Dynamic Documents and Reproducible Figures with RMarkdown

PlainText Working Group

Abstract

An example RMarkdown document for the participants at the third meeting of the PlainText Working Group

In this document we are going to learn the principles of good figure design and how to actually make figures in R with `ggpplot`. You are going to be interacting with the document and running some code, so that means you should be looking at the `.Rmd` and NOT the PDF file. We are going to be rendering to PDF but working in what is called the source file, the `.Rmd` file.

What is RMarkdown

An RMarkdown file is a plain text file that allows you to include R (and python) code along side the prose of your argument. It isn't just that the code is written in your document (in fact you often don't want that) but that the code is included do load data, manipulate it, creat figures and models from it and include those figures and model results directly in your document. That means that when the underlying data changes, you don't have to rerun your code to update the figures and copy and past it into your document. It auto-magically updates.

The most important thing a document format like this is used for is creating reproducible findings and ensuring that the process you used to get from your data to your insights is reproducible for your friends, colleagues and critics that might have some questions about why your scatter plot looks the way it does, or why a particularly decsive coefficient is both displays a large effect size and is signficant.

This is the capilaries flowing through the heart of the PlainText Working Group: making sure that your arguments are rendered beautifully in a document that can readily prove to others that you conscientiously executed good social scientific research.

Code Chunks

The code that you include in your documents goes into what are referred to as "code chunks", delimited areas in your document that indicate to your computer that there is code to be run through an interpreter rather than just rendering the alphanumeric characters as prose for your readers to read. The special characters that are used to indicate that you are dealing with a code chunk look like this:

That three backticks (“‘) followed by curlbrackets which contain an `r`. The `r` tells the computer what language you are writing code in. RMarkdown supports both R and python and I am told stata but it should assume that you are running R.] Each code chunk has a few basic options you need to decide about including:

- `include = FALSE` prevents code and results from appearing in the finished file. R Markdown still runs the code in the chunk, and the results can be used by other chunks.
- `echo = FALSE` prevents code, but not the results from appearing in the finished file. This is a useful way to embed figures.
- `message = FALSE` prevents messages that are generated by code from appearing in the finished file.
- `warning = FALSE` prevents warnings that are generated by code from appearing in the finished.
- `fig.cap = "..."` adds a caption to graphical results.

As a general rule, particularly when you are dealing with figures you want to set `echo` to false so that your figure appears but that the code that generates it, does not. Fortunately, we can set the defaults for all our code chunks in a single document within a code chunk at the top of the document (this is seperate from your YAML header).

```
require(knitr)
```

```
## Loading required package: knitr
```

```
opts_chunk$set(echo = FALSE,
               results = "hide",
               message = FALSE,
               warning = FALSE,
               fig.path = 'figures/fig-',
               cache.path = 'cache/report-',
               dev = "png",
               dpi = 300)
```

There are a few things to note about this code chunk. For one it is *named*: directly after the language is declared with `r` there is the word `Setup` which is the name of the code chunk. Then there are the actual options for this "Setup" code chunk and then the R argument `opts_chunk$set` which actually sets the default chunk options for the rest of the code chunks, so you don't have to include those arguments later.

Prose

Everything else in your document is what I think is reasonable to call "Prose", that is just like you would in a regular Markdown file, you use all the alphanumeric characters to create natural language an argument that you would in any plain text file. There are naturally all the bells and whistles that you will need to be aware of for representing style and including citations and we have covered or will cover soon. Here is a table that shows some of the basic Markdown syntax for review.

Element	Markdown Syntax

Heading	`# H1`
	`## H2`
	`### H3`

Bold	`**bold text**`
Italic	`*italicized text*`
Blockquote	`> blockquote`
Ordered List	`1. First item` `2. Second item`

Element	Markdown Syntax
	`3. Third item`
Unordered List	`- First item`
	`- Second item`
	`- Third item`
Horizontal Rule	`---`
Link	`[title](https://www.example.com)`
Image	`![alt text](image.jpg)`

Citations

Another key part of academic writing is including citations to other works that inform your research. To do this you will need to use a `.bib` file which is generated using a bibliographic software like Zotero or Mendeley. I highly recommend use a `.bib` file because it saves a lot of time formatting and creating the proper list of cited references.

To include citations in your `.Rmd` file you will need to specify where your `.bib` file is located by specidying its path in the YAML header at the top of the document. We have an example `.bib` file included in the directory that contains this source file called `plaintext.bib` with just a few sources in it. You then reference an item in that `.bib` file by writing an @ symbol followed by the "citation key" of the item you want to cite, like this `@durkheim_division_2008` which renders a citation like this (Durkheim [1893] 2008). When you include a `# References` section at the bottom of your rmarkdown or markdown document, it will automatically create the bibliography with only the citations you actually use in your document.

There are a variety of ways in which you can customize your bibliography so it matches the proper format of whatever journal you might be submitting to. That will require that you use something called a `CSL` file.

Loading Data

We are going to be using a few pieces of data from a few different places that I have collected for your use here and this is the first lesson. Load your data upfront, transform it, then you can use it anywhere else in your docunent. We aren't going to be doing that here because I want to demonstrate what we are doing as we go, but as a general rule when you work with your documents, include the calls to the libraries you need in a single code chunk, when you load data and create functions when you need and inlcude comments.

`ggplot`

`ggplot` is a library in R that was designed on the basis of a book that was written in the 1990s called *The Grammar of Graphics*, thus the "gg" in `ggplot`. It is the lingua franca of figure creation in R and is what we are going to be using here. There are also other libraries that have adopted the `ggplot` conventions and applied them to other kinds of figures not covered in the original package like `ggtree` for phylogenetic plots, `ggnet` for network data, and `ggrides` for ridge plots.

I'll be referring to "plots" and "figures" interchangably.

