Data Science: Visualisations in R - Assignment 1

dr. Mariken A.C.G. van der Velden

November 1, 2019

Instructions for the tutorial

Read the *entire* document that describes the assignment for this tutorial. Follow the indicated steps. Each assignment in the tutorial contains several excercises, divided into numbers and letters (1a, 1b, 1c, 2a, 2b, etc.).

Write down your answers with **the same number-letter combination** in a separate document (e.g., Google Doc), and submit this document to Canvas in a PDF format.

Note: this is the first time that this course has ever been taught. Because of rapidly changing techniques, the course tries to keep up/ reflect those changes. As a result, the assignments may contain errors or ambiguities. If in doubt, ask for clarification during the hall lectures or tutorials. All feedback on the assignments is greatly appreciated!

About these tutorials

Welcome to the first tutorial of the course Data Science: Visualisations in R. During six sessions you will be introduced to different ways of visualising data. The purpose of this is twofold. First of all you will be able to apply the techniques for your own research. For example, you can make 'pretty graphs' for your term papers. Secondly, and equally important, the aim of the course and the tutorials is to 'get your hands dirty' and prepare you for a field of work in which data visualizations play an increasingly important role.

The explosion of digital information and increasing efforts to digitise existing information sources has produced a deluge of data, such as digitised historical news archives, literature, policy and legal documents, political debates and millions of social media messages by politicians, journalists, and citizens. Visualizing the collected information allows you to explore and to learn about its structure. Good data visualisations enable you to communicate your ideas and findings. This is not only a valuable tool for scientists. Also outside of academia, there is clearly more demand for so-called data science skills. The aim of this course is to teach you this basic knowledge.

There are also some important topics that these tutorials will not cover. During the tutorials, we will mainly focus on small-scale in-memory datasets. This is the right place to start, because you cannot tackle big data questions unless you gain first expierence with some small data. Moreover, we will also not cover anything about *Python*, *Julia*, or any other programming languages for data science. We think that *R* is a good way to start your story into the data science visualization journey. Along the way, e.g. in your master, you have more opportunities to explore other programming languages to visualize (big) data.

Installing R and R Studio

During these tutorials, we will work with the software R. The overwhelming majority of the R community nowadays works with the interface *R Studio*. This is an integrated development environment, or IDE, for R.

To download R, go to CRAN, the *comprehensive R archive network*. CRAN is composed of a set of mirror servers distributed around the world and is used to distribute R and R packages. Please use the cloud mirror: *https://cloud.r-project.org*, which automatically figures out the mirror closest to you. Note: A new version of R comes out once a year, and there are 2-3 minor releases each year. It's a good idea to update regularly, despite the hassle that comes with updating.

Once R is installed, install R Studio from *http://www.rstudio.com/download*.

When you start R Studio, you will see four regions in the interface: 1) the code editor; 2) the R console; 3) the workspace and history; and 4) plots, files, packages and help. See e.g.p.36 of the book.

You will also need to install some packages. An R *package* is a collection of functions, data and documentation that extends the capabilities of base R. Using packages is key to the successfull use of R. The majority of packages we will work with in this course aer paert of the so-called tidyverse. The packages in the tidyverse share a common philosophy of data and R programming, and are designed to work together naturally.

You can install the complete tidyverse with a single line of code:

```
#install.packages("tidyverse") ## uncomment the line by deleting the '#'
```

You can type the line of code either in R Studio's console or code editor. For the former, you just have to press enter, for the latter you press Shift/enter (Windows) or command/enter (Mac). R will download the packages from CRAN and install them onto your computer.

You will not be able to use the functions, objects, and help files in a package until you load it with `library()`. Once you have installed a package, you can load it with the `library()` function:

```
library(tidyverse)
```

This tells you that tidyverse is loading the **gpglot2**, **tibble**, **tidyr**, **readr**, **purrr**, **dplyr**, **stringr**, and **forcats** packeges. These form the core of the tidyverse. Packages in the tidyverse cchange fairly frequently. You can see if updates are available, and optionally install them, by running `tidyverse_update()`.

When you will need other packages during the tutorials, this will always be stated explicitly in the assignment.

