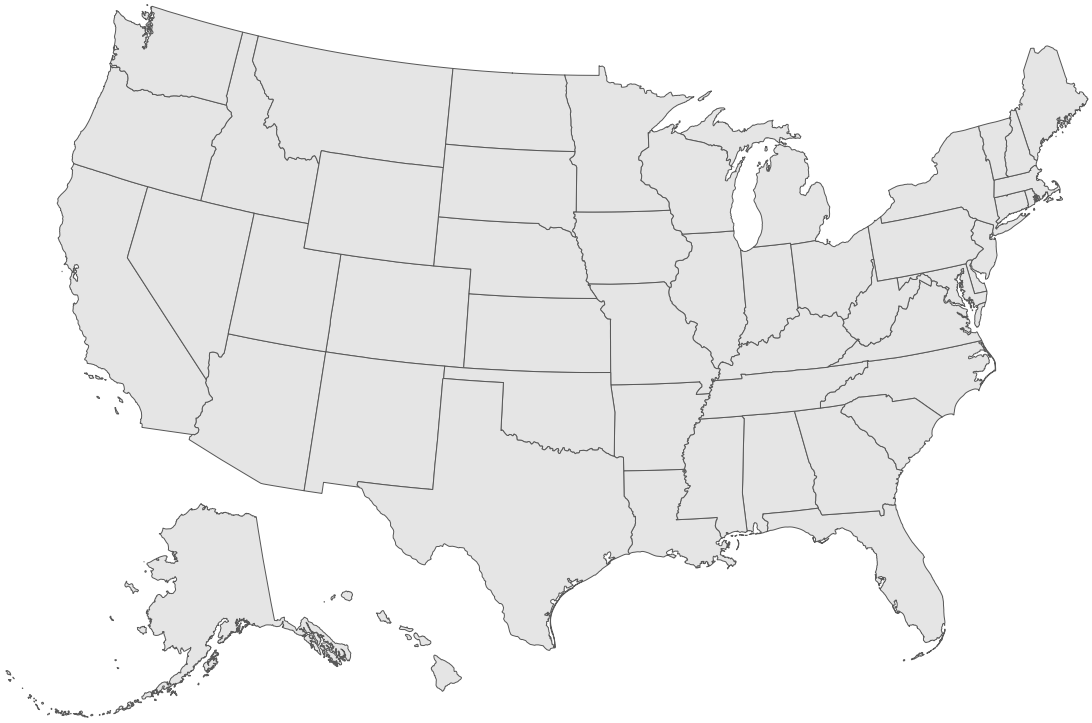
The us_sf is an R simple features geospatial object created with the `usa_sf()` function from the package `albersusa`.

The state_map creates a ggiraph map object from that us_sf object. The map code uses typical `ggplot()` syntax, but instead of `geom_sf()` it uses `geom_sf_interactive()`. There are also tooltip and data_id arguments in the `aes()` mapping. Finally, the code eliminates any background or axes with `theme_void()`.

```
us_sf <- usa_sf("lcc") %>%
  mutate(State = as.character(name))
```

```
## old-style crs object detected; please recreate object with a recent sf::st_crs()
```

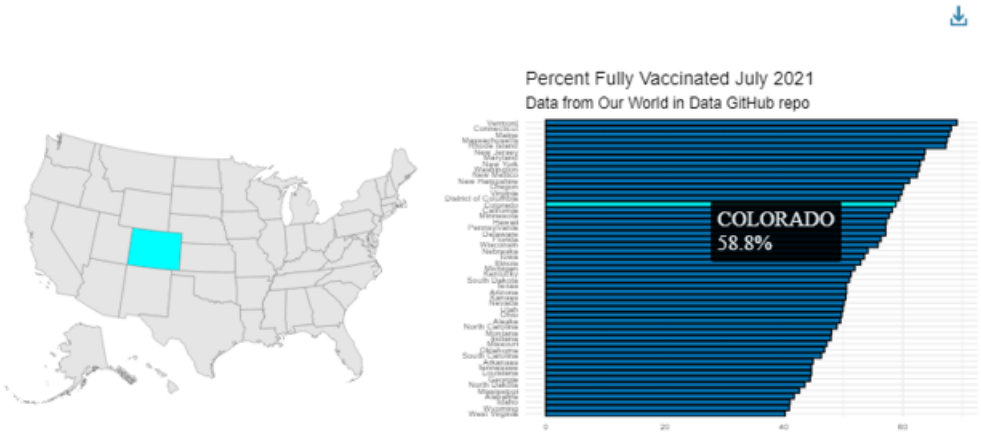
```
state_map <- ggplot() +
  geom_sf_interactive(data = us_sf, size = 0.125,
                      aes(data_id = State, tooltip = State)) +
  theme_void()

state_map
```



The next code block uses `girafe()` and its ggobj argument to display both the map and the vax graph, linked interactively.

```
linked_graph <- girafe(ggobj = state_map + latest_vax_graph,
                       width_svg = 10, height_svg = 5) %>%
  girafe_options(opts_hover(css = "fill:cyan;"))

linked_graph
```

It takes very little R code to make a static graphic interactive and to link two graphs together.



We can save the output from the `girafe()` function as an HTML widget and then save the widget to an HTML file using the htmlwidgets package. For example:

```
htmlwidgets::saveWidget(widget = linked_graph , here("figures", "linked_graph.html"))
```

12 Practical examples

12.1 Example 1: add statistics with `stat_summary()`

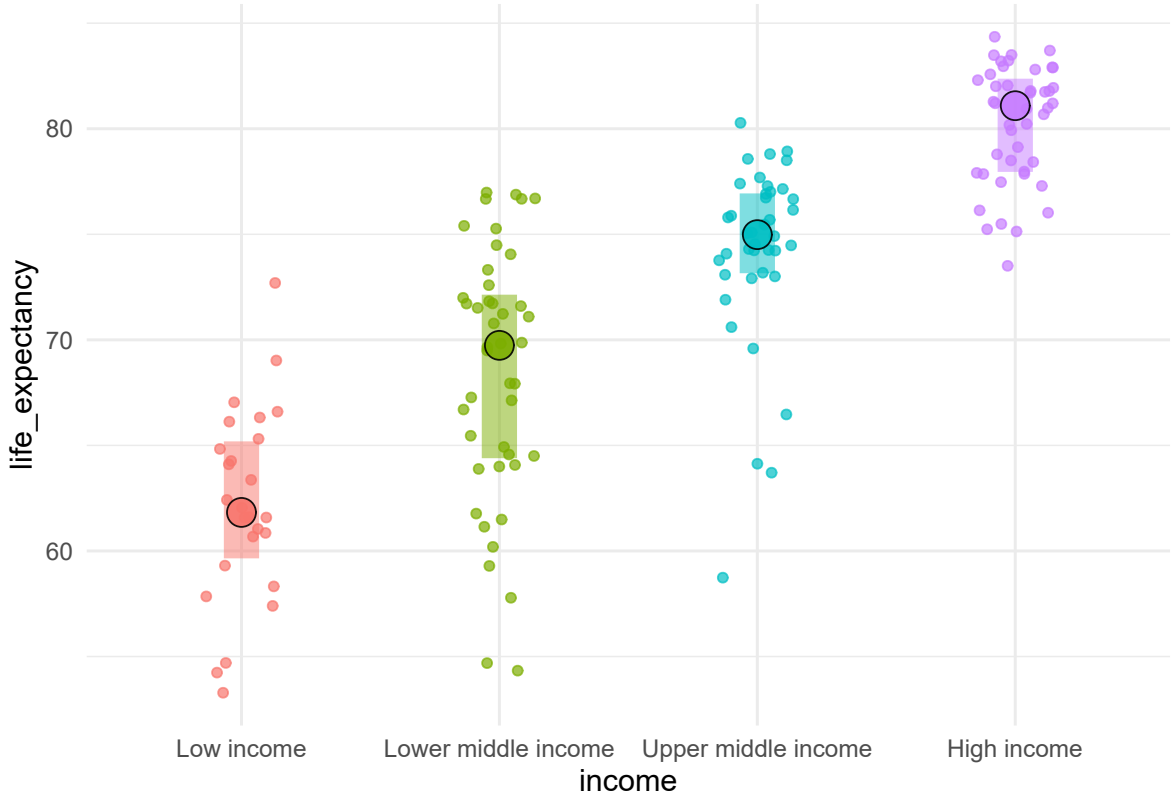
Let's go through on an example to understand how statistics can be overlayed in ggplot2. We will use the variables `income` and `life_expectancy` from `dat` and suppose we want to plot the median (and the interquartile range) of life expectancy for each income category.

