Data visualisation - part 1: visualisation with ggplot2

Hannah Meyer

01/08/2020

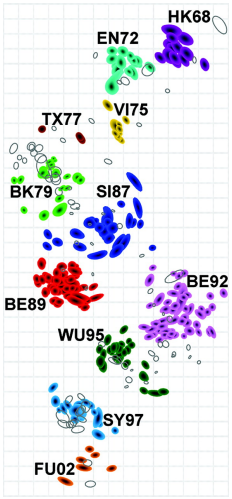
Contents

1 Background: Antigenic cartography

Antigenic Cartography is the process of creating maps that reflect the antigenic properties of a pathogen. In this course, we will use antigenic cartography data used to map the antigenic evolution of influenza viruses.

To create antigenic maps, measurements of laboratory binding assays (e.g. ELISA or hemagglutination inhibition assays) are converted into a coordinate space in which the distance between points reflects their similarity in the original assay. For instance, to create an antigenic map of influenza viruses, antigens and sera are titrated in a hemagglutination inhibition (HI) assay. The HI assay tests the ability of influenza viruses to agglutinate red blood cells and the ability of animal antisera raised against the same or related strains to block this agglutination. Sera with similar abilites to block agglutination will be in close proximity in the antigenic map and vice versa.

Smith *et al* (2004) mapped the antigenic evolution of human influenza A (H3N2) viruses, which have been a major cause of influenza epidemics since the 1968 Hong Kong influenza pandemic. We will use their data of 323 HI measurements and antigenic cooridinates derived from viruses in world-wide circulation from 1968 to 2003, which created the following map:



At the end of this course, you will be able to recreate this map, plot the distribution of measurements counts over time and visualise the location of virus processing on a word map.

2 Setting up

2.1 Starting a Rproject for your analysis

First, we will set up a project, where you will keep all files associated with that project - inlcuding your analysis reports, input data, results and figures.

- In R studio click *File > New Project*.
- Choose *Existing directory*, then click on *Browse* to find the folder that you created with the course material
- Click *Create Project*

In your files plane in R studio you can now see all the files you downloaded for this workshop. We will be able to easily read them into R and save results there.

There are other ways of specifying the folders from which you read and where you write results and save plots to, but I would highly recommend this strategy. It keeps everything neat and you have all the important parts of your analysis in one place.

R studio can be customised and I leave this to everyone to figure out what works best. However, I would recommend changing one default setting, which will ensure that when you start working in your project, you start of with a clean slate and none of previously computed data sticks around and confuses your analysis.

For this setting go to *RStudio > Preferences* (on Mac) and ** (windows/unix). In *General* untick *restore .RData into workspace at startup.*

We are all set now to go ahead with your first analyis and data visualisation in R!

2.2 Setting up an analysis report

The document you are looking at right now is a R Notebook. R Notebooks allow us to interleaf text describing our analysis with the R code that actually contains the analyses commands.

The text follows some simple markdown rules (for instance bold header sections etc, which we will not go into detail here). Important for us at this stage is that whenever we want to include analysis code into the document, we have to create an R code *chunk*. To include a new *chunk* click the *Insert* buttom at the top of your editor window and select *R*.

All code *chunks* have some default settings, concering their layout, execution etc, which can be heavily customised. For our beginners tutorial, we do not have to worry about all of these. I mention this here, as the following and first chunk of our document contains some basic options that I want to have applied to all chunks in the rest of the document. Specifically, it tells R studio that when I prepare this document for sharing with you as a pdf, that I want both the actual code and the results displayed in the document. It also specifies the width and alignment of the figures in the final document.

```
knitr::opts_chunk$set(echo = TRUE,
                      comment = "#>",
                      collapse = TRUE,
                      fig.width = 6, fig.asp = 0.618, fig.align = "center",
                      out.width = "70%")
```

I then follow with a chunk that loads all libraries required for my analysis. The following chunk loads the libraries that you installed as a preparation for the course into your R workspace:

```
library("tidyverse")
library("cowplot")
library("sf")
library("rnaturalearth")
library("RColorBrewer")
```

Note: Libraries only have to be installed once, however, they will have to be loaded into the R workspace whenever we open a new R session. Think of the libraries you install as tools that you buy: you buy a hammer the first time you realise you want to hang a picture and need a hook in the wall. Once you bought it, you need to actually bring it to the room where you want to hang the picture. After that, you have the hammer and can use it whenever you like. The hammer is the library that you install (buy) once and then load into your R workspace (use) whenever you have a task that can be accomplished with that hammer.

3 Data input

The first step in data analysis with R is to read the data into the R workspace. We can do this with the `read_csv` function.

3.1 A primer to R functions

Functions automate common tasks such as reading files into R, creating a histogram of your data and saving that histogram in a pdf document.

Functions take a set of arguments, evaluate them and return the result. There are two possible outcomes that we might want to see when we use a function:

If we simply want to see the result of the function displayed after execution, we just type the command and execute it. If we want to store the result in a variable for future use, we have to assign the outcome of a function into an object by using the assign operator `<-`.

To see a description of the function, its arguments and results, use the help function `?` by typing `?function_name` in the R console. This help function proves really useful whenever you want to use a new function or you want to look up some examples of how to use the function.