Running R Code

In the R console of R Studio, every line of code generally contains a command or instruction for the computer to do or calculate something (formally, such commands are often called statements). Like you learned in Python during the *Computational Thinking* course. In its simplest form, you can ask R to do simple sums, such as `2+2`:

```
2+2
```

```
## [1] 4
```

(note that the line `## [1] 4` is the output of this command: a single value 4)

Throughout the tutorials, we use a consistent set of conventions to refer to code:

- Functions are in a code font and followed by parentheses, like `sum()` or `mean()`. *Note*: If you like to explore a function use `?` in front of that function, e.g. `?sum`, and the explanation appears in the lower right panel of your R Studio.
- Other R objects (like data or function arguments) are in code font, without parentheses, like `x`.
- If we want to make clear what package an object comes from, we will use the package name followed by two colons, like `dplyr::mutate()` or `ggplot2:ggplot()`. This is also valid R code.

Data Visualization with ggplot2

During the course, we will learn you to visualize data using **ggplot2**. R has several systems for making graphs, but **ggplot2** is one of the most elegant and most versatile. **ggplot2** implements the *grammar of graphics*, a coherent systems for describing and building graphs. If you'd like to learn more about the theoretical underpinnings of **ggplot2**, I would recommend reading "A Layered Grammar of Graphics" (*http://vita.had.co.nz/papers/layered-grammar.pdf*)

The **layered grammar of graphics** tells us that a statistical graphic is a **mapping from data to aesthetic attributes** (colour, shape, size) **of geometric objects** (points, lines bars). In a formula, this looks like the following:

`Data + Aesthetic mappings + Layers` (Geometric objects, Stats) `+ Scales + Coordinate system + Faceting + Theme`

Key Components of that formula are: * **Data** = a data frame * A set of **aesthetic mappings** between variables in the data and visual properties (e.g. horizontal and vertical position, size, color and shape) * And at least one layer (describes how to render the observations; usually created with a **geom** function, e.g. `geom_point()`

Assignment 1a: Try a simple scatter plot

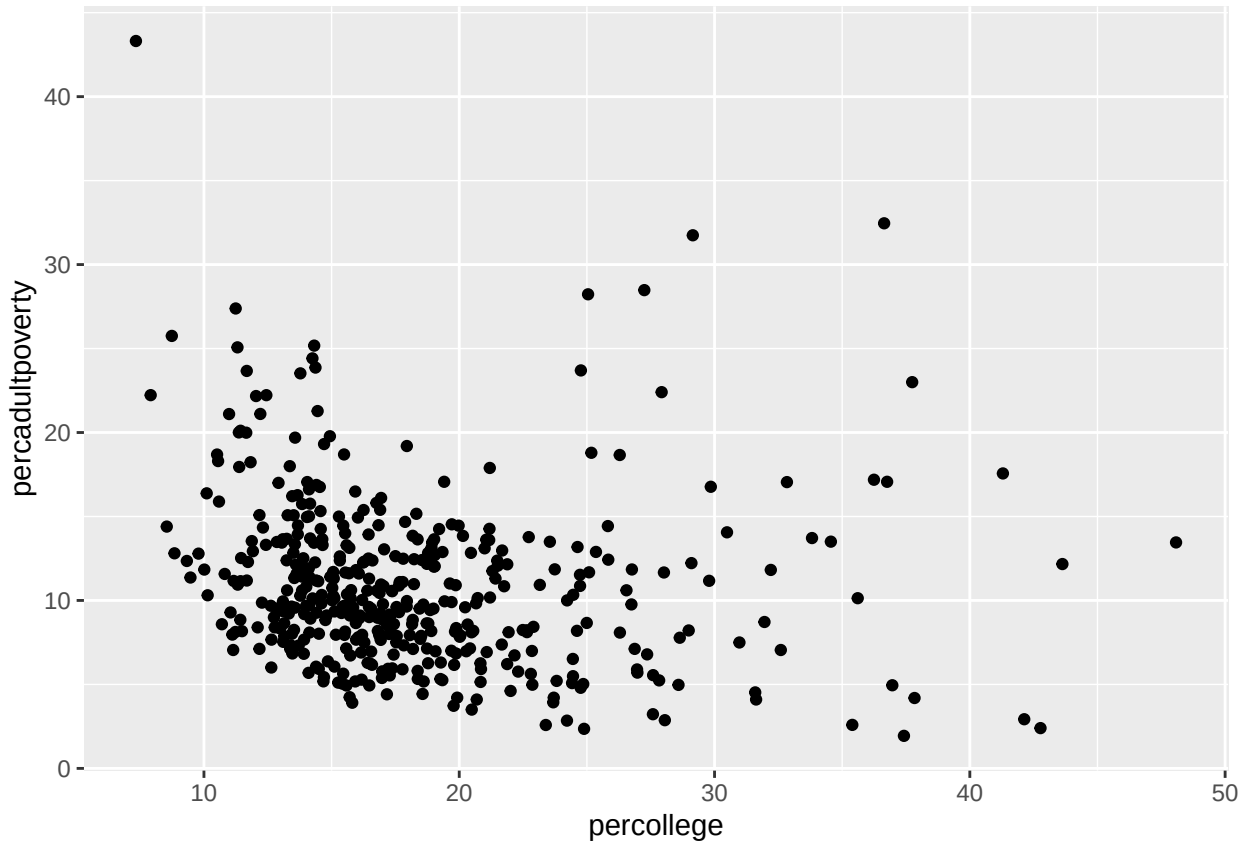
In this assignment, we will practice to get a little familiar with the above described key components of the grammar of graphics.