First, we will create a function to calculate the interquartile range (IQR).

```
IQR <- function(x) {
  tibble(ymin = quantile(x, 0.25, na.rm = T), # 1st quartile
         ymax = quantile(x, 0.75, na.rm = T))  # 3rd quartile
}
```

Next, we use two `stat_summary()` layers to add our stats. In the first layer we add IQR and we can set the `fun.data` argument to the specific function defined above `IQR()`.

```
set.seed(236)
dat %>%
  mutate(income = factor(income, levels = c("Low income",
                                            "Lower middle income",
                                            "Upper middle income",
                                            "High income"))) %>%
  filter(!is.na(income)) %>%
  ggplot(aes(x = income, y = life_expectancy)) +
  geom_jitter(aes(colour = income),
              width = 0.15,
              size = 1.5,
              alpha = 0.7,
              show.legend = FALSE) +
  stat_summary(aes(color = income),
               geom = "linerange",
               fun.data = "IQR",
               alpha = 0.5,
               size = 6.5,
               show.legend = FALSE) +
  stat_summary(aes(fill = income),
               geom = "point",
               fun = "median",
               alpha = 0.9,
               size = 5,
               shape = 21,
               show.legend = FALSE) +
  theme_minimal(base_size = 12)
```

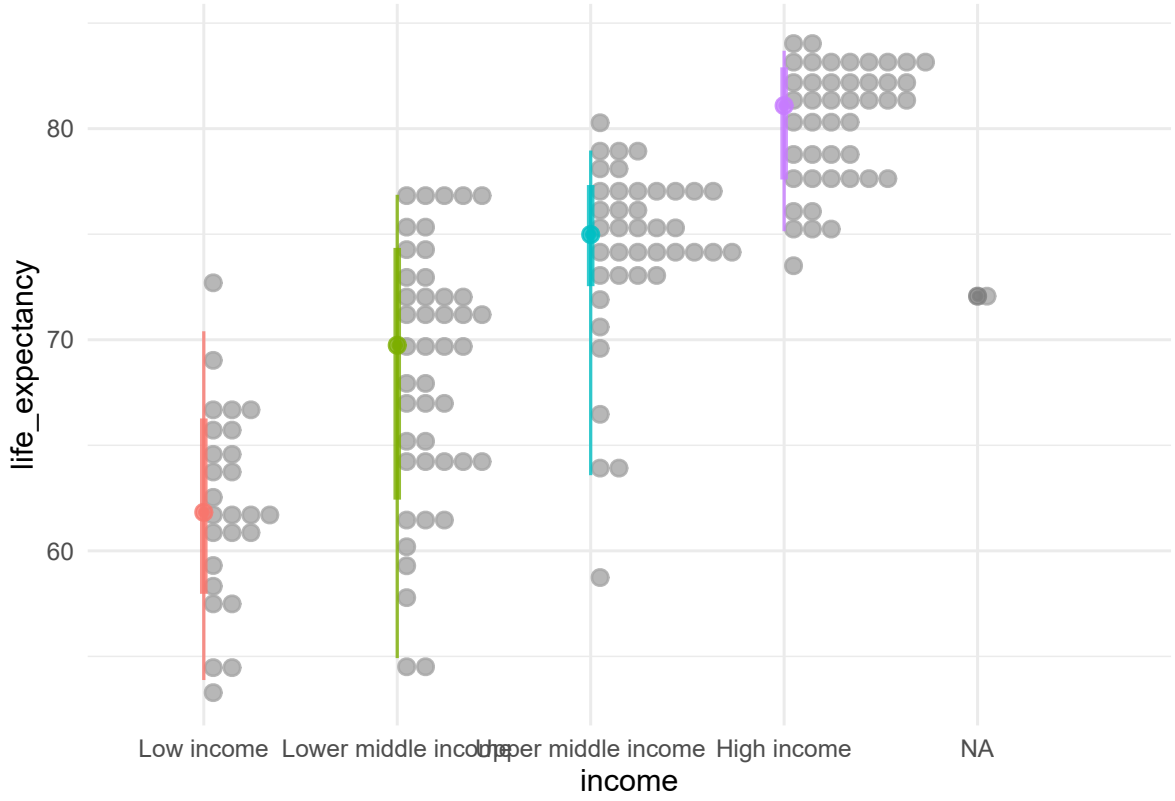


We can save our plot in a folder, for example, named `figures`:

```
ggsave(
  here::here("figures",
             paste0("figure_name", format(Sys.time(), "%Y%m%d_%H%M%S"),
                    ".tiff")),
       type = "cairo",
       width = 10, height = 8, dpi = 320,
       compression = "lzw")
```

The new package `ggdist` provides a flexible set of `ggplot2` geoms and stats designed especially for visualizing distributions and uncertainty. For example, eye plots combine densities (as violins) with intervals to give a more detailed picture of uncertainty than is available just by looking at intervals.

```
dat %>%
  mutate(income = factor(income, levels = c("Low income",
                                            "Lower middle income",
                                            "Upper middle income",
                                            "High income"))) %>%
  ggplot(aes(x = income, y = life_expectancy, color = income)) +
  stat_dotsinterval(alpha = 0.8, show.legend = F) +
  theme_minimal(base_size = 12)
```



12.2 Example 2: add smooth lines with `geom_smooth()`

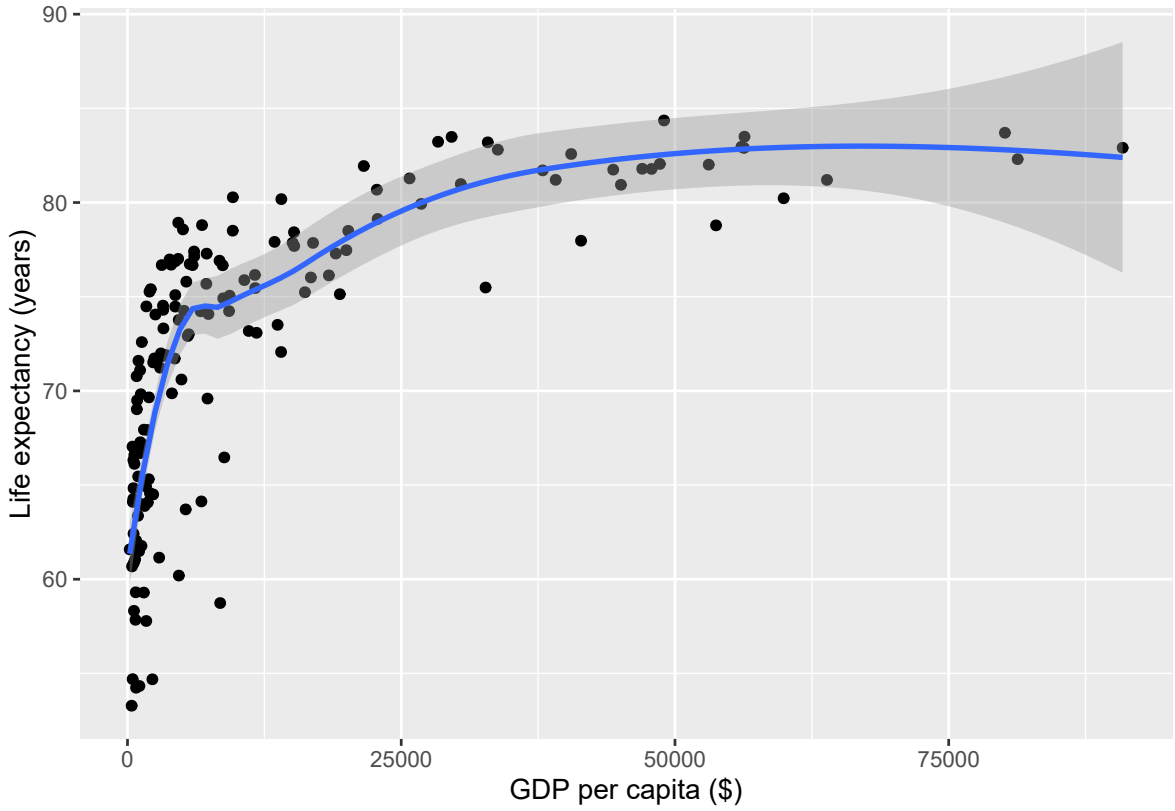
```
dat %>%
  ggplot(aes(x = gdp_capita, y = life_expectancy)) +
  geom_point() +
```

```
  geom_smooth() +
  labs(x = "GDP per capita ($)",
     y = "Life expectancy (years)")
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```

```
## Warning: Removed 2 rows containing non-finite
## values (stat_smooth).
```