4 Reading your data

In the following chunk, we pass the filename of the data that we want to load to `read_csv` function and assign the ouput to a new object called `coord`.

```
coord <- read_csv("2004_Science_Smith_data.csv")
#> Parsed with column specification:
#> cols(
#>   name = col_character(),
#>   location = col_character(),
#>   year = col_double(),
#>   cluster = col_character(),
#>   type = col_character(),
#>   x.coordinate = col_double(),
#>   y.coordinate = col_double()
#> )
```

`read_csv()` prints out a column specification that gives the name and type of each column.

4.1 Exercises

1. Use the help function`?` to have a look at the documentation of `read_csv`.
2. Have a look at the `coord` object by creating a new chunk, typing `coord` and executing the code chunk.
3. Do you find the message printed by `read_csv` represented in `coord`?

5 Data and object types

The objective of this course is data visualisation and we will not spend a huge amount of time on learning R's data structures. In the following, we will just look at a few basic data types and objects that we will encounter in this course:

5.1 Data types

The most common data types in R are:

- `int`, which stands for integers i.e 1, 2, 3;
- `dbl`, which stands for doubles, or real numbers i.e. 1.2, 1.7, 9.0;
- `chr`, which stands for character vectors, or strings i.e. "a", "b", "word";
- `lgl`, which stands for logical, vectors that contain only TRUE or FALSE;
- `fctr`, which stands for factors, which R uses to represent categorical variables with fixed possible values.

5.2 Object types

There are many different data types in R. For the purpose of data visualisation and this workshop, we will only work with two object types:

- `data.frame`: a list of variables with the same number of observations. Variables are in columns, observations in rows. Rows have unique rownames.
- `tibble`: tibbles are 'opinionated data frames' with an improved printing display, stricter rules for re-formating that aid in avoiding bugs and no rownames.

Note: Tibbles are newer than data.frames and some old functions will not work with them and might require actual data.frames; For the majority of our analyses and visualisations we will work with tibbles.

5.3 Exercises:

1. Which data types are present in our dataset?
2. How many observations and variables are in our dataset?
3. What are the variables?

6 A recipe for generating graphs with ggplot2

In the following section, we will generate our first plots using the `ggplot2` package. Data visualisation in `ggplot2`, however simple or complex, follows a general recipe:

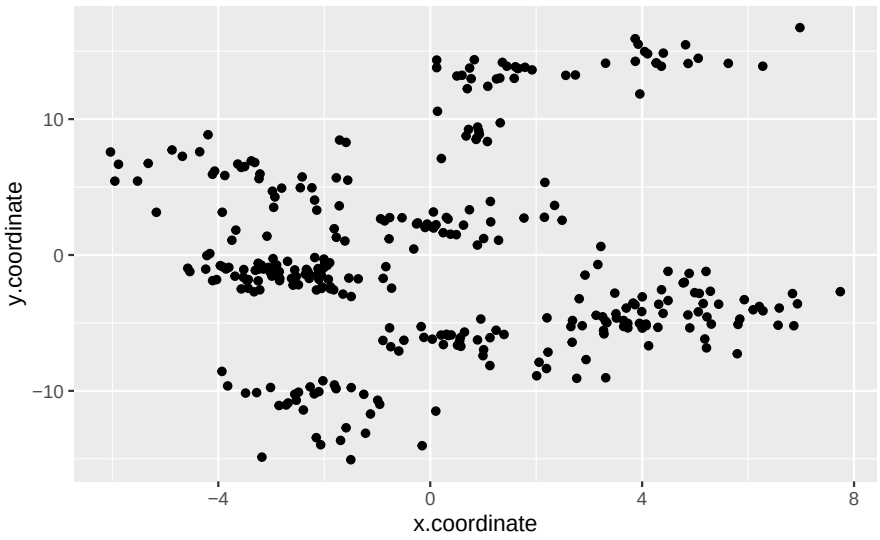
1. Setting up a coordinate system with the function `ggplot()`: provide the dataset to use in the graph
2. Adding layers with `geom_xxx()` functions:

- layers are quite literally added to `ggplot()` object, by using the `+` operator;
- each geom function expects a mapping argument which defines how variables are mapped to visualisation. The mapping argument is provided with `aes()`, where the x and y arguments describe which variables to map to the x and y axes. ggplot2 looks for the mapped variables in the data argument to `ggplot()`;
- There are many geom functions that each add a different type of layer to a plot. Their names are very descriptive, for instance:
  - `geom_point` adds a layer of points to the coordinate system, effectively creating a scatterplot;
  - `geom_histogram` adds a histogram layer;
  - `geom_boxplot` adds a boxplot layer.

Note: The add operator `+` can never be at the start of a line! When adding multiple layers to a plot, we will always end the layer line with the `+` sign, never start a new layer with a `+`.