If you'd like to know whether areas in which more people with a college degree live are more likely to have low levels of poverty, we can use the `midwest` data to make a simple scatter plot, using the following key components:

- **Data** = `midwest`, you can inspect the data with either `head(midwest)` or `str(midwest)`
- **Aesthetic mappings** = percent college educated (variable `percollege` in `midwest`) mapped to x position, percent adult poverty (variable `percadultpoverty` in `midwest`) mapped to y position
- **Layer** = `geom_points()`

This leads to the following code:

```
ggplot(midwest, aes(x = percollege, y = percadultpoverty)) +
geom_point()
```




```
## There are two other ways to write the code for the same outcome.
#ggplot(midwest) +
#geom_point(aes(x = percollege, y = percadultpoverty))

#ggplot() +
#geom_point(data = midwest, aes(x = percollege, y = percadultpoverty))
```

1a) Replicate the codes and check whether you get three times the same graph. Pay attention, when `ggplot()` *is empty, you need to specify which data you use, using the* `data =` *statement in the layer* `geom_point()`.

1b) Explore the options of `geom_point()` *by adding a color to the points based on the values of* `percadultpoverty`. *Add only the used code into your assignment.*

1c) While the different colors for percent adult poverty (variable `percadultpoverty`*) give us un overview of the severity of adult poverty in the entire region the Midwest in the US. We might also want to know whether some states are worse of than others. Explore the options of* `geom_point()` *by adding a color to the points based on the values of* `state`. *Add only the used code into your assignment.*

1d) To combine both the variation in state and severity of adult povert, add a color for `percadultpoverty` *to the figure and shapes for the different states. Add the code to your assignment.*

1e) The figure created in 1e might become hard to read. Therefore, instead of using shapes for states, you can use `facet_grid` *to split the graph on state. Explore this command. You can also try out different fixed colors for* `percadultpoverty` *looking at http://www.stat.columbia.edu/~tzheng/files/Rcolor.pdf. Pay attention, when you want to fix a color, it needs to be outside the* `aes` *statement and in between "...". Add the code and the figure to your assignment. You can save the plot either by clicking on the "Export" button in the lower right panel or use* `ggsave()`.

Layers

There are many different layers one could choose from to visualize data. Next to a visual display, layers serve the purpose of giving a statistical summary of the data at hand. Furthermore, layers could be used to add additional metadata (e.g. annotations, maps, etc.). The geoms in `ggplot2` define the shape of the elements on the plot. The basic shapes one can choose from are points, lines, polygons, bars, texts; i.e. `geom_point()`, `geom_line()`, `geom_polygon()`, `geom_bar()`, and `geom_text()`. For statistical summaries of the data, one could use `geom_histogram()`, `geom_smooth()`, or `geom_density()`. For today's tutorial, we will work with the basic layers. Next week, we will dive into the statistical summary layers.