```
## Warning: Removed 2 rows containing missing values
## (geom_point).
```

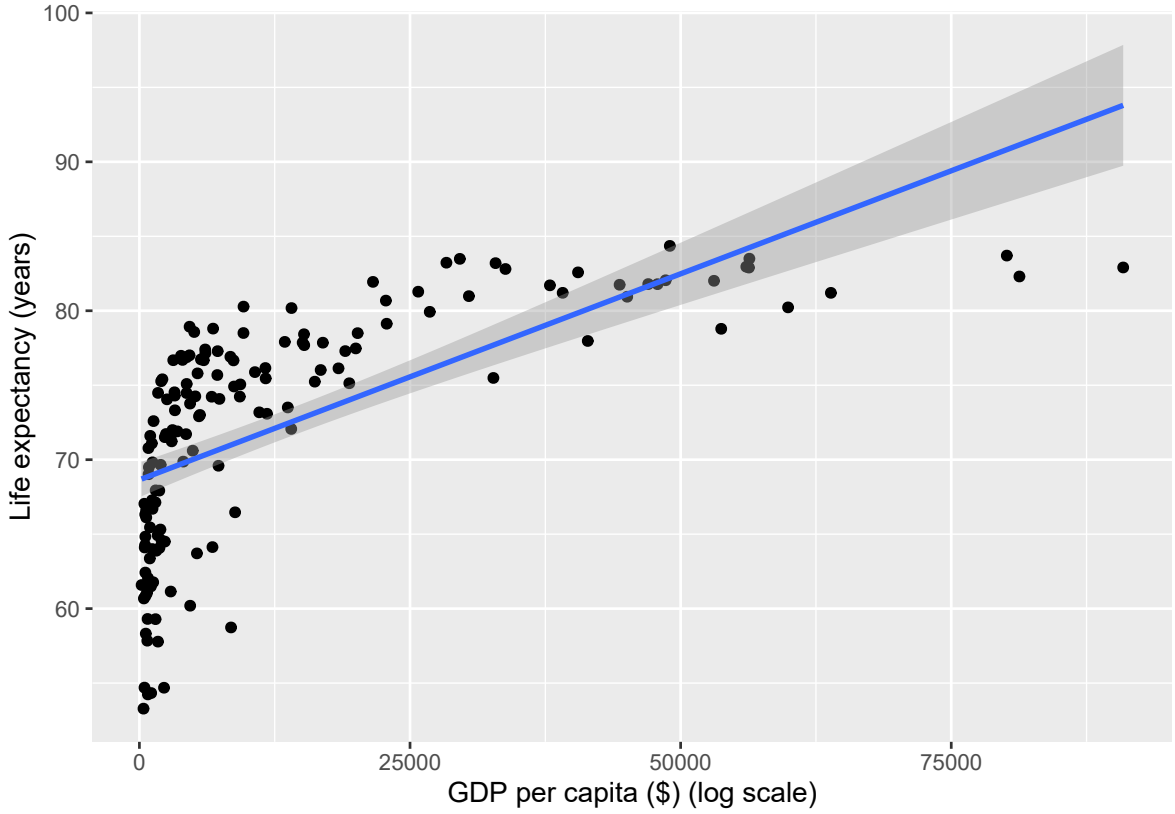


The console message R tells us the `geom_smooth()` function is using a method called `loess`. Loess smoothing is a process by which many statistical softwares do smoothing. In `ggplot2` this should be done when you have less than 1000 points, otherwise it can be time consuming.

This suggests that maybe there are other methods that `geom_smooth()` understands, and which we might tell it to use instead. Instructions are given to functions via their arguments, so we can try adding `method = "lm"` (for "linear model") as an argument to `geom_smooth()`:

```
dat %>%
  ggplot(aes(x = gdp_capita, y = life_expectancy)) +
  geom_point() +
  geom_smooth(method="lm", lwd = 1) +
  labs(x = "GDP per capita ($) (log scale)",
     y = "Life expectancy (years)")
```

```
## `geom_smooth()` using formula 'y ~ x'

## Warning: Removed 2 rows containing non-finite
## values (stat_smooth).

## Warning: Removed 2 rows containing missing values
## (geom_point).
```

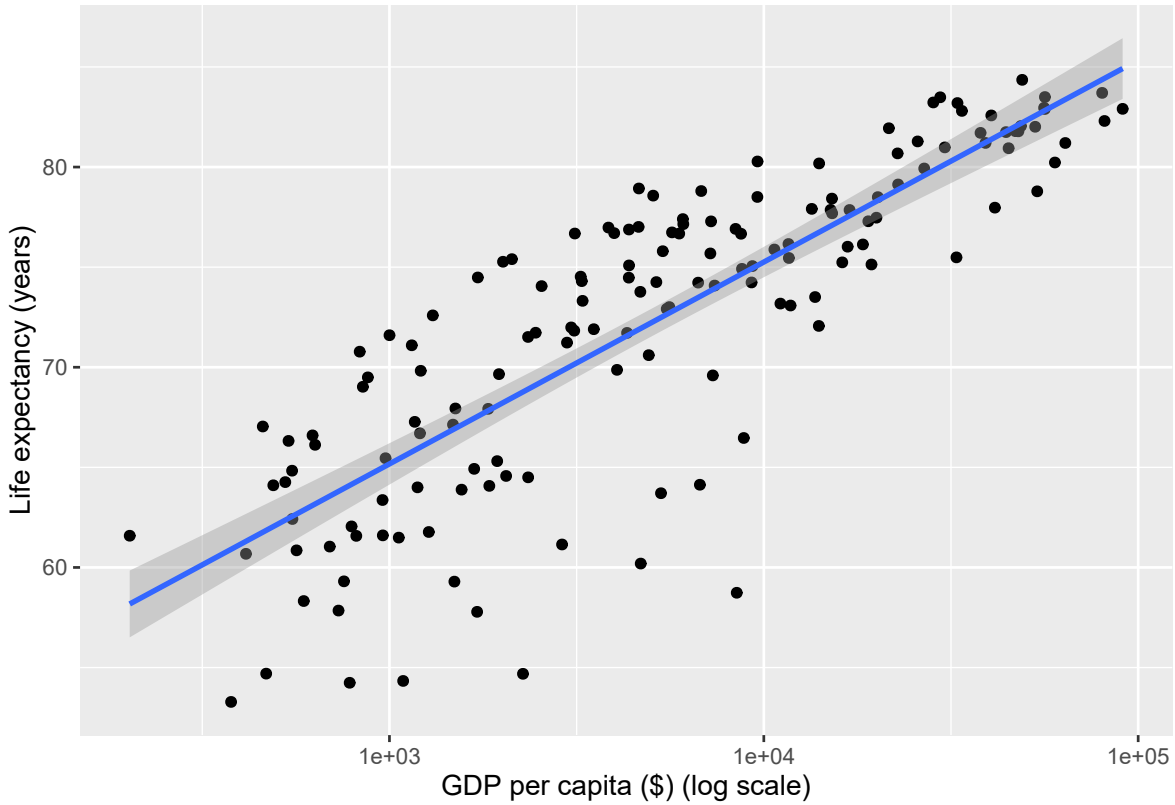


In our plot, the data is quite bunched up against the left side. Gross domestic product per capita is not normally distributed across our country years. The x-axis scale would probably look better if it were transformed from a linear scale to a log scale. For this we can use the function called `scale_x_log10()`.

```
dat %>%
  ggplot(aes(x = gdp_capita, y = life_expectancy)) +
  geom_point() +
  scale_x_log10() +
  geom_smooth(method="lm", lwd = 1) +
  labs(x = "GDP per capita ($) (log scale)",
     y = "Life expectancy (years)")
```

```
## `geom_smooth()` using formula 'y ~ x'

## Warning: Removed 2 rows containing non-finite
## values (stat_smooth).
```

```
## Warning: Removed 2 rows containing missing values
## (geom_point).
```

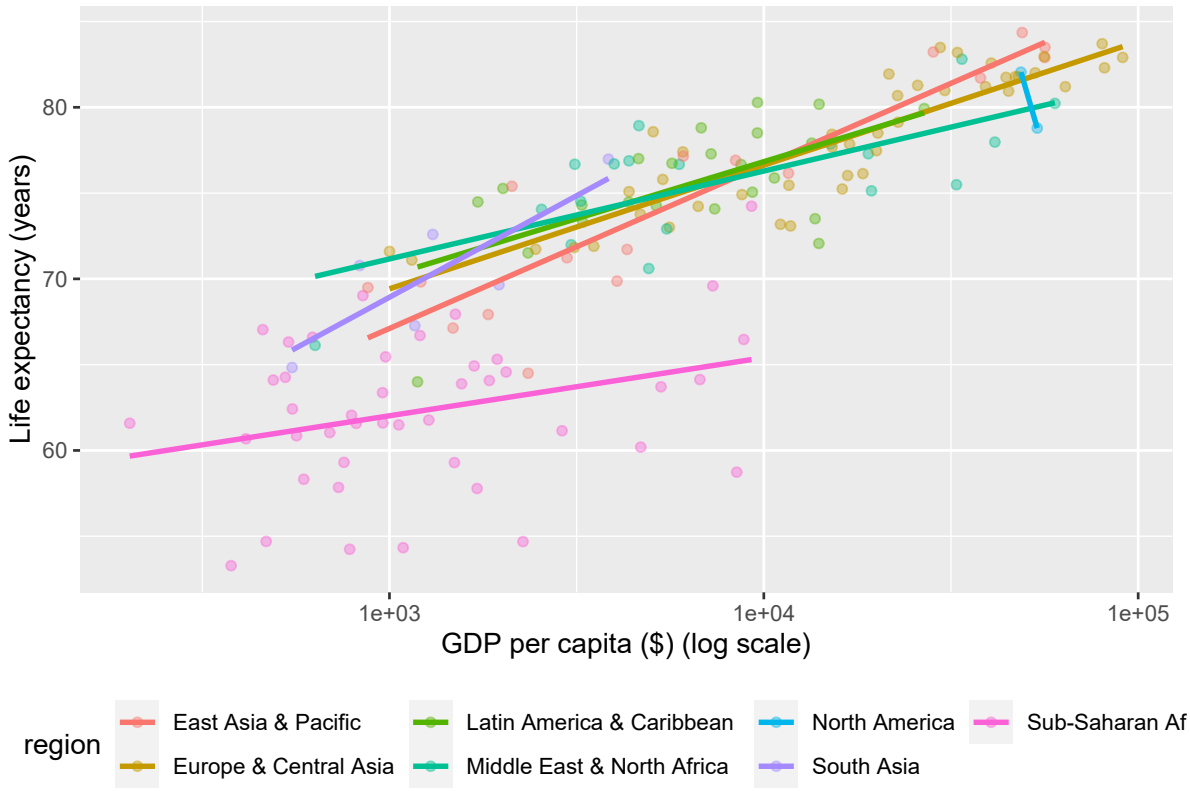