6.1 Our first plot

```
p <- ggplot(data=coord)
p + geom_point(mapping=aes(x=x.coordinate, y=y.coordinate))
```



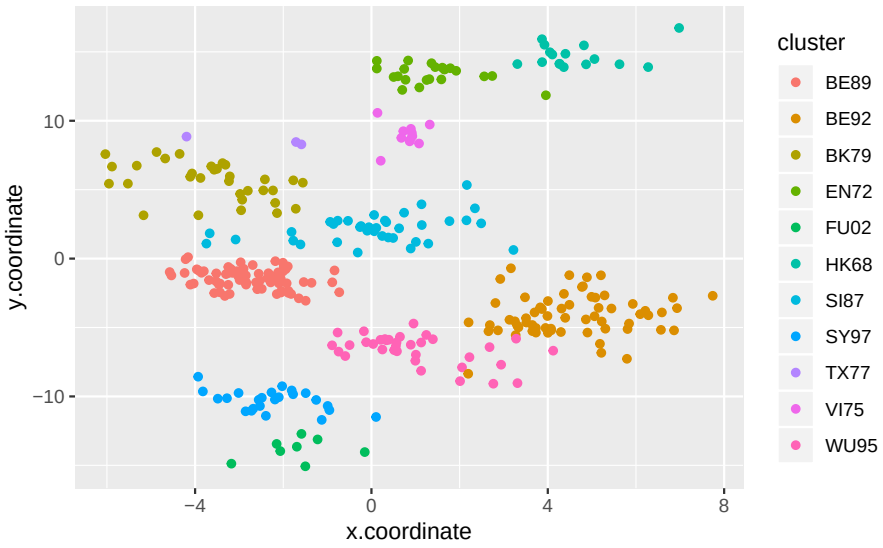
6.2 Exercises

1. Run ggplot(data = coord). What do you see?
2. What makes this simple plot look very different from the map that we want to achieve?
3. What other information in our data object `coord` could we use?

6.3 Mapping additional aesthetics

To map additional information onto our 2d scatter plot, ggplot2 makes use of aesthetics. We have already seen aesthetics in the example above, where we mapped the `x.coordinate` and `y.coordinate` to the x- and y-axis using `aes(x=x.coordinate, y=y.coordinate)`. Broadly speaking, aesthetics are the visual properties of the objects in your plot. They include for instance the size, shape, or color of your points. The different flavors of an aesthetic are called levels. The levels in the shape aesthetic are for instance round, triangular and square. Levels of the color aesthetic could be blue, red and yellow. In our graph above, we have not used any of these aesthetics yet. Let's start by introducing color to the plot. As in the original publication, we can color the points in our plot by cluster name. We do this by simpling specifying the color aestetic in the mapping:

```
p + geom_point(aes(x=x.coordinate, y=y.coordinate, color=cluster))
```



`ggplot2` automatically assign a unique color level to each unique value of cluster. This assigment process is called scaling. Depending on data and aesthetic, `ggplot2` selects a reasonable scale and constructs a legend that explains the mapping between levels and variable values, in this case color and cluster. However, we can also provide our own color-scheme.

6.4 Setting scales

Adding scales to `ggplot` objects follows the same scheme as adding layers: we add the scale to the existing object by the `+` operator. Similar to `geom_xxx`, we have `scale_xxx_yyy`: xxx specifies the aestetic for which we are providing the scale, yyy specifies the type of scale we want. For instance:

- `scale_color_continuous` sets a continuous scale for the asthetic color;
- `scale_shape_discrete` sets a discrete scale for the asthetic shape.

When selecting a scale, we need to consider what type of data we are displaying and what message we want to convey:

- qualitative data: unordered, distinct categories, as in our example cluster names;
- sequential data: ordered data that progresses from low to high, as in our example 'year of isolation';
- diverging data: data from low to high, with emphasis on mid-range values as well for instance correlations that range from -1 to 1, where the mid-range around 0 ie no correlation are equally important to be visualised

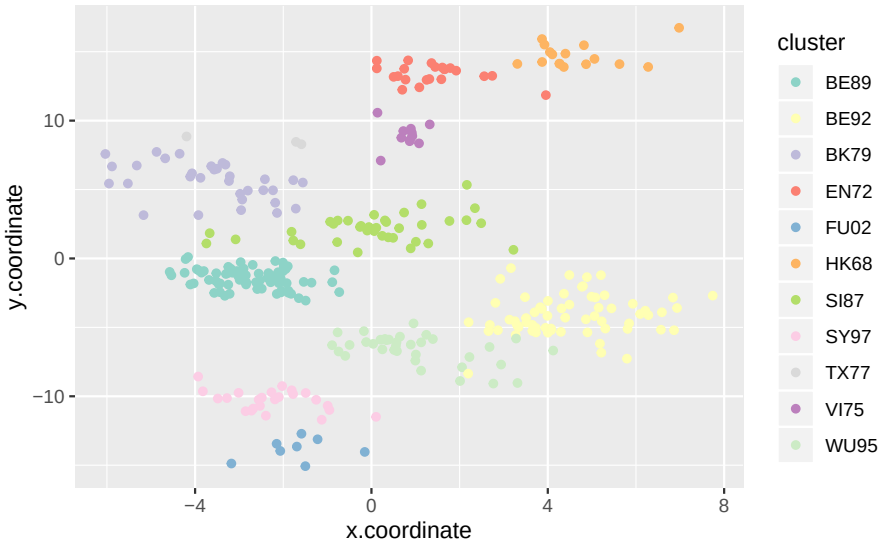
6.4.1 Color scales

The colorbrewer website provides a great resource to pick appropriate color scales.