Assignment 2: Apply different basic layers

2a) create a data set using the following code

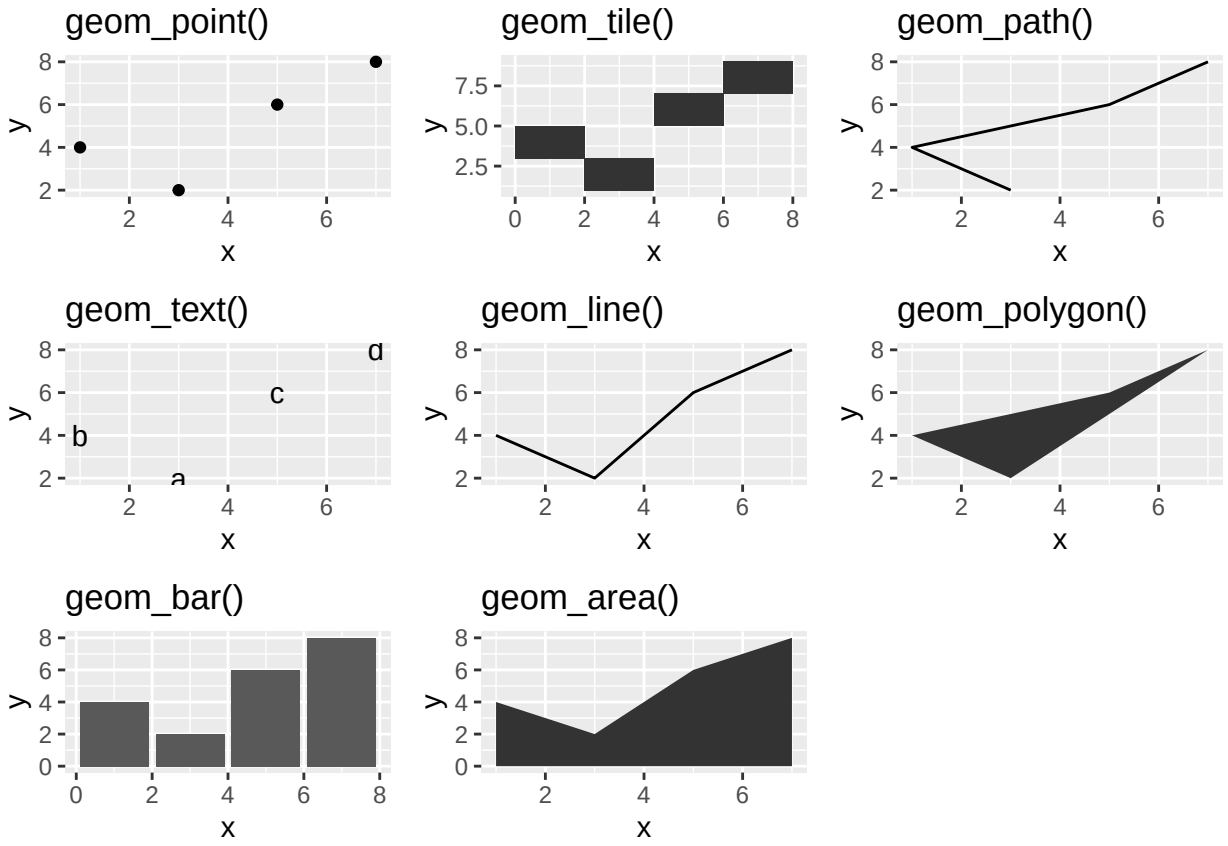
```
my_first_data_frame = data.frame(x = c(3,1,5,7),
                                 y = c(2,4,6,8),
                                 label = c("a", "b", "c", "d"))

my_first_data_frame
```

```
##   x y label
## 1 3 2     a
## 2 1 4     b
## 3 5 6     c
## 4 7 8     d
```

Do note that in R, in order to store an object, you always have to name it. The name itself doesn't matter, you can pick any name!

2b) The figure below uses `my_first_data_frame` *to display different layers. Replicate each figure. To add a title, use* `ggtitle()`*.To have all the graphs combined, have a look at http://www.cookbook-r.com/Graphs/Multiple_graphs_on_one_page_(ggplot2)/. Copy the code and add the combined graphs into your assignment.*

For a little help download the Data Visualization Cheat Sheet here: https://www.rstudio.com/wp-content/uploads/2016/11/ggplot2-cheatsheet-2.1.pdf



2c) Think of types of data for which each plot would be a good visualization.

Good Luck

The deadline for the assignment is **November 7, 10am**!

References

Healy, K. (2018). *Data visualization: a practical introduction.* Princeton University Press.