There is a *grammar* to `ggplot` that once you acquire will allow you to make all sorts of compelling and visually parsimonious plots. The basic setup is thus:

1. Call the `ggplot` function and feed it data in the `data` argument, where the "data" in the following example is a dataframe with all the data you want to plot.

```
ggplot(data = dataframe)
```

2. Tell `ggplot` what you want mapped to aesthetic properties of the plot with the `aes` argument. At a minimum you will likely want to give the aesthetic mapping both an "x" and a "y" value (a column in the dataframe) to the respective axis of the plot. This isn't the case if you are making a histogram or density plot but we will cover that shortly. This defines the basic structure of the plot and you can then layer on additional properties.

```
ggplot(data = dataframe, aes(x = Variable_X, y = Variable_Y))
```

3. Call a `geom` which defines the kind of plot you want to make. For example a line plot is made with `geom_line`, a histogram with `geom_histogram`, a scatter plot with `geom_point`, and a bar plot with `geom_bar`. You "add" `geoms` to the base plot that you defined in step two using a literal addition symbol (+) at the end of the line where the base plot is defined.

```
ggplot(data = dataframe, aes(x = Variable_X, y = Variable_Y)) + geom_line()
```

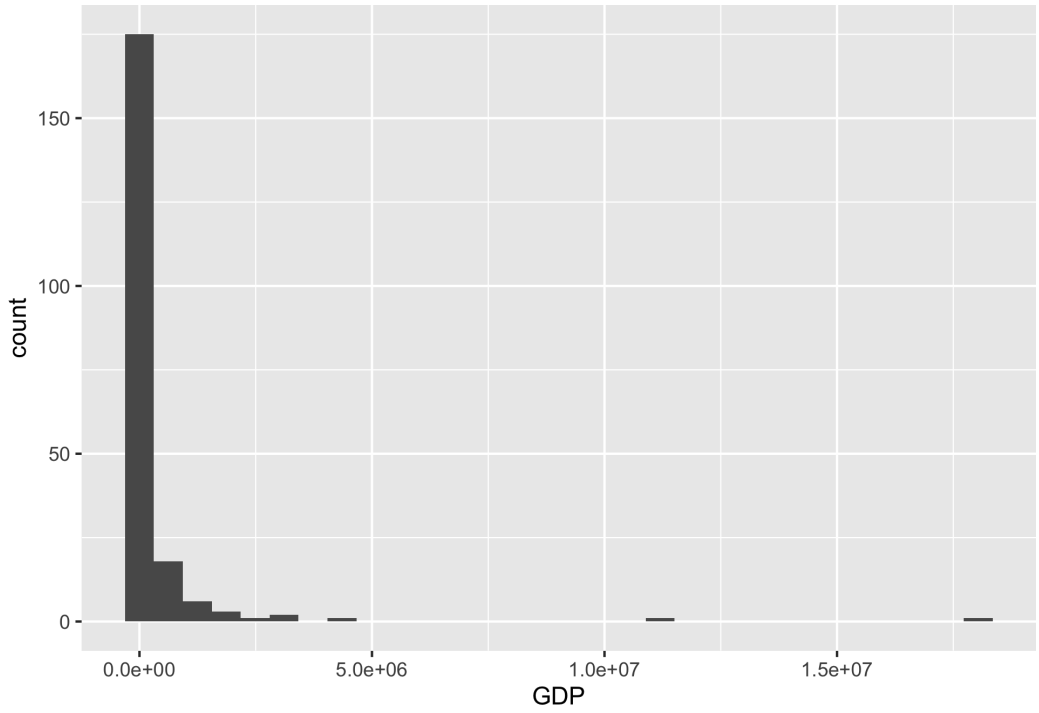
That is the fundamental structure of any plot or figure you can make with `ggplot` and there is a lot more complexity to add using different `geoms` and aesthetic mappings that we will cover shortly. And we will start that by actually creating some plots to "learn by doing".

Distributions

When we start to look our data with visualizations we often want to know the distribution of values that the data can take. We usually do this with histograms, to count the number of times a given value appears. We are going to use some data from the United Nations to demonstrate this. The UN dataset has infoarmation about every member country such as the landmass, GDP, population, infant mortality and some attributes of the country's economy.

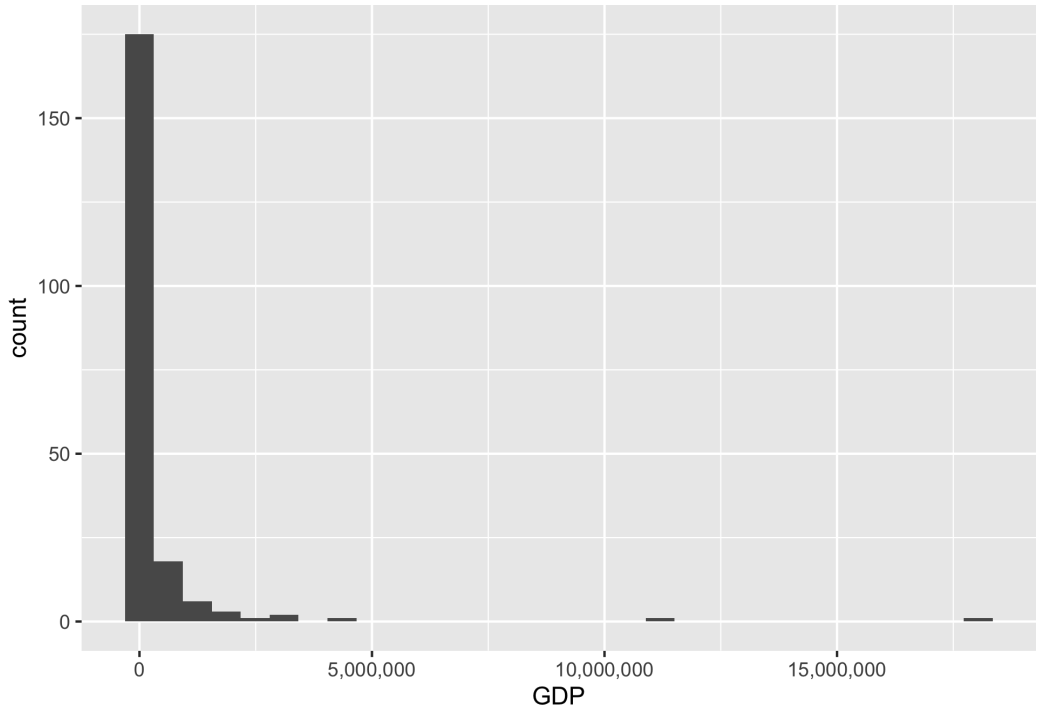
Here we will use the grammar we learned above to start building a plot, by calling `ggplot` telling it what data we want to use, and providing an aesthetic mapping for the X and Y axes and calling a `geom` to create the plot. In this case we are going to be using the `geom_histogram` function to create a histogram. Because a histogram only counts the values in a single variable at a time, when we tell `ggplot` an aesthetic mapping with the `aes` function we only give it an X axis variable.

```
ggplot(data = un, aes(x = GDP)) + geom_histogram()
```