```
dat %>%
  ggplot(aes(x = gdp_capita, y = life_expectancy, color = region)) +
  geom_point(alpha = 0.4) +
  scale_x_log10() +
  geom_smooth(method="lm", lwd = 1, se=FALSE) +
  labs(x = "GDP per capita ($) (log scale)",
     y = "Life expectancy (years)") +
  theme(legend.position = "bottom")
```

```
## `geom_smooth()` using formula 'y ~ x'
```

```
## Warning: Removed 2 rows containing non-finite
## values (stat_smooth).
```

```
## Warning: Removed 2 rows containing missing values
## (geom_point).
```

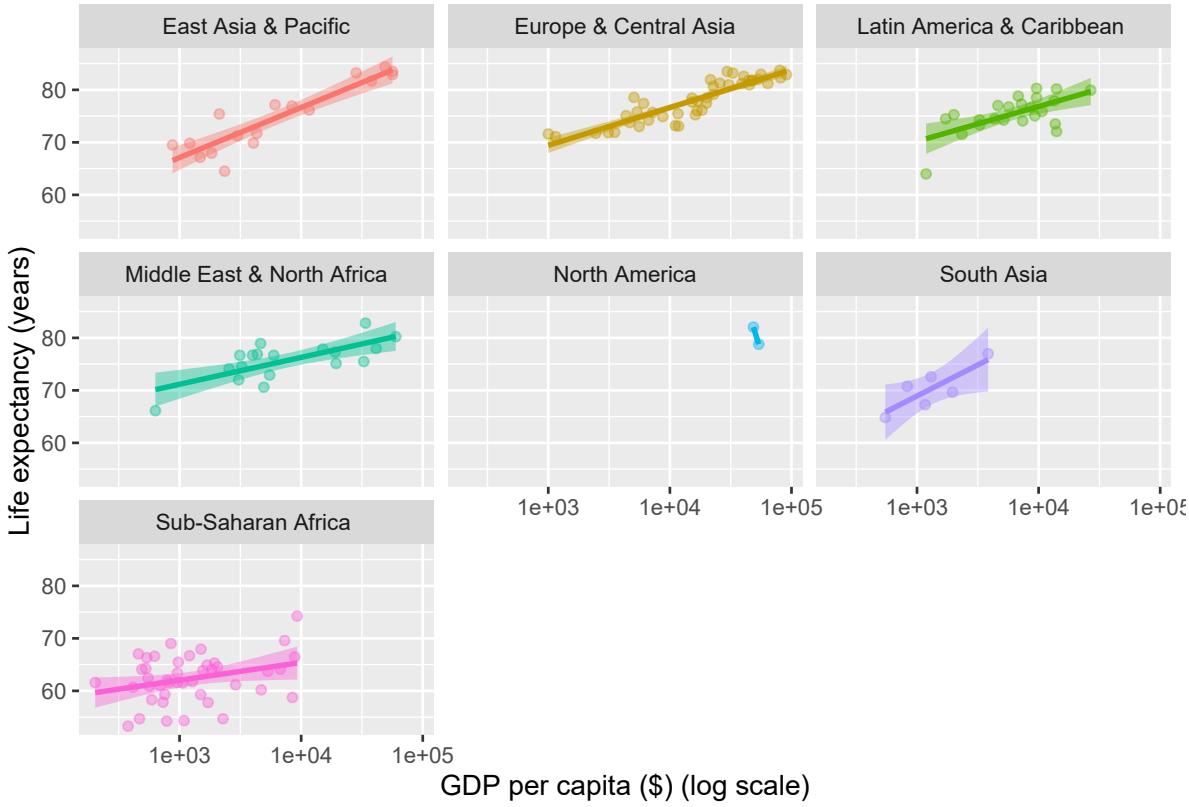


In above figure the individual data points have been colored by region, and a legend with a key to the colors has automatically been added to the plot. In addition, instead of one smoothing line we now have seven. There is one for each unique value of the region variable. This is a consequence of the way aesthetic mappings are inherited. Along with `x` and `y`, the color aesthetic mapping is set in the call to `ggplot()` that we used to create the object. Unless told otherwise, all geoms layered on top of the original plot object will inherit that object's mappings. In this case we get both our points and smoothers colored by region.

We might also consider facing according to region and shading the standard error ribbon of each line to match its dominant color. The color of the standard error ribbon is controlled by the fill aesthetic. Whereas the color aesthetic affects the appearance of lines and points, fill is for the filled areas of bars, polygons and, in this case, the interior of the smoother's standard error ribbon.

```r
dat %>%
  ggplot(aes(x = gdp_capita, y = life_expectancy, color = region)) +
  geom_point(alpha = 0.4) +
  scale_x_log10() +
  geom_smooth(aes(fill = region), method="lm", lwd = 1) +
  labs(x = "GDP per capita ($) (log scale)",
     y = "Life expectancy (years)") +
  facet_wrap(~region, ncol=3) +
  theme(legend.position = "none")
```

```
## `geom_smooth()` using formula 'y ~ x'

## Warning: Removed 2 rows containing non-finite
## values (stat_smooth).

## Warning in qt((1 - level)/2, df): NaNs
## produced

## Warning: Removed 2 rows containing missing values
## (geom_point).

## Warning in max(ids, na.rm = TRUE): no non-
## missing arguments to max; returning -Inf
```



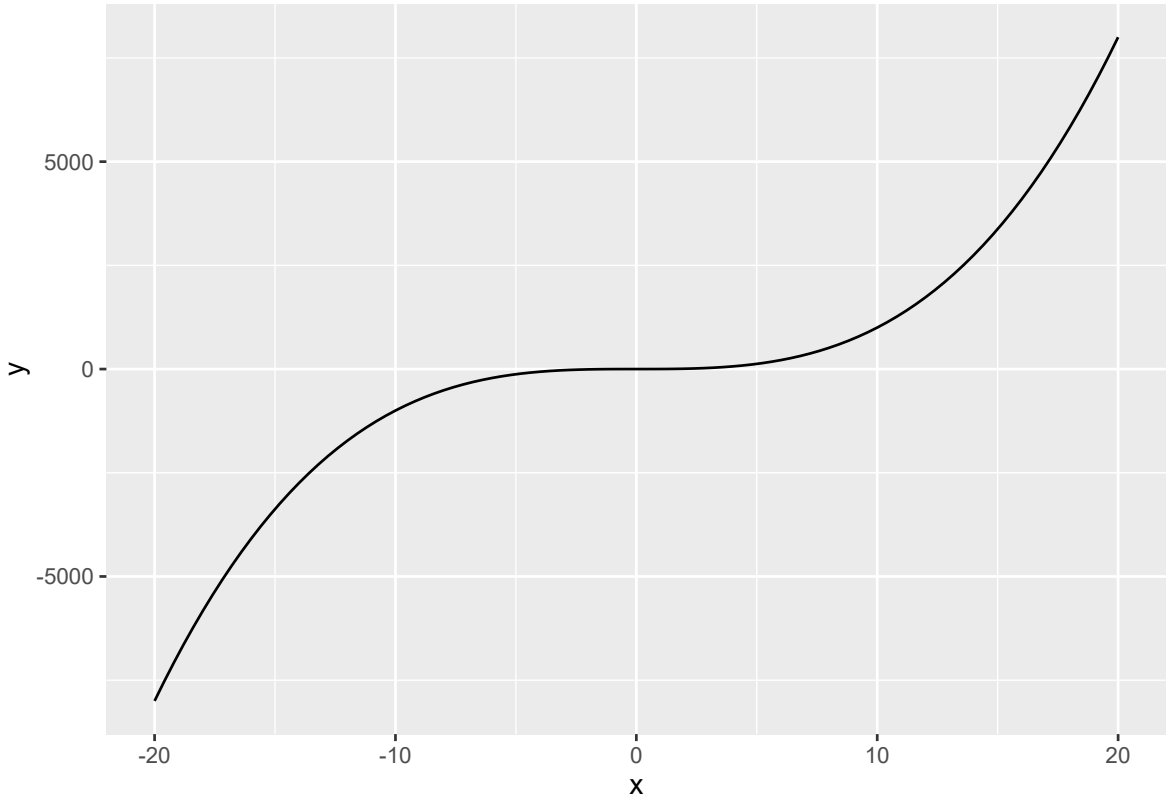
If it is what we want, then we might also consider to make it interactive with `ggplotly()`:

```
gg1 <- dat %>%
  ggplot(aes(x = gdp_capita, y = life_expectancy, color = region)) +
  geom_point(alpha = 0.4) +
  scale_x_log10() +
  geom_smooth(aes(fill = region), method="lm", lwd = 1) +
  labs(x = "GDP per capita ($) (log scale)",
     y = "Life expectancy (years)") +
  facet_wrap(~region, ncol=3) +
  theme(legend.position = "none")

ggplotly(gg1) %>%
  highlight("plotly_hover")
```

12.3 Example 3: visualize functions with `stat_function()`

A very neat way to visualize functions in `ggplot2` is to use the function `stat_function`. `stat_function1` allows us to visualize arbitrary functions. For example, we could draw a simple polynomial on an empty plot:

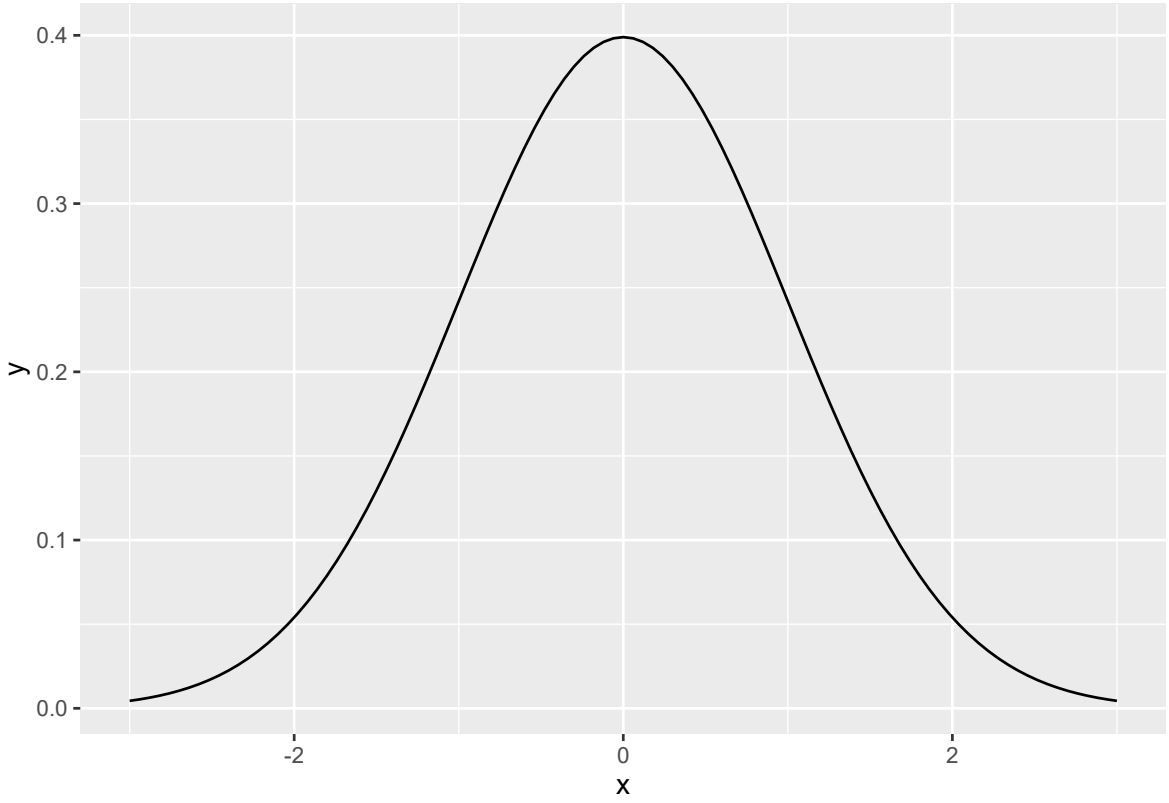
```
ggplot(data.frame(x = c(-20, 20)), aes(x)) +
  stat_function(fun = function(x) { x**3 },
                geom = "line")
```



As you can see, since we do not have any data, we created a dataframe that defines the limits of the visualization. The first argument of `stat_function()` is `fun`, which obviously stands for function.

We can can make up our own functions from scratch (as in the example above) or use predefined functions like `dnorm()` to visualize the normal distribution:

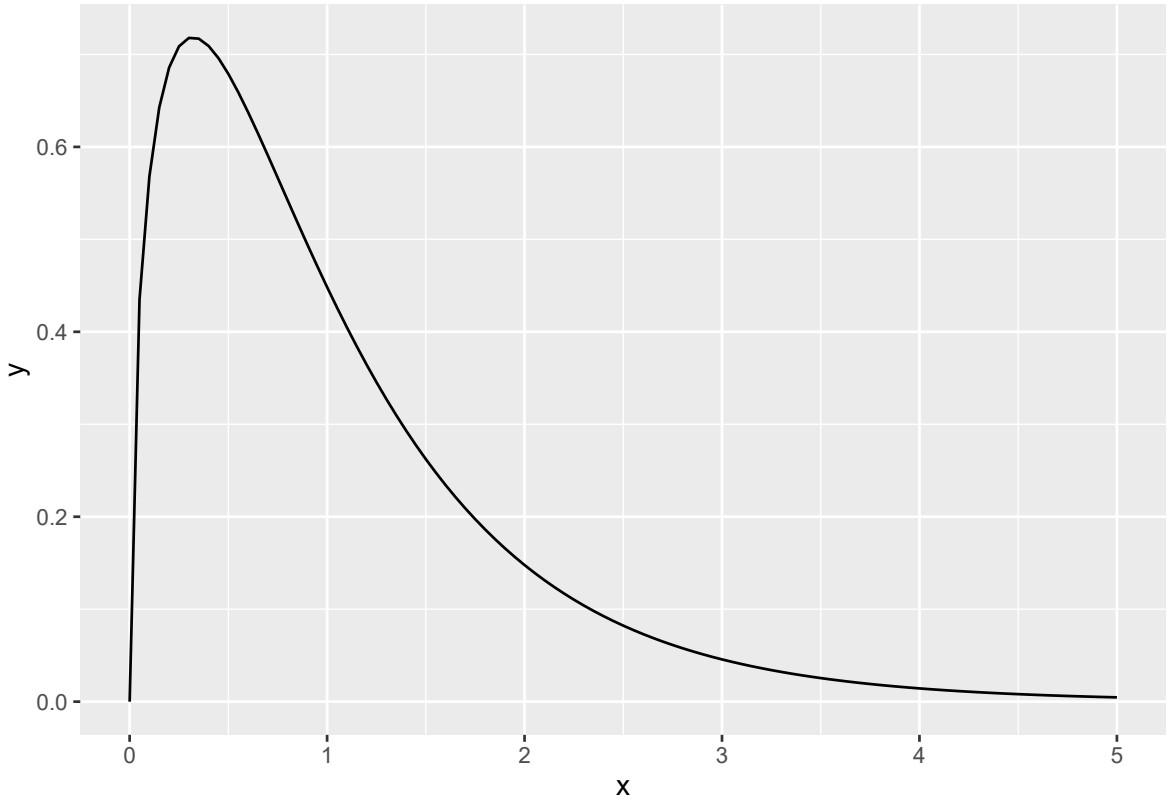
```
ggplot(data.frame(x = c(-3, 3)), aes(x)) +
  stat_function(fun = dnorm,
                geom = "line")
```



We can also create a particular F distribution with `stat_function`:

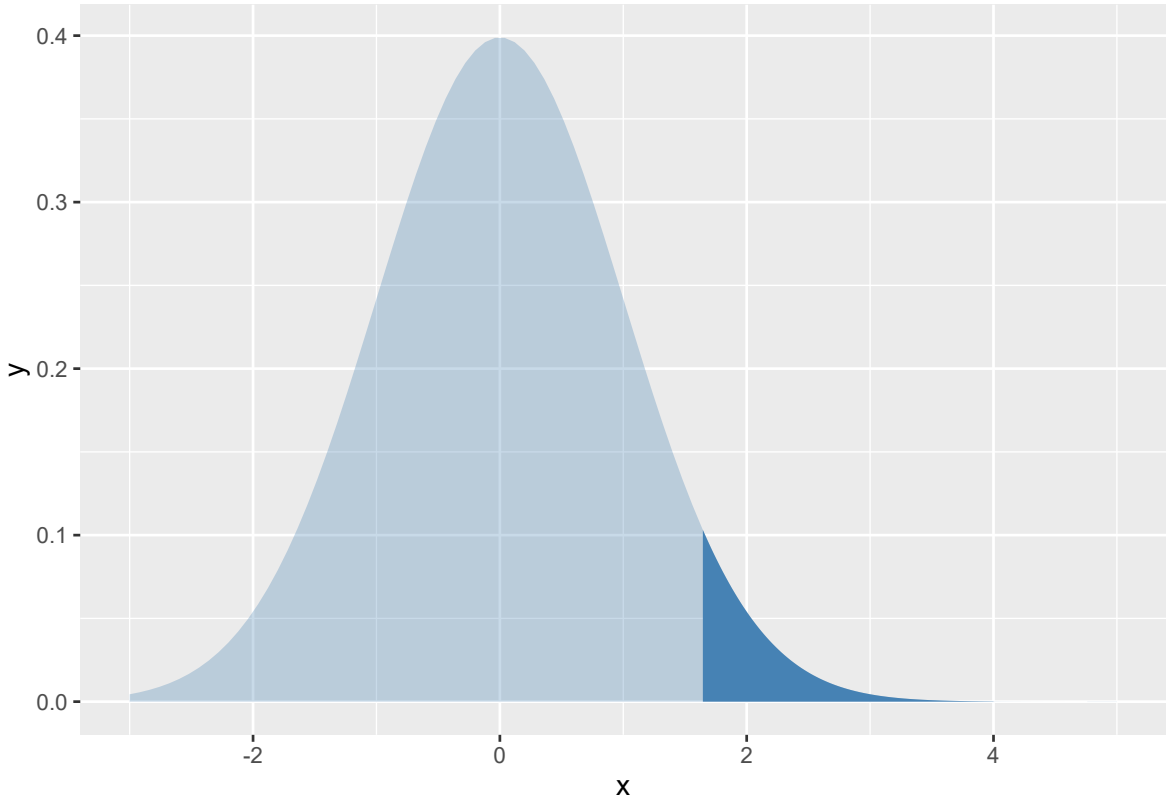
```
ggplot(data.frame(x = c(0, 5)), aes(x)) +
  stat_function(fun = df,
                geom = "line",
                fill = "steelblue",
                args = list(
                  df1 = 3,
                  df2 = 47
                ))
```