Wickham, H., & Grolemund, G. (2016). *R for data science: import, tidy, transform, visualize, and model data.* " O'Reilly Media, Inc."

Data Science: Visualisations in R - Assignment 2

Healy (2018) Data visualization, Chapters 3 and 4

dr. Mariken A.C.G. van der Velden

November 8, 2019

Instructions for the tutorial

Read the *entire* document that describes the assignment for this tutorial. Follow the indicated steps. Each assignment in the tutorial contains several excercises, divided into numbers and letters (1a, 1b, 1c, 2a, 2b, etc.).

Write down your answers with **the same number-letter combination** in a separate document (e.g., Google Doc), and submit this document to Canvas in a PDF format.

Note: this is the first time that this course has ever been taught. Because of rapidly changing techniques, the course tries to keep up/ reflect those changes. As a result, the assignments may contain errors or ambiguities. If in doubt, ask for clarification during the hall lectures or tutorials. All feedback on the assignments is greatly appreciated!

Mapping of variables

Last week, we have started to get to know `ggplot2`'s grammar of graphics. There is some structured relationship, some *mapping*, between the variables in your data and their representation in the plot displayed. Not all mappings make sense for all types of variables, and (independently of this) some representations are harder to interpret than others.

`ggplot2` provides you with a set of tools to map data to visual elements on your plot, to specify the kind of plot you want, and then to control the fine details of how it will be displayed. Figure 3.1 (p. 56 in the book) demonstrates the workflow of `ggplot2`.

The most important thing to get used to with `ggplot2`, and we played a little with this in last week's tutorial, is the way you use it to think about the logical structure of your plot. The code you write - and wrote - specifies the connections between your data and the plot elements, and the colors, points, and shapes you see on the screen. In `ggplot2`, these logical connections between your data and the plot elements are called *aesthetic mappings* or just *aesthetics*.

You begin every plot by telling the `ggplot()` function what your data is and how the variables in this data logically map onto the plot's aesthetics. Once you've specified the aesthetics, you specify the layers with one of the `geom_` functions.

Geoms: an overview

To represent distributions for:

- Continuous variables: `geom_dotplot()`, `geom_histogram()`, `geom_freqpoly()`, `geom_density()`
- Categorical variables: `geom_bar()`, `geom_text()` (also: summary of continuous variable)

To represent relationships:

- Between a continuous and a factor variable: `geom_boxplot()`, `geom_jitter()`, `geom_violin()`, `geom_point()`
- Between a summarized continuous variable and a factor variable: `geom_bar(stat="identity")`

- Between two continuous variables: `geom_point()`, `geom_smooth(method = c("loess", "gam", "lm", "rlm"))`

To represent time series and areas:

- `geom_lines()`, `geom_path()`, `geom_area()`, `geom_polygon()`, `geom_tile()`

Reshaping data to visualize

Sometimes data frames are stored into a so-called *long-format*, other times they are stored in a *wide-format*. In a long-format table, every variable is a column and every row is an observation. In a wide-format table, some variables are spread out over multiple columns. For example, different years, as displayed in Table 3.1 (p. 57 of the book). We can remind ourselves with what type of data we're dealing by typing `head(data)` in the console to display the first 6 rows by default. For example:

```
library(tidyverse)# don't forget to call the tidyverse package from the library again,
#you don't have to install it!

#install.packages("gapminder") # uncomment this if you have not installed the package yet.
library(gapminder)

head(gapminder)
```

```
## # A tibble: 6 x 6
##   country     continent  year lifeExp      pop gdpPercap
##   <fct>       <fct>     <int>   <dbl>    <int>     <dbl>
## 1 Afghanistan Asia       1952    28.8  8425333      779.
## 2 Afghanistan Asia       1957    30.3  9240934      821.
## 3 Afghanistan Asia       1962    32.0 10267083      853.
## 4 Afghanistan Asia       1967    34.0 11537966      836.
## 5 Afghanistan Asia       1972    36.1 13079460      740.
## 6 Afghanistan Asia       1977    38.4 14880372      786.
```

Steps to build a plot

Let's say we want to plot life expectancy against per capita GDP for all country-years in the data. We'll do this by ccreating an object that has some of the necessary information in it and build it up from there. First, we have to tell the `ggplot()`function what data we are using:

```
p <- ggplot(data = gapminder)
```

Now, `ggplot()` knows the data, but not yet the *mapping*: we need to tell which variables in the data should be represented by which visual elements in the plot. In `gpglot2`, mappings are specified using the `aes()` function.