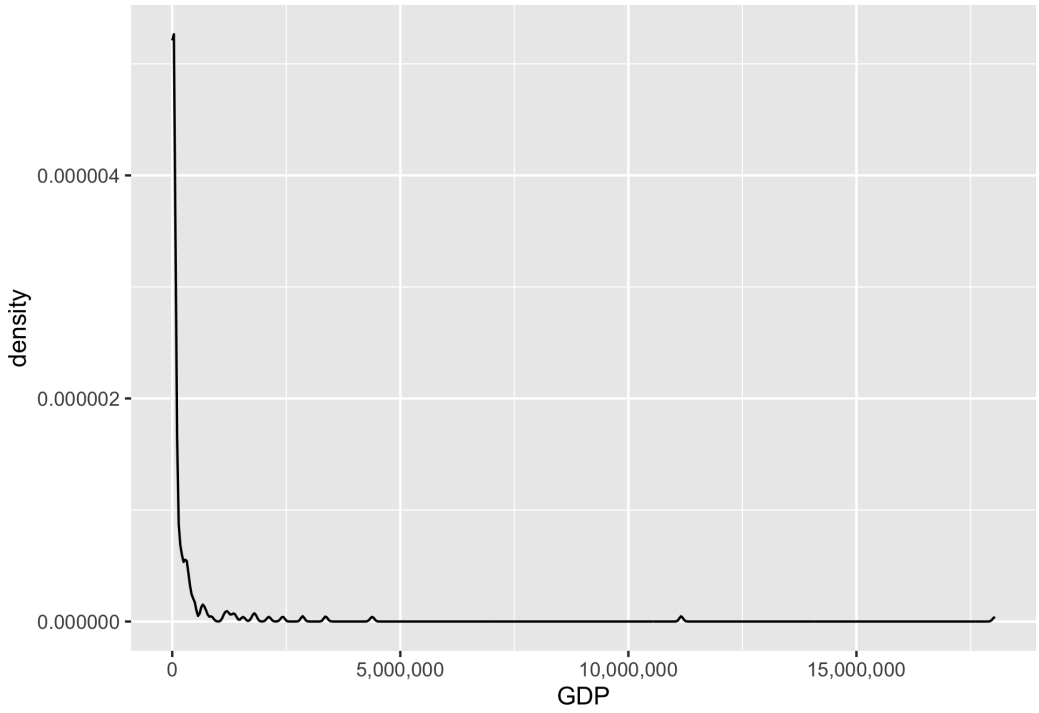
A histogram gives some immediate and information about the distribution: in this case it shows that GDP is highly positively skewed, with most countries falling on the left tail of the X axis and a very long sparse right tail. But there are some issues with this plot, mainly the fact that the X axis is not very interpretable as the values are reported in scientific notation rather than as integers. We can change that by telling `ggplot` to report the values as integers, and we do this with the `scale_x_continuous` function.

```
ggplot(data = un, aes(x = GDP)) + geom_histogram() +
  scale_x_continuous(labels = scales::comma)
```



Another way of visualizing distributions is by using the `geom_density` function which uses a kernel density estimate to produce a smooth curve over the observed frequencies of values in the data set. When using it the Y axis is a little harder than in a histogram, where the Y axis is simply the count of the observed value, but it is typically smoother and helps to "see" the distribution better. It is, in part, a matter of taste.

```
ggplot(data = un, aes(x = GDP)) + geom_density() +
  scale_x_continuous(labels = scales::comma) +
  scale_y_continuous(labels = scales::comma)
```



Trends Over Time

One of the primary things we want to look at, particularly with longitudinal data, is what happens as time goes by. To demonstrate this we will be using data about organ donation in Organisation for Economic Co-operation and Development (OECD) countries. The data set includes information on 17 different countries in the OECD over 13 years and rates of organ donation and several other metrics including GDP, population, whether one opts into or out of organ donation, and the number of motor vehicle deaths per capita.

To begin lets look at the focal variable in this dataset `donors` which capture the rate of organ donation per million people in a country's population over time (the `year` variable). We can do this with a simple line plot, using the `geom_line` function in `ggplot`. In this first plot we are going to be mapping the rate of organ donation (the column/variable `donors`) to the Y axis and the year in which the observation occurred (`year`) to the X axis to show change in rates over time.

```
ggplot(data = organdata, aes(x = year, y = donors)) +
    geom_line()
```

In R you can arbitrarily break up lines of code across lines to keep it "tidy". The style conventions in R is to keep lines under 80 characters long, that way when you view them in a text editor, they are completely visible without scrolling horizonatally.

This is not as informative as we would have liked it to be because there are multiple countries in the dataset and so multiple countries value's are getting mapped onto the same year, making those awkward vertical lines with diagonal connectors. What we need to do is use another aesthetic mapping to tell `ggplot` to group some of the rows of the data frame and plot them across the X axis as single lines. We can do this by adding an argument to the `aes` function inside the call to `ggplot`

```
ggplot(data = organdata, aes(x = year, y = donors, group = country)) +
  geom_line()
```