`ggplot2` has direct access to these color schemes:

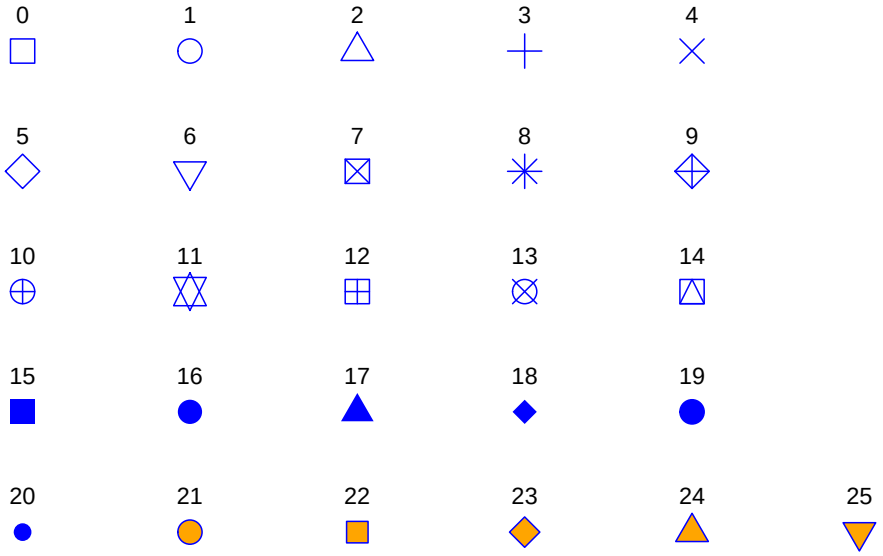

YlOrRd
YlOrBr
YlGnBu
YlGn
Reds
RdPu
Purples
PuRd
PuBuGn
PuBu
OrRd
Oranges
Greys
Greens
GnBu
BuPu
BuGn
Blues
Set3
Set2
Set1
Pastel2
Pastel1
Paired
Dark2
Accent
Spectral
RdYlGn
RdYlBu
RdGy
RdBu
PuOr
PRGn
PiYG
BrBG

```
p +
  geom_point(aes(x=x.coordinate, y=y.coordinate, color=cluster)) +
  scale_color_brewer(type="qual", palette = "Set3")
```



6.4.2 Shape scales

There are 25 point shapes available in R:



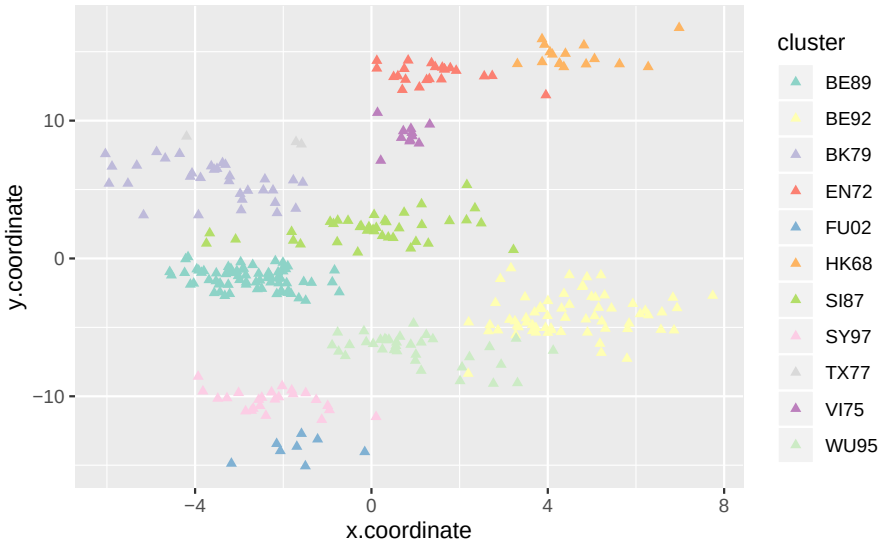
We can use the shape aesthetic and scale in analogy to how we specified color.

Note: Shapes 0-20 work in conjunction with the color aestetic, shapes 21-25 with the color and fill aesthetic.

6.5 Manual aesthetics

In addition, we could also decide that we would like to display all our points in the data set as number 17 triangles. To set an aesthetic manually, you move it outside the mapping argument and specify the level:

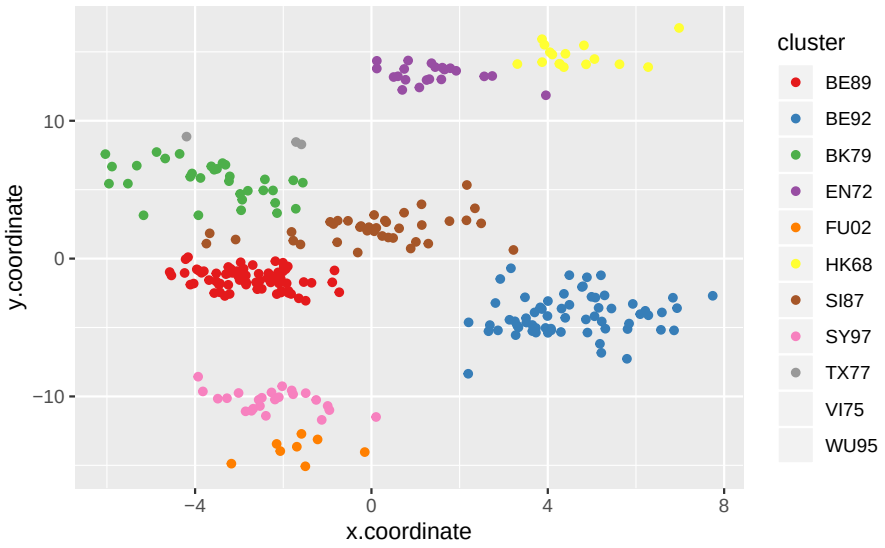
```
p +
  geom_point(aes(x=x.coordinate, y=y.coordinate, color=cluster), shape=17) +
  scale_color_brewer(type="qual", palette = "Set3")
```