```
## Warning: Ignoring unknown parameters: fill
```



Next, we add areas under the curves using the geom `area`. Let's see an example for the normal distribution with the critical area (α = 0.05):

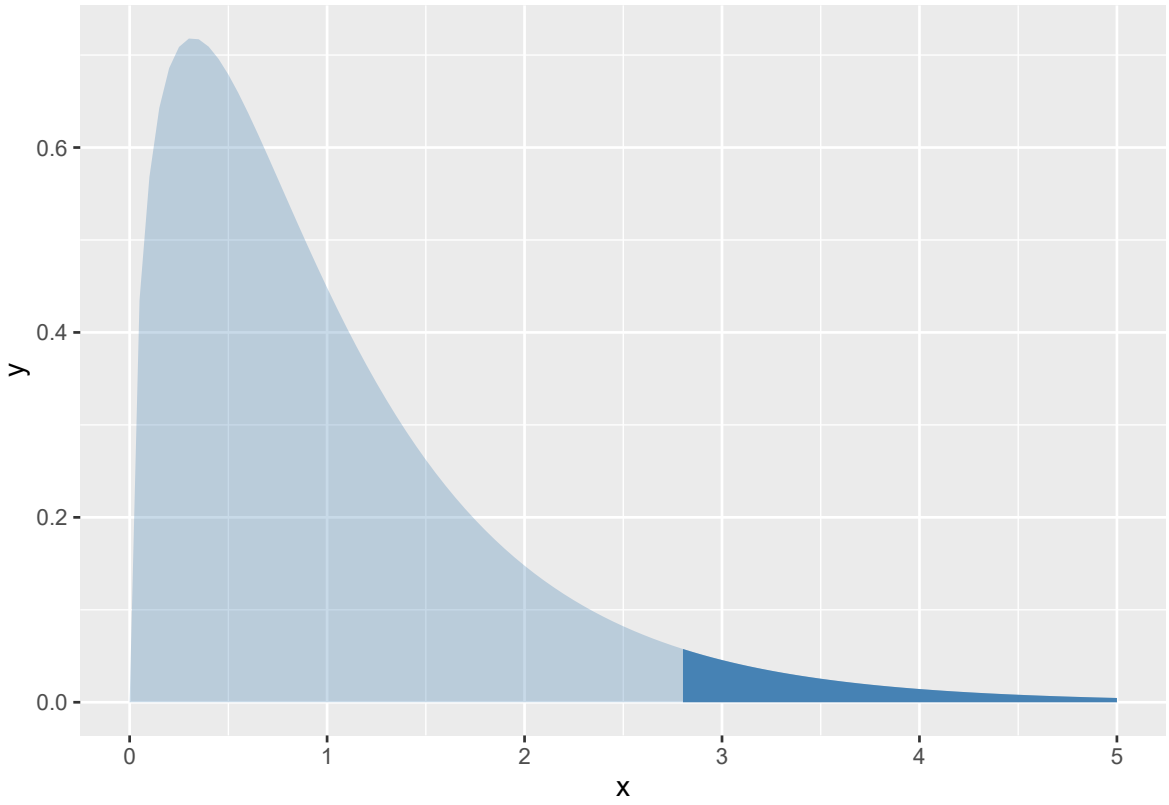
```
ggplot(data.frame(x = c(-3, 5)), aes(x)) +
  stat_function(
    fun = dnorm,
    geom = "area",
    fill = "steelblue",
    alpha = 0.3
  ) +
  stat_function(
    fun = dnorm,
    geom = "area",
    fill = "steelblue",
    xlim = c(qnorm(0.95), 4)
  )
```



Similarly, the area plot for the `F-distribution`:

```
ggplot(data.frame(x = c(0, 5)), aes(x)) +
  stat_function(fun = df,
                geom = "area",
                fill = "steelblue",
                alpha = 0.3,
                args = list(
                  df1 = 3,
                  df2 = 47)) +
  stat_function(fun = df,
                geom = "area",
                fill = "steelblue",
                xlim = c(qf(0.95, 3, 47), 5),
                args = list(
                  df1 = 3,
```

```
                    df2 = 47))
```



12.4 Example 4: dose-response curves `ggprism`

In pharmacology experiments, a drug's effect on a receptor is typically investigated by constructing a dose-response curve. A dose-response curve describes the relationship between increasing the dose (or concentration) of the drug and the change in response that results from this increase in concentration.

A typical dose-response curve will span a large concentration range. For this reason, the dose in these curves is usually represented using a semi-logarithmic plot. On a semi-logarithmic plot, the amount of drug is plotted (on the `x` axis) as the log of drug concentration and response is plotted (on the `y` axis) using a linear scale.

We will create a graph that shows the 'response' of e.g., a cell surface receptor when we add increasing concentrations of an agonist molecule in the presence or not of a

competitive inhibitor. The underlying data set comes from one of the xy dose response tutorials included with Prism.

```
df <- read_csv(here("data", "dose_response.csv"))
```

```
## Rows: 56 Columns: 5
```

```
## -- Column specification --------------------
## Delimiter: ","
## chr (1): treatment
## dbl (4): agonist, log.agonist, rep, response
```

```
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

First, we need to define a formula for `geom_smooth()` to use. Here, we will define a four-parameter log-logistic model manually. See this notebook for more info on dose response curves in R.

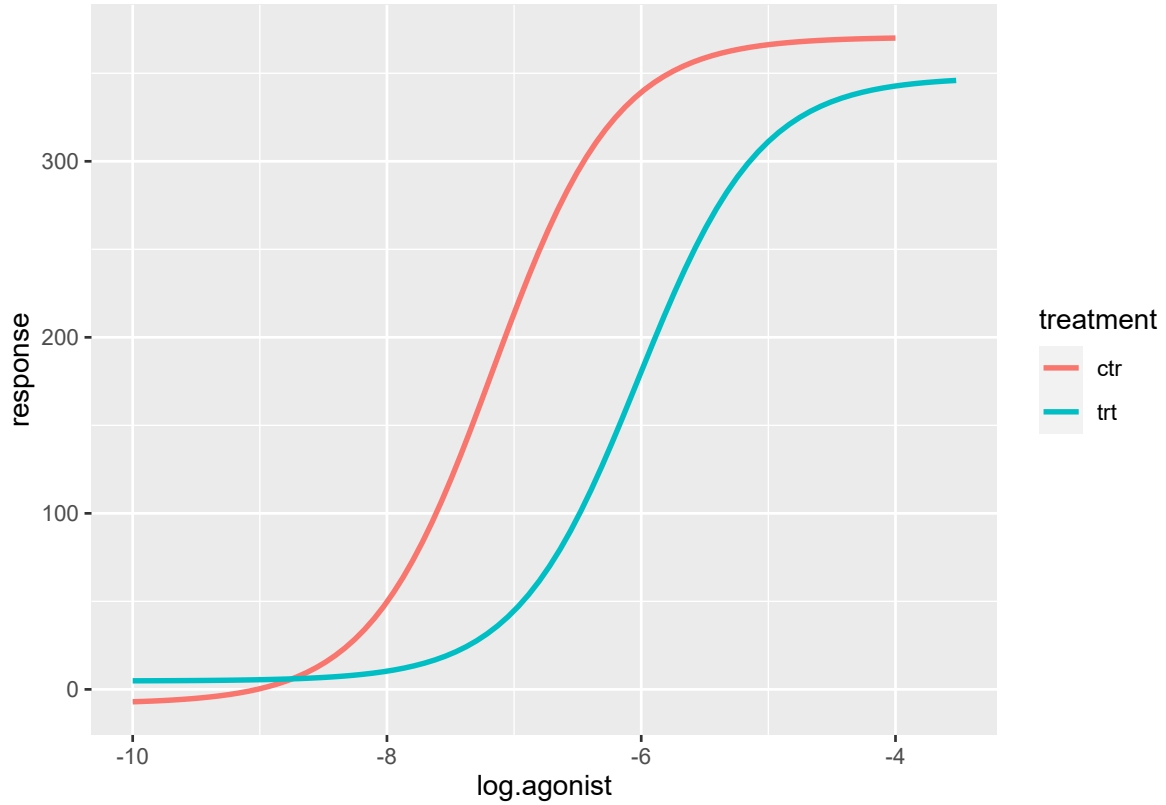
```
# define model
dose_resp <- y ~ min + ((max - min) / (1 + exp(hill_coefficient * (ec50 - x))))
```