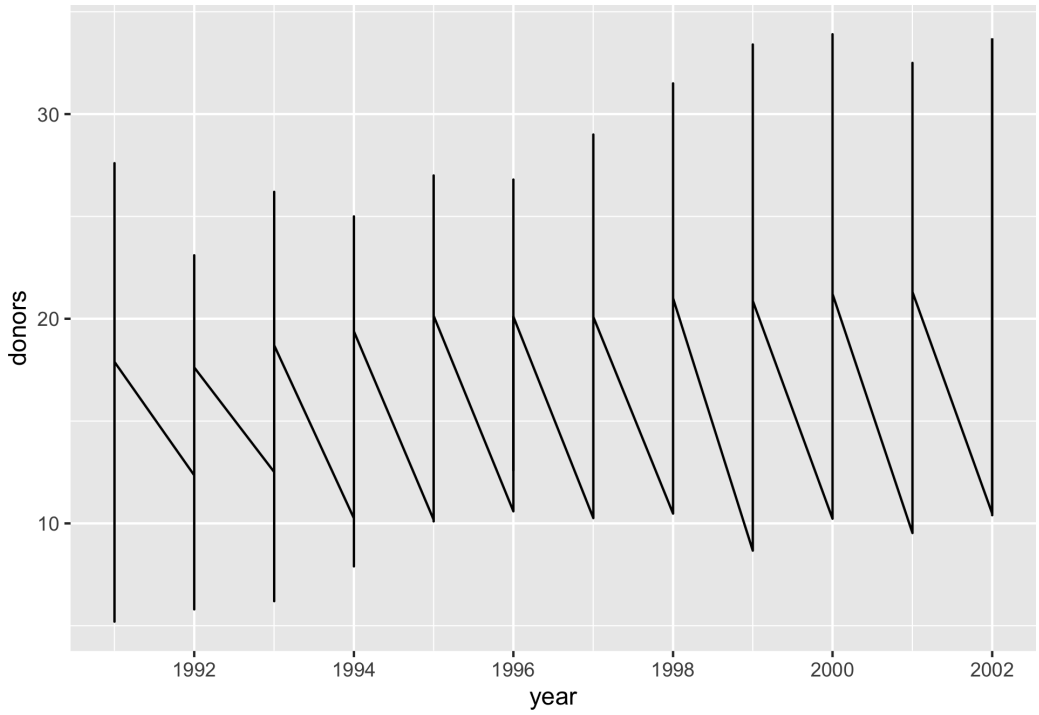
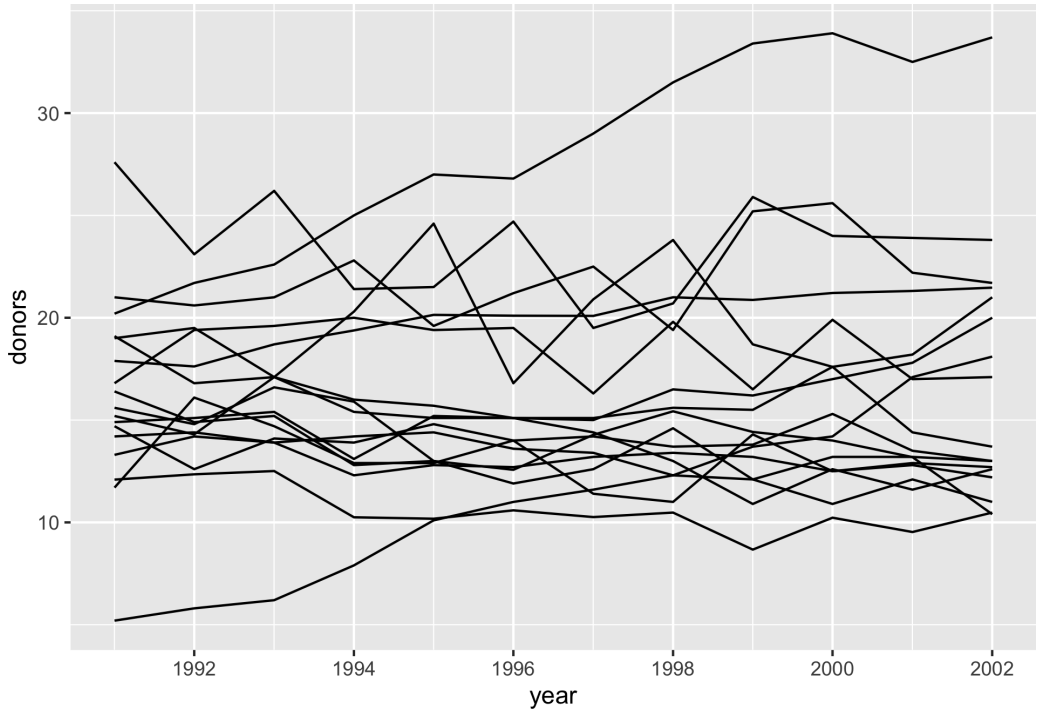


Figure 1: Something seems amiss here.

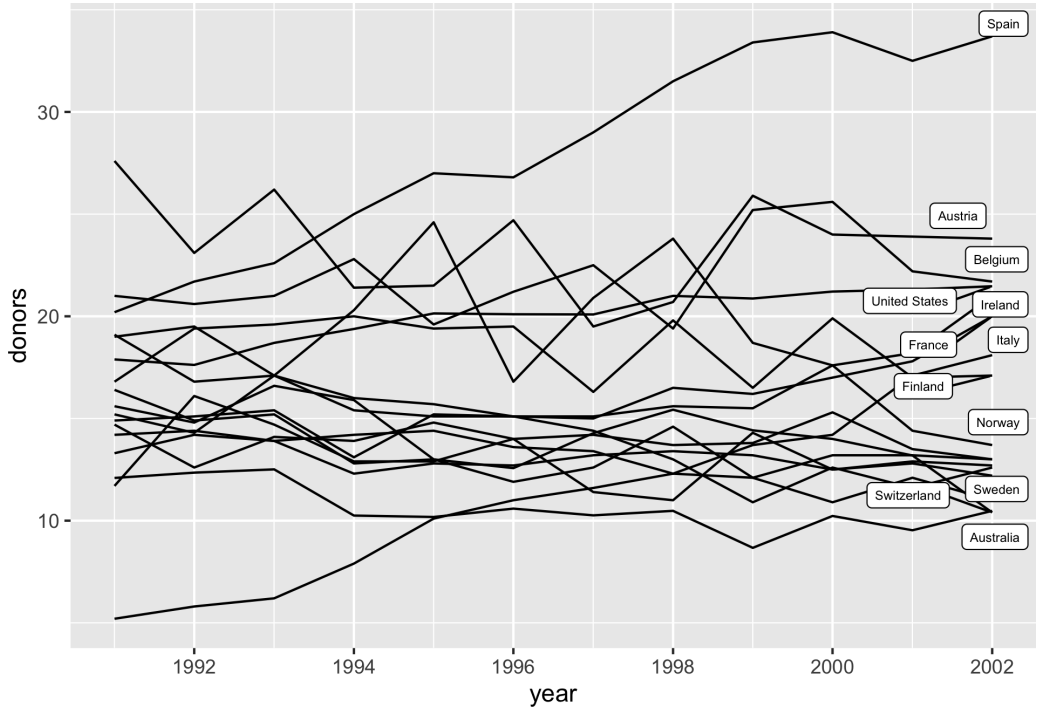


This figure is better because it actually shows the different countries change in rate of organ donation over time but the lines aren't labeled in any way so we don't know who is where and what they are doing. To do this we need to do a little data transformation and use a new `geom` to

add an extra layer to our plot. geom_label_repel is not in the base version of ggplot but in another library called ggrepel that we loaded in the "Data" code chunk above.

```
# Find the end of each line by looking at the highest value
# for each country in the variable mapped to the Y axis.
data_ends <- organdata %>%
  group_by(country) %>%
  top_n(1, year)

# Then add it to the plot using geom_label_repel
ggplot(data = organdata, aes(x = year, y = donors, group = country)) +
  geom_line() +
  geom_label_repel(aes(label = country), data = data_ends, size = 2)
```



Now we can see what is going on a little bit better but the interpretability of the plot is still pretty hard. It is pretty clear that Spain is not behaving like the other countries in the data set and that there looks like there is two other distinct groups of countries, with the United States in one and Sweden in another. To help your reader grasp the plot immediately it is best to provide informative axis labels. The default behavior of `ggplot` is to just use the variable that is mapped to that axis as the label but we can specify what to actually call it with another layer to the plot. This time it isn't a `geom` function we call but a unique function called `labs` which allows you to add X and Y axis labels, a title, subtitle.

```
ggplot(data = organdata, aes(x = year, y = donors, group = country)) +
  geom_line() +
  geom_label_repel(aes(label = country), data = data_ends, size = 2) +
   labs(y = "Donation Rate Per Million",
        x = "Year",
        title = "Rate of Organ Donation Over Time (OECD)")
```