6.5.1 Exercises

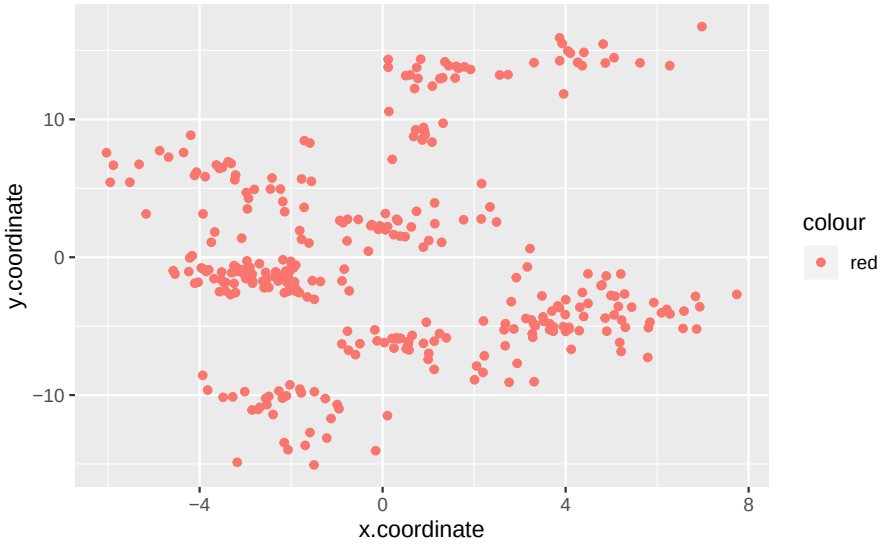
1. Try changing the cluster aesthetic to `size` and `shape`. Does this convey the same level of information as a color scale?
2. What other variable in our dataset would be well represented by a `shape` scale? Add a shape aesthetic for the variable you identified.
3. Generally speaking, which type of data lends itself to shape scales, which to size, which to color?
4. Why does this not work?

```
p +
  geom_point(aes(x=x.coordinate, y=y.coordinate, color=cluster)) +
  scale_color_brewer(type="qual", palette = "Set1")
#> Warning in RColorBrewer::brewer.pal(n, pal): n too large, allowed maximum for palette Set1 is 9
#> Returning the palette you asked for with that many colors
#> Warning: Removed 46 rows containing missing values (geom_point).
```



5. Why does this code not color all points in red?

```
p +
  geom_point(aes(x=x.coordinate, y=y.coordinate, color="red"))
```



6. Advanced: Change the overall shape of points to number 24 triangles, than color by cluster and fill by type.

6.6 Setting labels and themes

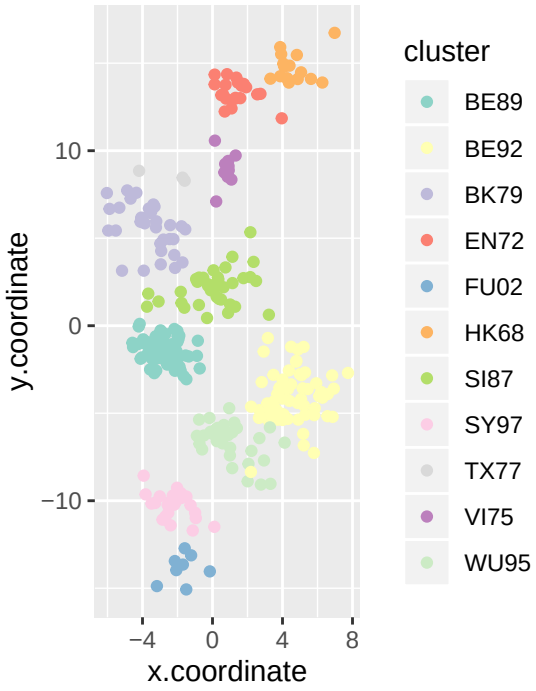
So far, we have been concerned with the data layers of the plot, using `geom_xxx` to visualise different variables and `scale_xxx_yyy` to customise them. In the following section, we will have a look at customising the 'canvas' of the plot, i.e. the background, axis labels etc.

6.6.1 Coordinate system

Looking at the antigenic map (Figure 1 in the Smith *et al* (2004) paper), we notice that their axis ratio of 1:1, ensuring that one unit on the y-axis is equivalent to one unit on the y-axis. Our plot has a different ratio.

Based on the range of the `x.coordinate` and `y.corrdinate` variable, `ggplot` automatically chose the axis limits and more importantly here, their ratios. We can easily change this default by adding a `coord_xx` layer, in this case `coord_fixed`, which will ensure a 1:1 ratio per default.