where ec50 is half maximal effective concentration; the Hill coefficient affects the steepness of the slope.

Now we begin constructing the plot, layer-by-layer. The order of each step below is important and if you mix them around the final plot will not appear as it should.

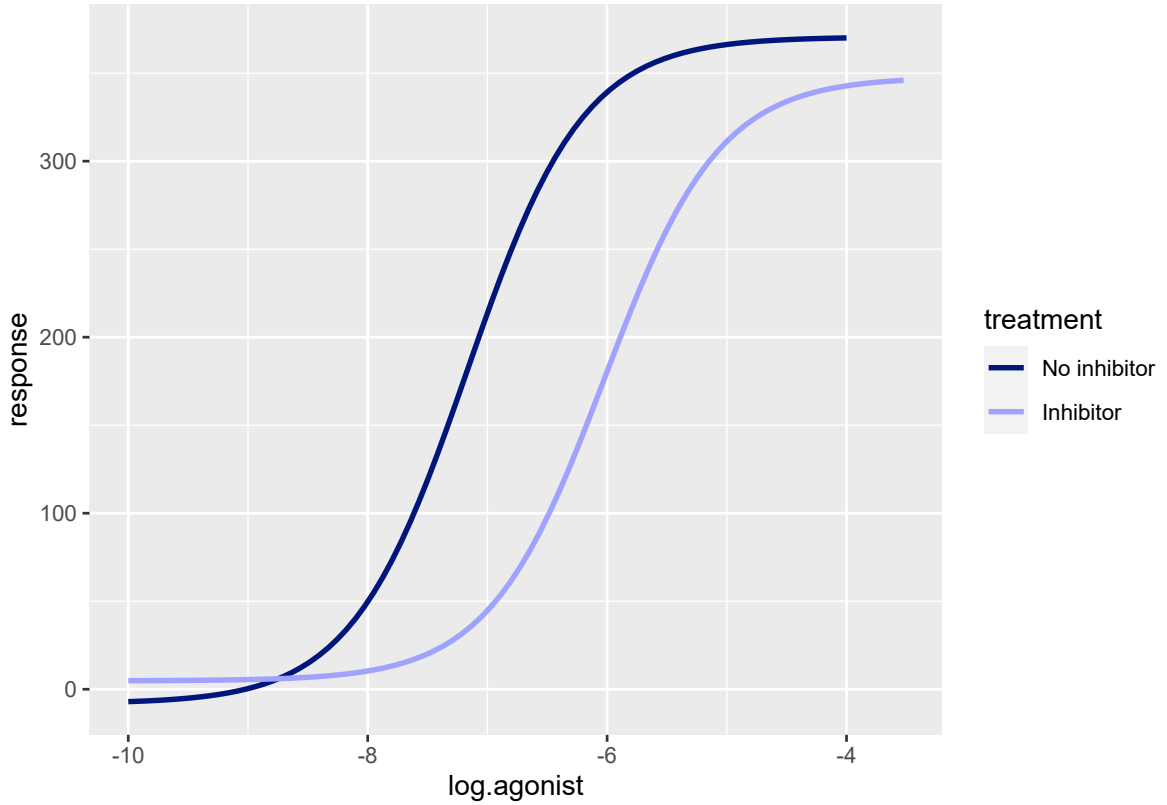
```
# plot the log10(agonist concentration) vs the response
graph1 <- ggplot(df, aes(x = log.agonist, y = response)) +
  geom_smooth(aes(color = treatment), method = "nls",
              formula = dose_resp, se = FALSE,
              method.args = list(start = list(min = 1.67, max = 397,
```

```
                                        ec50 = -7,
                                        hill_coefficient = 1))
)

graph1
```



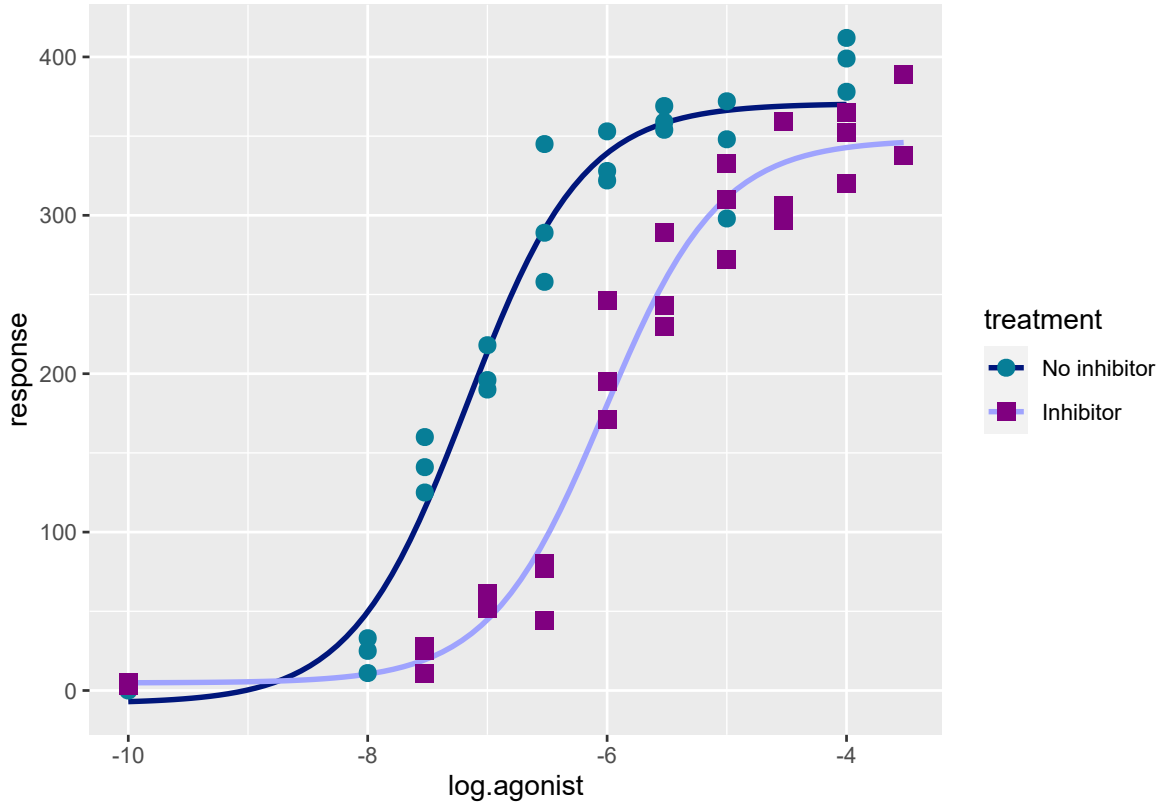
We can adjust color manually and we can give the two treatment groups more informative names in the legend.

```
graph2 <- graph1 +
  scale_color_manual(labels = c("No inhibitor", "Inhibitor"),
                     values = c("#00167B", "#9FA3FE"))
graph2
```