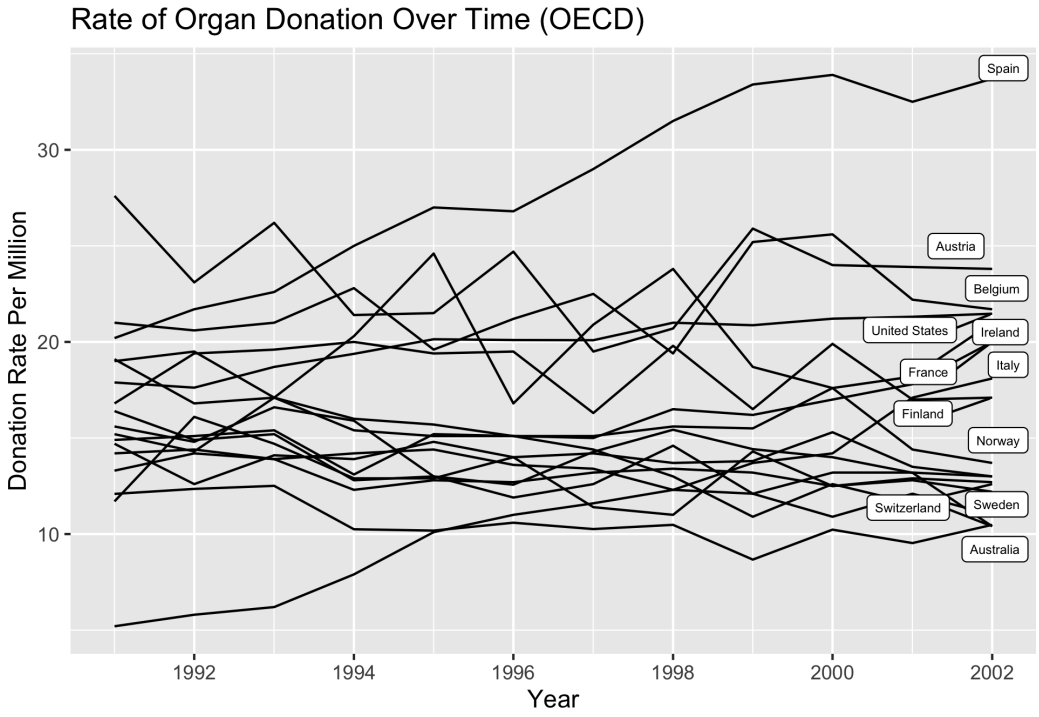
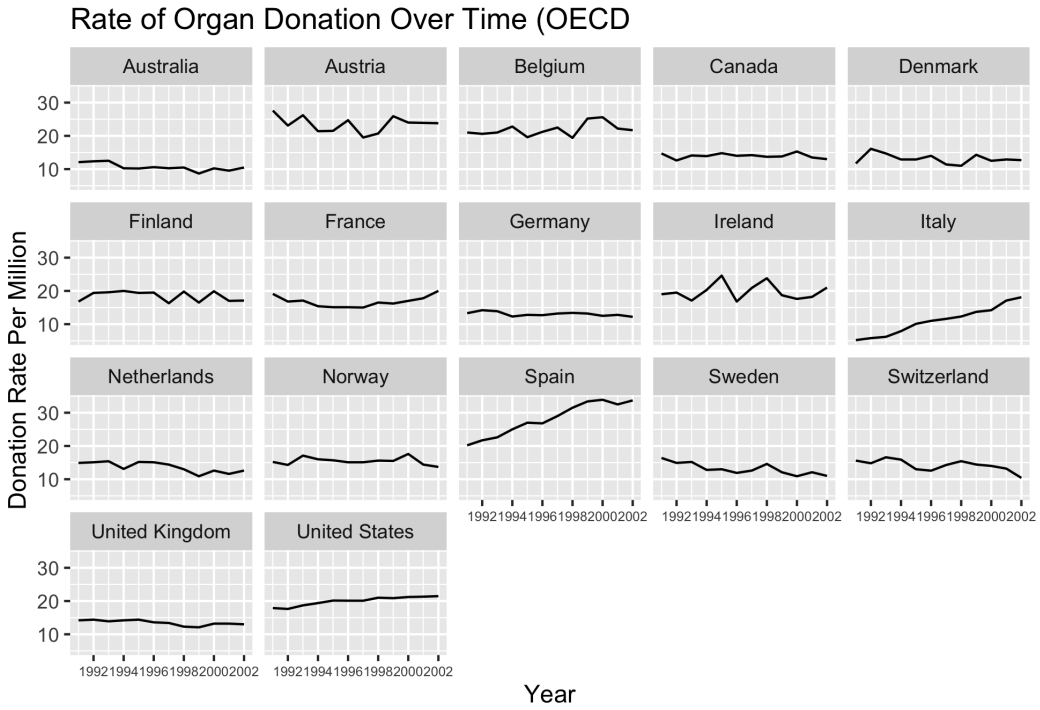


Figure 2: Properly labeled axes and a title help to make the figure more interpretable immediately to the reader

That looks more like a real plot but we might want to actually take a look at the individual countries to see what is going on in each of them and see if there are any particulalry surprising changes over time. As the figure stands above, that is somewhat obscured. We could do that by breaking up the data using filters and creating one plot per country but that is tedious. Luckily, `ggplot` has another function that allows you to created a *paneled* figure, so that a single plot is made for each country. We do this by using the `facet_wrap` function.

```
ggplot(data = organdata, aes(x = year, y = donors)) +
    geom_line() +
  facet_wrap(~country) +
  labs(y = "Donation Rate Per Million",
       x = "Year",
       title = "Rate of Organ Donation Over Time (OECD") +
  theme(axis.text.x = element_text(size = 6))
```