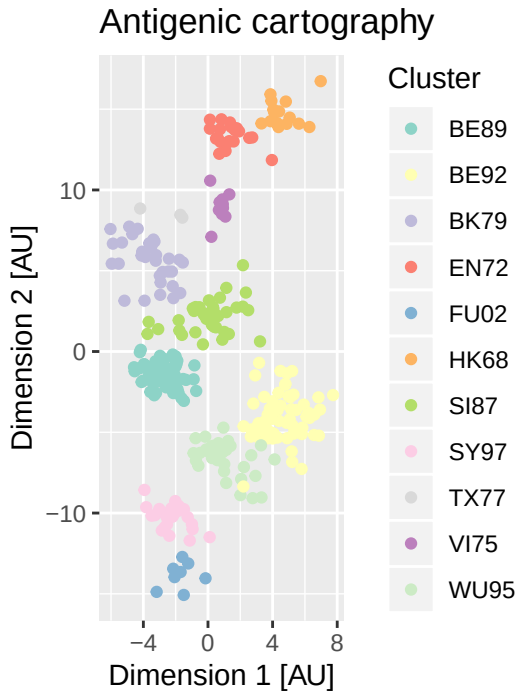
```
p +
  geom_point(aes(x=x.coordinate, y=y.coordinate, color=cluster)) +
  scale_color_brewer(type="qual", palette = "Set3") +
  coord_fixed()
```



6.6.2 Labels

Currently, the axis and legend labels in our plot are simply the name of the variables we mapped to the aesthetics. We can change the axis labels by providing arguments to the `labs` layer. Specifically, x and y axis labels can be set by the `xlab` and `ylab` arguments respectively, the title is specified by the `title` argument. To change the name of a legend, we have to specify the aesthetic we mapped it to, in this `color`.

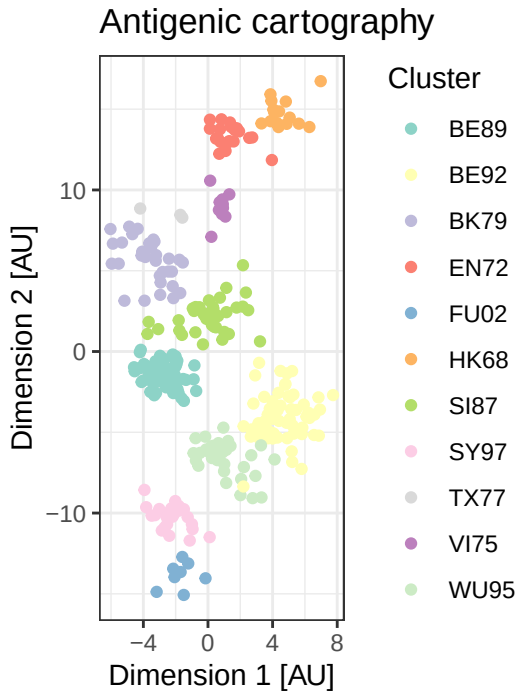
```
p +
  geom_point(aes(x=x.coordinate, y=y.coordinate, color=cluster)) +
  scale_color_brewer(type="qual", palette = "Set3") +
  labs(x="Dimension 1 [AU]",
       y="Dimension 2 [AU]",
       title="Antigenic cartography",
       color="Cluster") +
  coord_fixed()
```



6.6.3 Themes

Finally, we can customise the non-data components of our plot. While the developers of `ggplot2` had a strong preference for the default grey background, this is not to everyone's liking. As with everything else you've seen so far, there are some build-in options for customising. Here, we use the `theme_bw` to change to a white background.

```
p +
  geom_point(aes(x=x.coordinate, y=y.coordinate, color=cluster)) +
  scale_color_brewer(type="qual", palette = "Set3") +
  labs(x="Dimension 1 [AU]",
       y="Dimension 2 [AU]",
       title="Antigenic cartography",
       color="Cluster") +
  coord_fixed() +
  theme_bw()
```



6.6.4 Exercises

1. What other coordinate system options exist? Hint: type `?coord_` in a new chunk and press tab to see other options.
2. Rename the legend title for the `shape` aesthetic
3. Test different themes and see how it effects the plot, for instance use `theme_void`, `theme_dark` and `theme_classic`. Similar to Exercise 1, you can type `?theme_` and tab to see other possible build in themes.
4. Why does this not work?

```
p +
  geom_point(aes(x=x.coordinate, y=y.coordinate, color=cluster)) +
  scale_color_brewer(type="qual", palette = "Set3") +
  labs(x="Dimension 1 [AU]",
       y="Dimension 2 [AU]",
       title="Antigenic cartography",
       color="Cluster") +
  coord_fixed()
  + theme_light()
```

6.7 Saving your plots

You can save your plots with the `ggsave` function. `ggsave` will save the most recent plot to your project directory. As argument we only have to provide the name of the file we want to save it to. `ggsave` will determine the format of the output file based on the file ending of the filename that you provide. For instance, the code chunk below will save our most recent plot to a pdf document named "antigenic_cartography.pdf".

```
ggsave(filename="antigenic_cartography.pdf")
#> Saving 6 x 3.71 in image
```

Not only does `ggsave` determine the format of the file from the name, it also determines the size of the file from the size we chose for displaying the plot in out analysis. To make it reproducible, it is good practice to specify the size and units.

```
ggsave(filename="antigenic_cartography.pdf", width=5, height=8, unit="cm")
```

Note: `ggsave` overwrites the previous file of that name without warning!

6.7.1 Exercises

1. Save the plot as png and jpeg.
2. Change the size of the plot with width and height; what happens to figure labels and legends?

7 Beyond scatter plots

In **section 4**, we have learned the basic recipes for generating a plot with `ggplot2`. We created a 2D scatter plot, encoding visual information in a color and shape scale and made sure we convey the right message by ensuring appropriate labels and coordinate systems.

In the following section, we will get to know a couple more plot types.