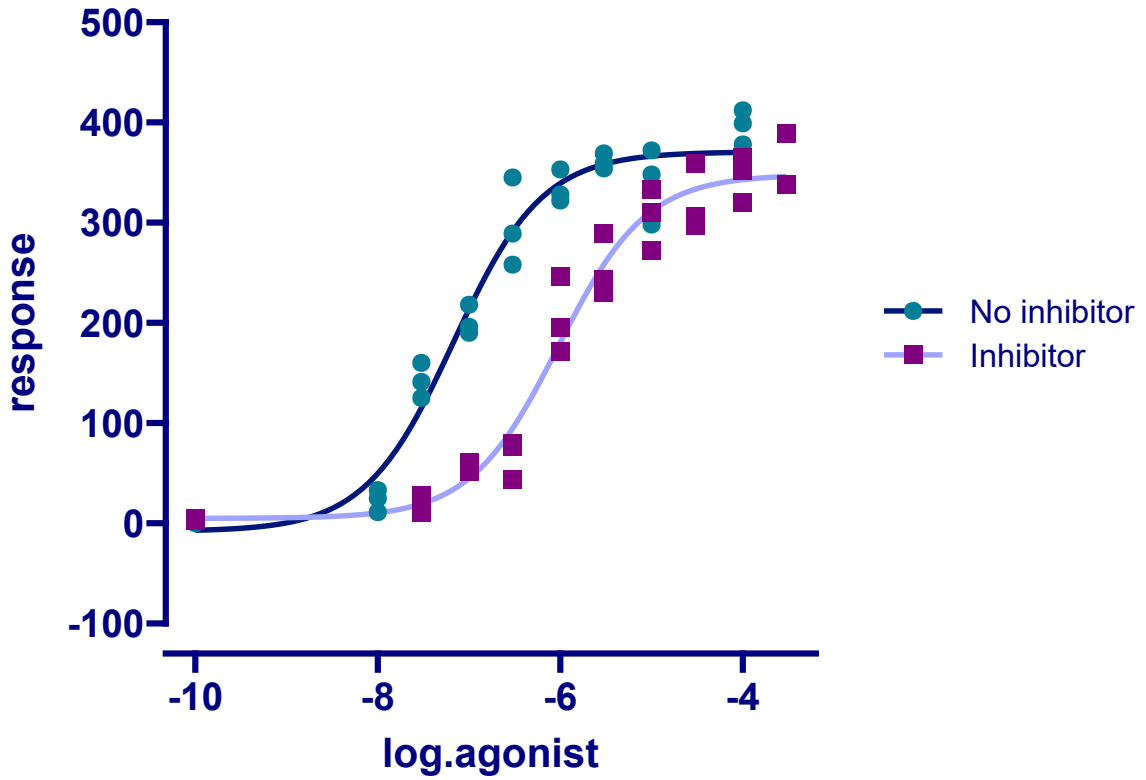
Now we can plot the observed data using `geom_point`. We'll change the point shape and color depending on the treatment type. To use a separate color scale for the points versus the curves we can use the `new_scale_color()` function from package `ggnewscale`. `new_scale_color()` must be before `geom_point()` otherwise the new color scales will not work.

```
graph3 <- graph2 +
  new_scale_color() +
  geom_point(aes(color = treatment, shape = treatment), size = 3) +
  scale_color_prism(palette = "winter_bright",
                    labels = c("No inhibitor", "Inhibitor")) +
  scale_shape_prism(labels = c("No inhibitor", "Inhibitor"))

graph3
```



We'll apply `theme_prism` and we'll change the `y` axis limits and appearance. The `guide = prism_offset()` in `y` axis guide will shorten the axis line to the outer-most tick marks.

```
graph4 <- graph3 +
  theme_prism(palette = "winter_bright", base_size = 16) +
  scale_y_continuous(limits = c(-100, 500),
                     breaks = seq(-100, 500, 100),
                     guide = "prism_offset")
graph4
```



Then we'll change the `x` axis limits and appearance. This step is somewhat complicated by 2 things:

- We are working with log10 transformed data. First we define the major ticks with the breaks and limits arguments. Then we use the `prism_offset_minor()` axis guide to add minor tick marks. Finally, we use the `minor_breaks` argument and give it a vector that defines the `x` axis position of each individual minor tick mark. This can be done with `%o%` to quickly generate a multiplication table and `as.numeric()` to flatten the table to a vector.

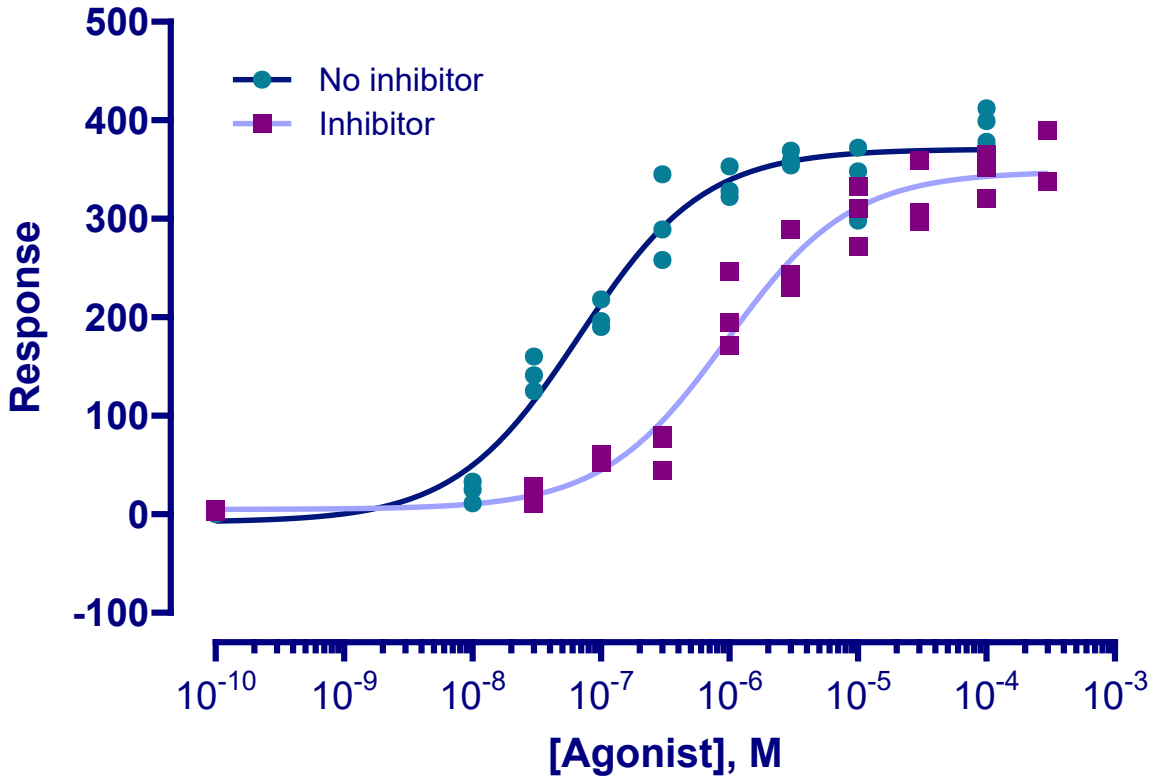
```
lab_reformat <- unique(as.numeric(1:9 %o% 10 ^ (-10:-4)))
lab_reformat
```

```
##  [1] 1e-10 2e-10 3e-10 4e-10 5e-10 6e-10
##  [7] 7e-10 8e-10 9e-10 1e-09 2e-09 3e-09
## [13] 4e-09 5e-09 6e-09 7e-09 8e-09 9e-09
```

```
## [19] 1e-08 2e-08 3e-08 4e-08 5e-08 6e-08
## [25] 7e-08 8e-08 9e-08 1e-07 2e-07 3e-07
## [31] 4e-07 5e-07 6e-07 7e-07 8e-07 9e-07
## [37] 1e-06 2e-06 3e-06 4e-06 5e-06 6e-06
## [43] 7e-06 8e-06 9e-06 1e-05 2e-05 3e-05
## [49] 4e-05 5e-05 6e-05 7e-05 8e-05 9e-05
## [55] 1e-04 2e-04 3e-04 4e-04 5e-04 6e-04
## [61] 7e-04 8e-04 9e-04
```

- We want the axis text labels to be in the format 10^x but at the moment they are in the format x. To do this we feed the label argument with `math_format(10^.x)` which defines a math expression that, for example, takes the number -7 and convert it into the expression 10^-7, .

```
graph5 <- graph4 +
  scale_x_continuous(limits = c(-10, -3), breaks = -10:-3,
                     guide = "prism_offset_minor",
                     minor_breaks = log10(lab_reformat),
                     labels = math_format(10^.x)
  ) +
  theme(legend.title = element_blank(),
        legend.position = c(0.05, 0.95),
        legend.justification = c(0.05, 0.95)) +
  labs(x = "[Agonist], M", y = "Response")

graph5
```



13 Activities

13.1 Activity 1

Create an appropriate bar plot visualization for life expectancy of the countries France, Germany and United Kingdom based on the data of `dat`. What is the problem with this graphical representation? What approach can be used to address the issue?

13.2 Activity 2

Using the `dat` visualize how the life expectancy in `Europe & Central Asia` varies across the countries by plotting a square for each country. Additionally, map the data of variable `pop_density` to a color aesthetics.

13.3 Activity 3

Using the `dat` visualize the countries with the higher proportion of vaccination/population (top 20 countries). Use the package `ggflags` to add the flag for each country.