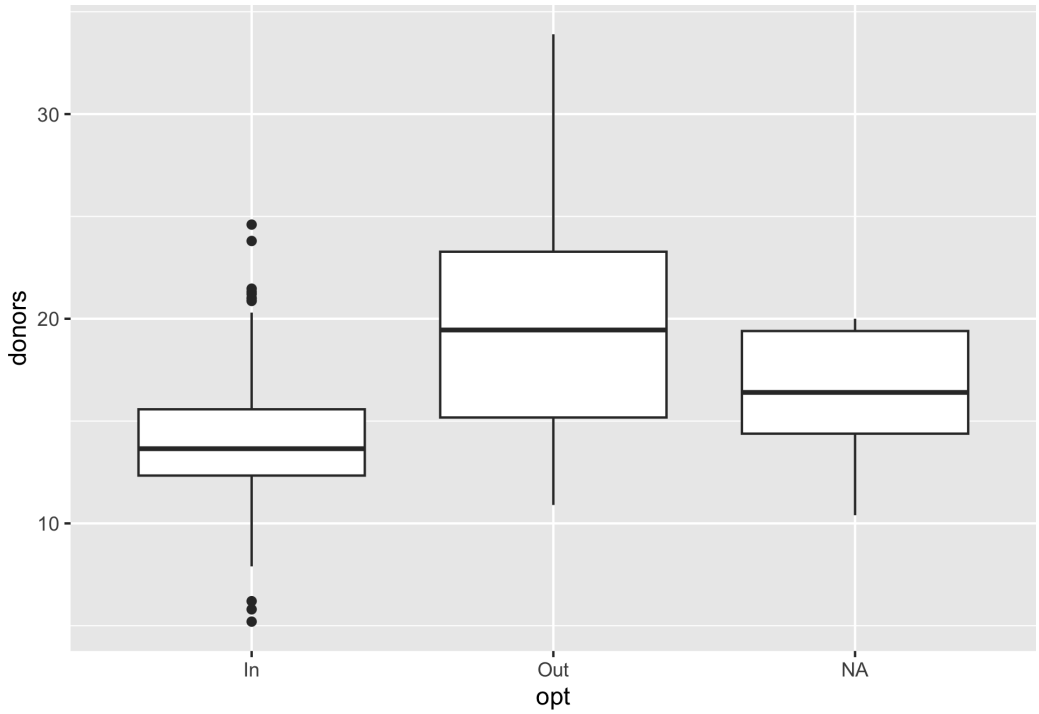
NOTE: There is a tilde before the variable we are grouping the data by for the panelling. That essentially tells the function to ”group by” that variable and plot the X and Y axis.



Now that helps to see hwo organ donation rates vary over time by country with a share range of X and Y values so each individual plot is comparable. What more, each plot is labled with the country it is displaying so each is clearly identifiable.

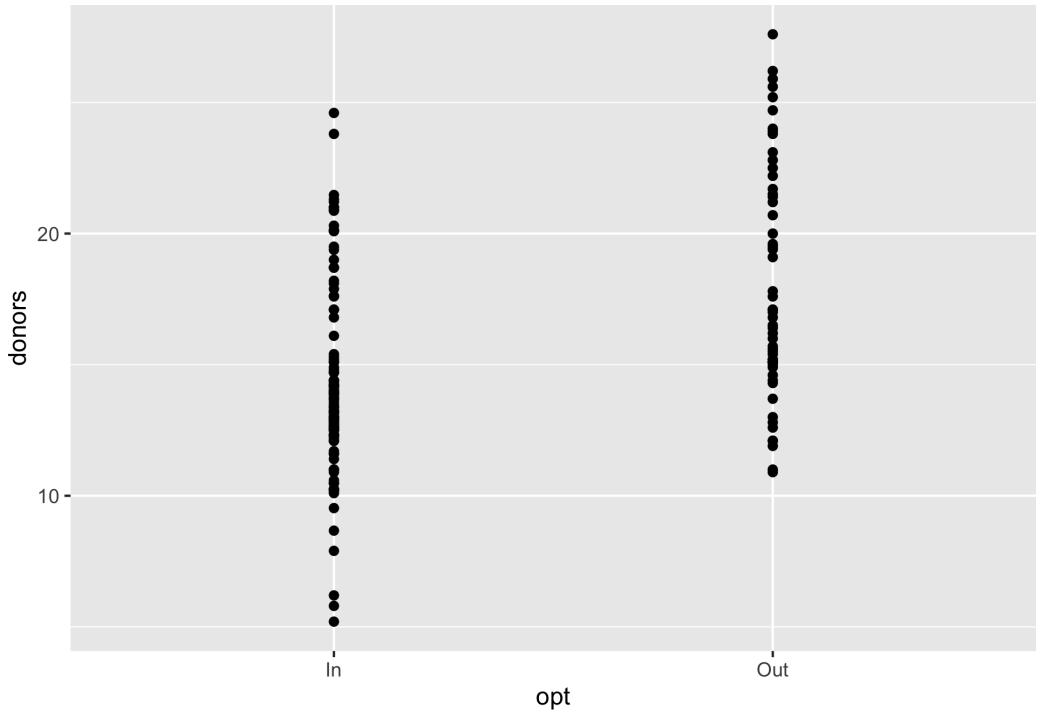
Variation by Group

Another thing we will want to check out is the variation in the focal variable, the rate of organ donation in each country, by some categorical difference in those countries. One that immmediately comes to mind is whether organ donation programs are something citizens opt in to or out of. My own informal hypothesis would be that there would be higher rates of organ donation in countries that include everyone in organ donation and require that they opt out rather than voluntarily opt in. To do this we would want to check out the mean and variation of organ donation across that category. We can use a "box and whisker plot" to plot this. This kind of plot displays a few importantn statistics: the median, lower and upper quartile (the box), and the minimum and maximum (the whiskers). The `ggplot` does something slightly differnet in that it shows outliers as individual points so it isn't strictly speaking using the minimum and maximum to the whiskers. We use the `geom_boxplot` function to do this after specifying the categorical variable in the X axis to plot the two groups and the same Y axis mapping as the plots above.