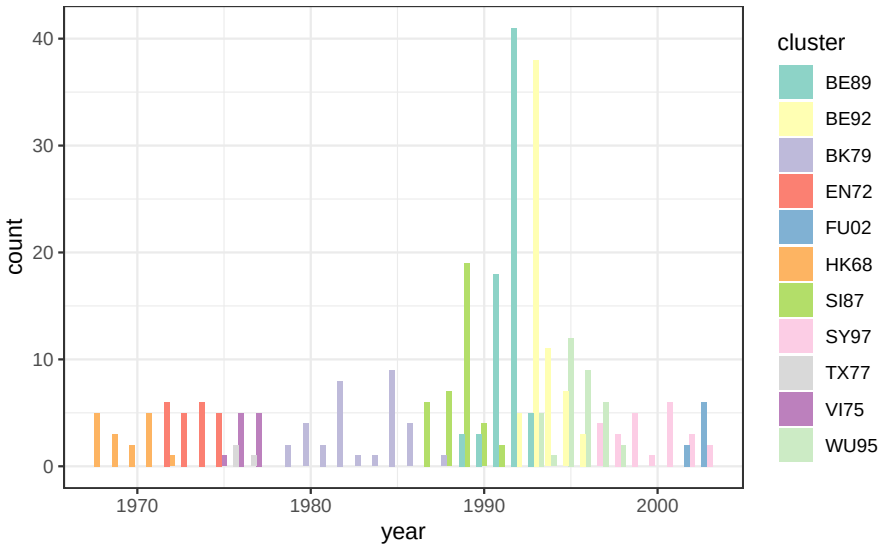
7.1 Bar charts and histograms

Bar charts and histogram visualise the number of observations (count) for a specified variable, typically with the variable on the x-axis and the count on the y-axis. For histograms, the x-axis is divided into bins and the number of observations in each bin is counted. Bar charts are a special case of histogram, where the bin width is 1, i.e. the counts at each value of the variables are displayed. Bar charts are best described by the calling the help function `?geom_bar`:

```
There are two types of bar charts: geom_bar() and geom_col(). geom_bar()
makes the height of the bar proportional to the number of cases in each
group [..]. If you want the heights of the bars to represent values in the
data, use geom_col() instead. geom_bar() [...] counts the number of cases at
each x position. geom_col() [...] leaves the data as is.
```

In the following, we plot the number of antigen measurements per year and color them by cluster. As we want the plotting function to figure out the counts, we use `geom_bar`. For consistency to our previous plots, we use them same theme and color scheme; note, we use `geom_fill` (not `geom_color` as above) to fill the bars with the specified color. In addition to the aesthetics, we also specify positions for our bar chart. To display bars next to each other, we have to specify `position_dodge`, and setting `preserve="single"`, keeps a constant width for each bar. More on this in the exercises to this section.

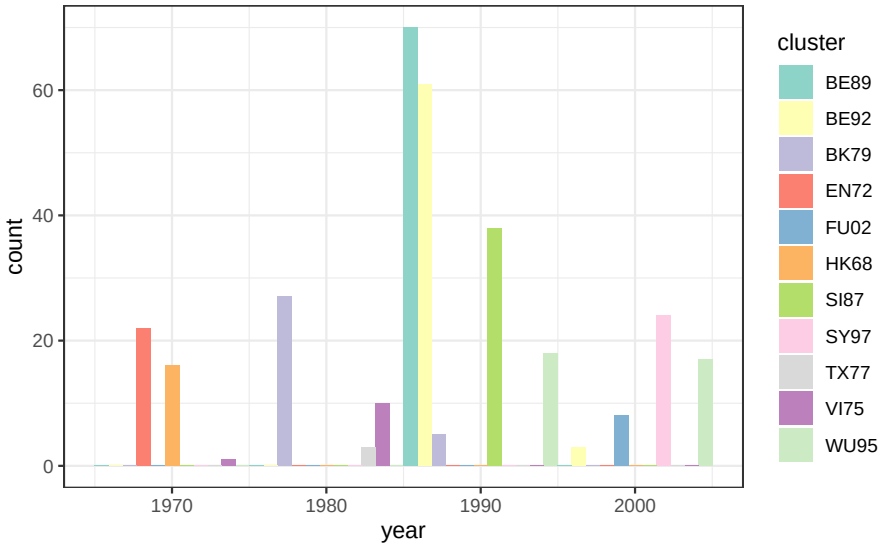
```
p + geom_bar(aes(x=year, fill=cluster),
             position=position_dodge(preserve="single")) +
  scale_fill_brewer(type="qual", palette = "Set3") +
  theme_bw()
```



We can use the same set-up with `geom_histogram`, to visualise the distribution of antigenic measurements in larger time intervals. The bin width specifies the range over which to summarise the variable. Again, the help function gives good insight with `?geom_histogram`:

```
binwidth    [...] You should always override this value, exploring multiple
            widths to find the best to illustrate the stories in your data
            [...].
```

```
p + geom_histogram(aes(x=year, fill=cluster),
                   position=position_dodge(preserve="single"),
                   binwidth = 10) +
  scale_fill_brewer(type="qual", palette = "Set3") +
  theme_bw()
```



7.2 Boxplots

Below, we take a different look at the distribution of antigenic measurements by year, using a boxplot.