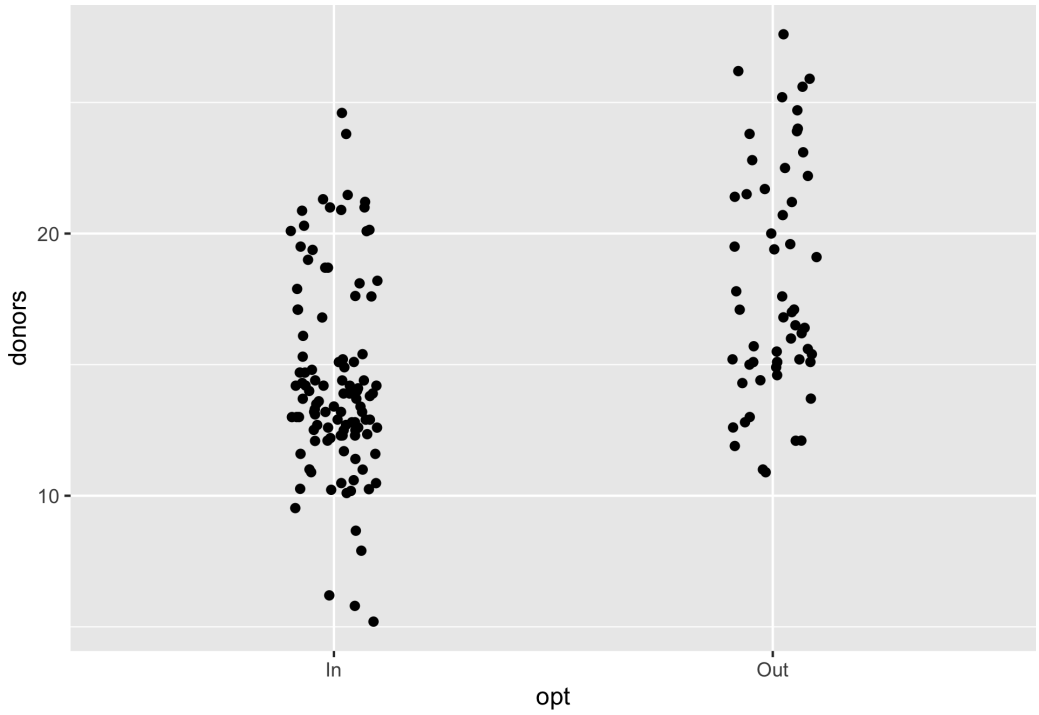


A few things should jump out to you: there does seem to be a difference between the two focal categories (opting in v. opting out) but the amount of variance in the two is different. Whatmore there is a third inexplicable category. Apparently not every country has data on whether they have an opt in or out policy and so there is missing data. It is also important to note that we are pooling all the data across all the years observations occurred. Whether or not that is a defensible visualization strategy will be a matter of debate.

A principle that we are going to cover is that making a good figure helps you visualize where your data *is* (Martin 2018). The box and whisker plot obscures where each data point actually is. So let's do two things: drop the missing data, and use a different geom that is going to let us see where each data point is. To do that we use a function that drops missing data (`na.omit()`) and a different `geom`. This time we use `geom_scatter` to make a scatter plot.



A scatter plot lets you see every data point but the categorical variable makes all the data points line up and we can't see where every data point is. Considering that the X axis is a categorical (really a nominal) level variable and the specific position along that axis is not really meaningful we can "jitter" the points to let us see where they are along the Y axis. This is another geom that we can add to the plot which will randomly scramble the position along the X axis between a specific range.



This looks much better and we can see some interesting things between the two countries. There does seem to be a difference between the two groups: the countries that require citizens to opt in to organ donation do tend to have a lower rate than those that require citizens to opt out, and they are much more concentrated around what is probably the mean. *But*, there are only a few observations below the probable mean of the Opt In group. These are what could be driving down the mean value and the difference between the groups.

I know that I am playing fast and loose here with the comparisons and that a conscientious socail scientist, like you, would do a few tests before this visualziation and after to interrogate the differences. But we are learning about figure creation so we want to make figures.

There is another variable in the data set that we can use to help interrogate the differences between these two groups. The data set includes a variable (`world`) that captures a typology outlined by Esping-Andersen in *The Three Worlds of Welfare Capitalism* which identifies three types of welfare state regimes by which advanced capitalist democracies can be categorized. These include: liberal, conservative, and social democratic. I'm not very familiar with what these specifically mean but we can add this to our plot to learn another feature of figure design. We can color the individual points by which category they fall in to. To do this we add another argument, `color`, to the aesthetic function in the call to `ggplot`.

There are some notable and interesting differences on the basis of both the opt in/out comparison and the different welfare state regimes. There are many more Corporatist regimes in the Opt Out category (particularly with high rates of organ donation) compared to the Social Democrats in the same group. Let's clean up the figure to make it more immediately interpretable for a possible reader. We are going to add proper axis lables, a title, legend title and change the plots *theme*. The theme styles the general appearance of the plot, and in this case we are going to change the plot backgroup from the standard `ggplot` grey to a clean white.

I once heard a professor deride the "ggplotification" of social scientific visualizations, and specifically the signature grey background.