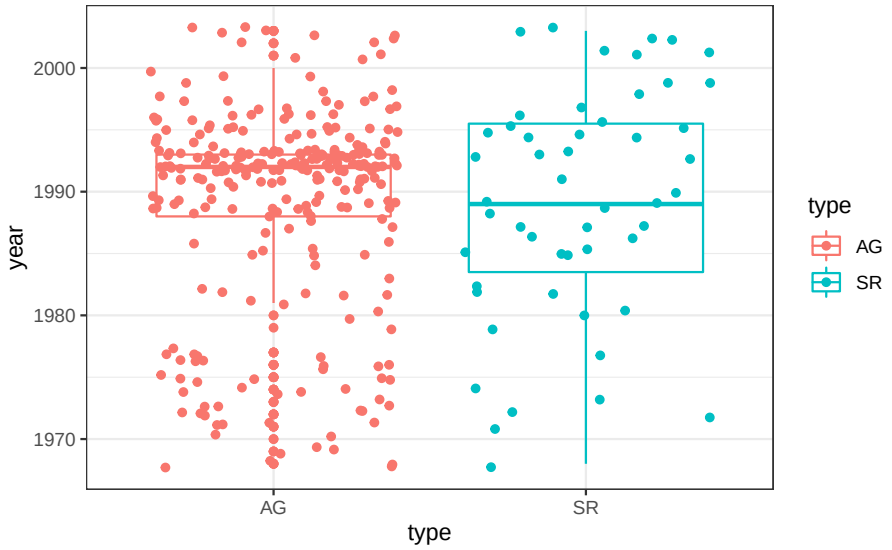
"The boxplot compactly displays the distribution of a continuous variable. It visualises five summary statistics

(the median, two hinges and two whiskers), and all"outlying" points individually." [`?geom_boxplot`].

We treat `year` as a continuous variable and show its distribution split by the type of measurement.

To show both the summary of the distribution (boxplot) and the actual data, we can add a `geom_jitter` to the plot. It plots the original y-values and adds a jitter to the x-values to avoid overplotting.

```
p + geom_boxplot(aes(x=type, y=year, color=type)) +
  geom_jitter(aes(x=type, y=year, color=type)) +
  theme_bw()
```



7.3 Geographical maps

So far, we have worked with visualisations of continuous, discrete and categorical variables. We have worked with the `year` (discrete/continuous), `cluster`, `type` (both categorical), `x.coordinate` and `y.coordinate` (both continuous) variables in out data set and used these to display the antigenic maps and distribution of measurements across time, separated by both cluster and type. For all these observations, we have an additional variable, the location of data generation. In the following, we will see how we can use `ggplot2` to visualise geographic data.

First, we read a file that contains the coordinates (latitude and longitude) of all locations that we find in the `location` of our coord object. We then use the `ne_countries` of the `rnaturalearth` package that we loaded in the beginning to create an object that countains coordinates of all countries.

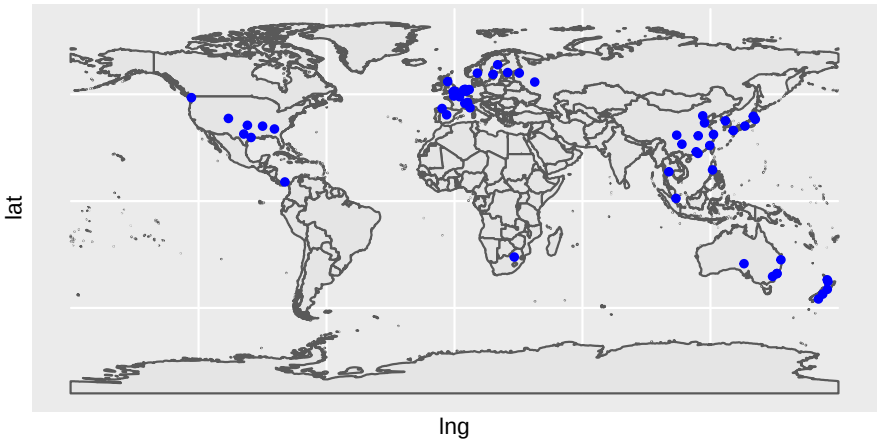
As with above, we will then create a `ggplot` object. So far, we provided the data that we want the visualisation to be applied to in the `ggpot` call. In this case, we have two different data sets that we want to visualise, the world map and the locations. In this case, we can also specify the appropriate dataset to each geom separately.

We use the `geom_sf` to display the world map and a points layer with `geom_point` to visualise the locations.

```
locations <- read_csv("locations.csv")
#> Parsed with column specification:
#> cols(
#>   location = col_character(),
#>   name = col_character(),
#>   lat = col_double(),
#>   lng = col_double()
#> )
```

```
world <- ne_countries(scale = "medium", returnclass = "sf")

g <- ggplot()
g + geom_sf(data = world) +
  geom_point(data=locations, aes(x=lng, y=lat), color="blue")
```



7.4 Exercises

1. Test different options for the `position` argument of `geom_bar`. Hint: use the Details paragraph in `?geom_bar` to find a description about possible options.
2. What happens when you choose `preserve="total"` in `position_dodge` of `geom_histogram`?
3. Customise the color scale and plot labels in the boxplot showing the distribution of measurements per type and year. What happens if you choose `aes(fill)` instead of `aes(color)`?
4. Change `geom_jitter` to `geom_point` to see why zgeom_jitter is a better visualisation of the data. Go back to using `geom_jitter` and play with the `width` argument to customise your plot.
5. What would be a good `theme` for the world map? Add it to the plot.