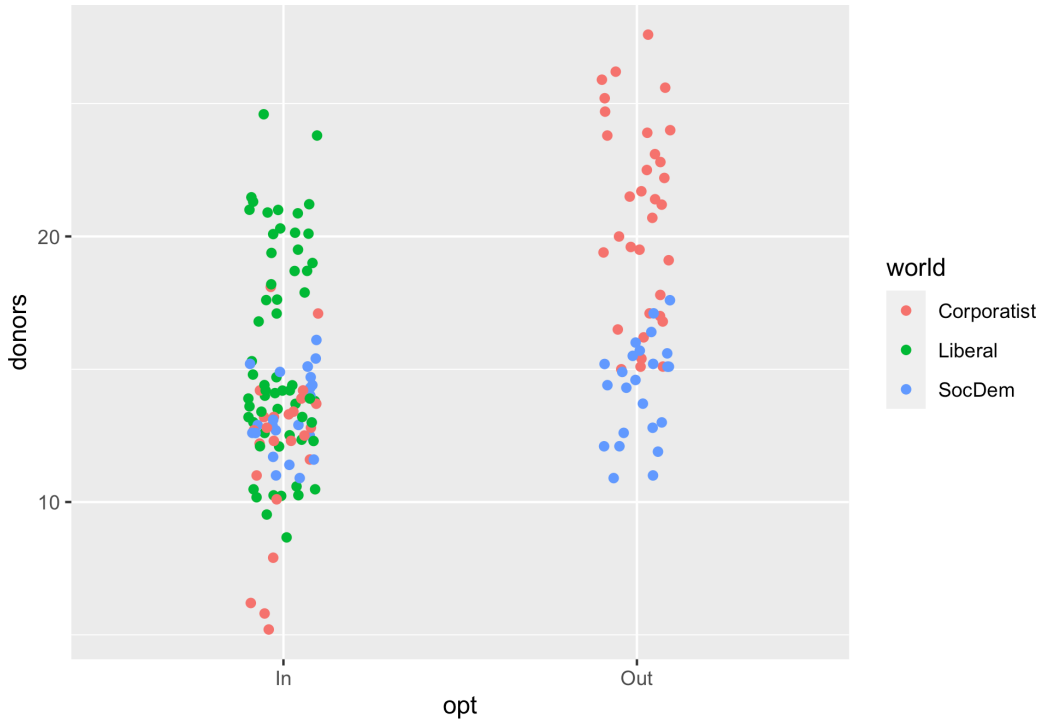
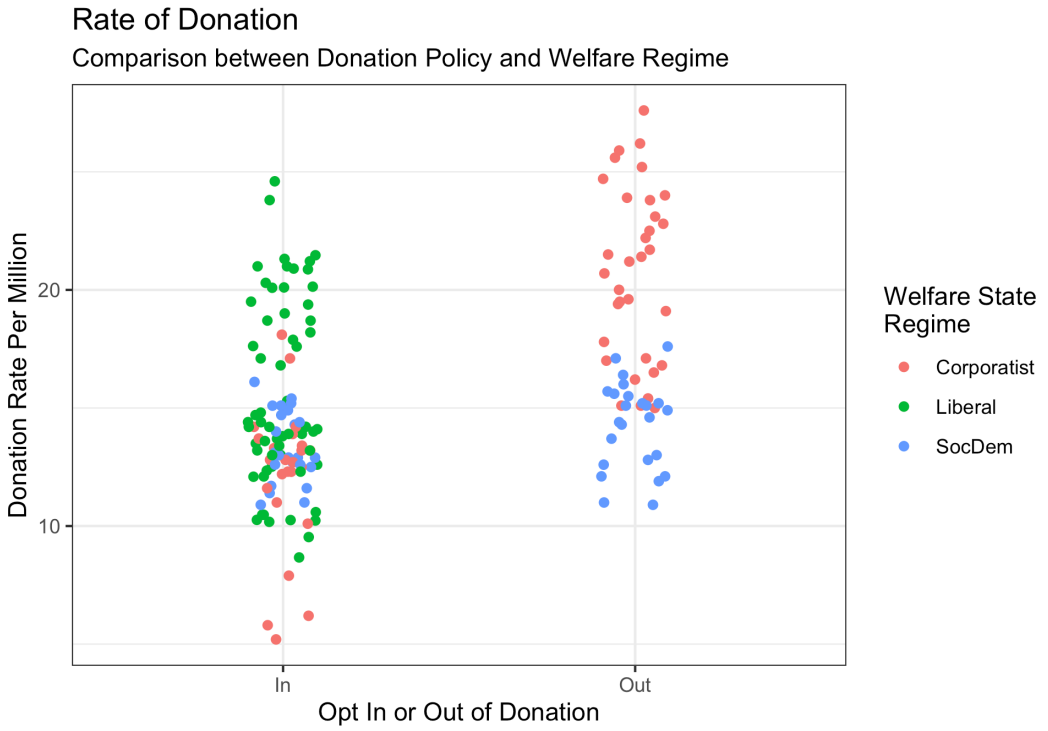
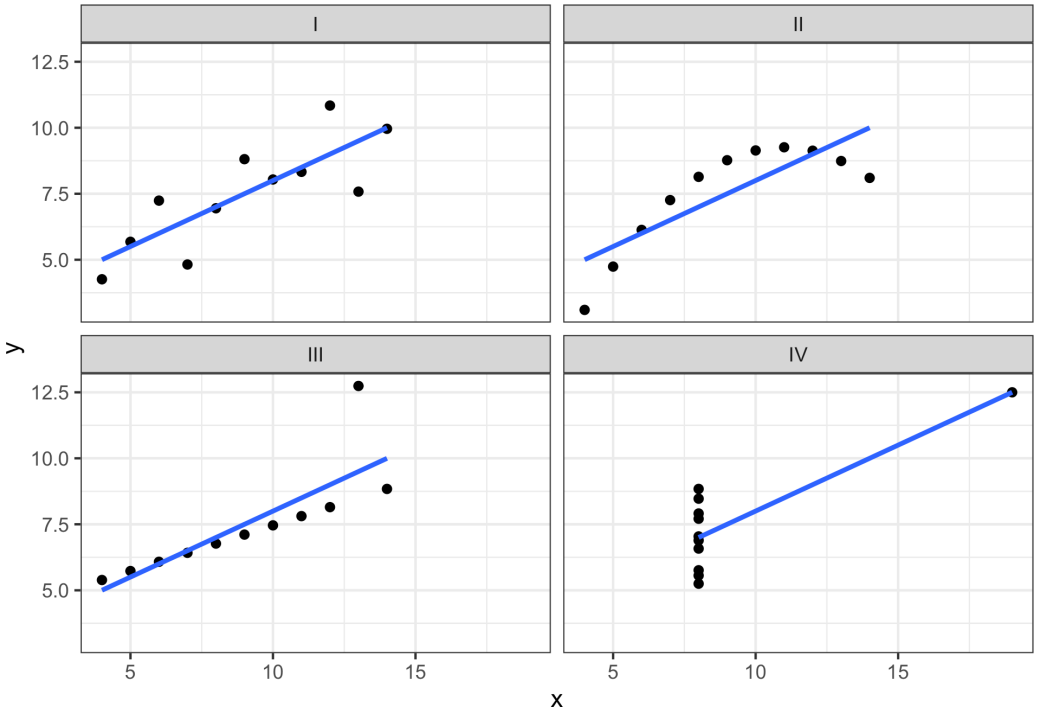


Figure 3: There are notable differences in terms of the two groups and the different categories of welfare state regimes.



Anscombe's Quartet

A key thing that we use visualizations for is understanding what our data can acutally say, and where it actually *is*. Anscombe's Quartet powerfully demonstrates the need to actually visualize your data so that you know where it is. The basic idea is that you can generate 4 different sets of data that from a statistical analysis will tell you that they are identical, **but** visualizing them shows that there are very different relationships going on.



And that is why you should visualize your data kids.

Comments for Code

There is a convention in computer science to include "comments" in your code, prose style notes that annotate your code and make it clearer what you did and why. You indicate a comment in R by including a pound or hashtag (#) at the beginning of the line and the same for Python. It is important to include comments in your code for two reasons:

1. Showing what you did: If you fully adopt an open science posture to your own research and writing you'll probably disseminate a curated dataset and the source files that generate all your findings, including the RMarkdown file of your actual article. Providing informative comments to your code will make it clearer what you did and why to whomever is really interested in understanding what you did.
2. Knowing what you did: The more important reason, or at least the reason that is going to be most useful to you, is that you need to know what *YOU* did. You are going to put the project down and go on vacation or work on other things and you'll need to get back and know what you were doing. Most of the time, really nearly all the time, you are the only one who needs to know what was happening in your document and so write comments knowing what you'll need to know in the future.

References

Durkheim, Émile. (1893) 2008. *The Division of Labor in Society*. 13. [Repr.]. New York: Free Press.

Martin, John Levi. 2018. *Thinking Through Statistics*. Chicago ; London: The University of Chicago Press.