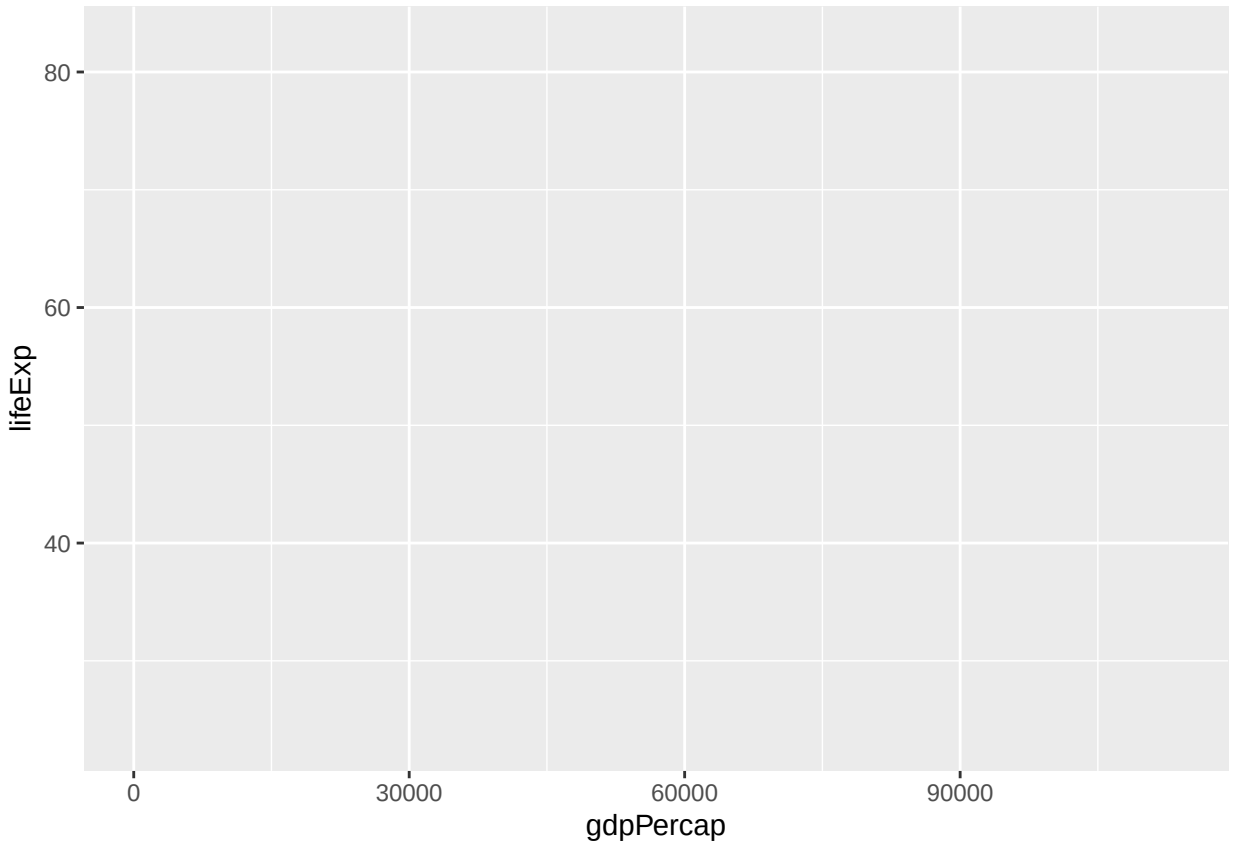
```
p <- ggplot(data = gapminder, mapping = aes(x = gdpPercap, y = lifeExp))
```

The code above gives `ggplot()` two arguments: 1) `data`; and 2) `mapping`. The latter thus *links variables* to *things you will see* on the plot. The `x` and `y` values are the most obvious aesthetics. Other aesthetic mappings include color, shape, size, and linetype.

It is important to note that a mapping does not directly say what particular colors or shapes, for example, will be on the plot. Rather, it says which *variables* in the data will be *represented* by visual elements.

So, what happens if we type in the object `p` in the R console?

```
p
```

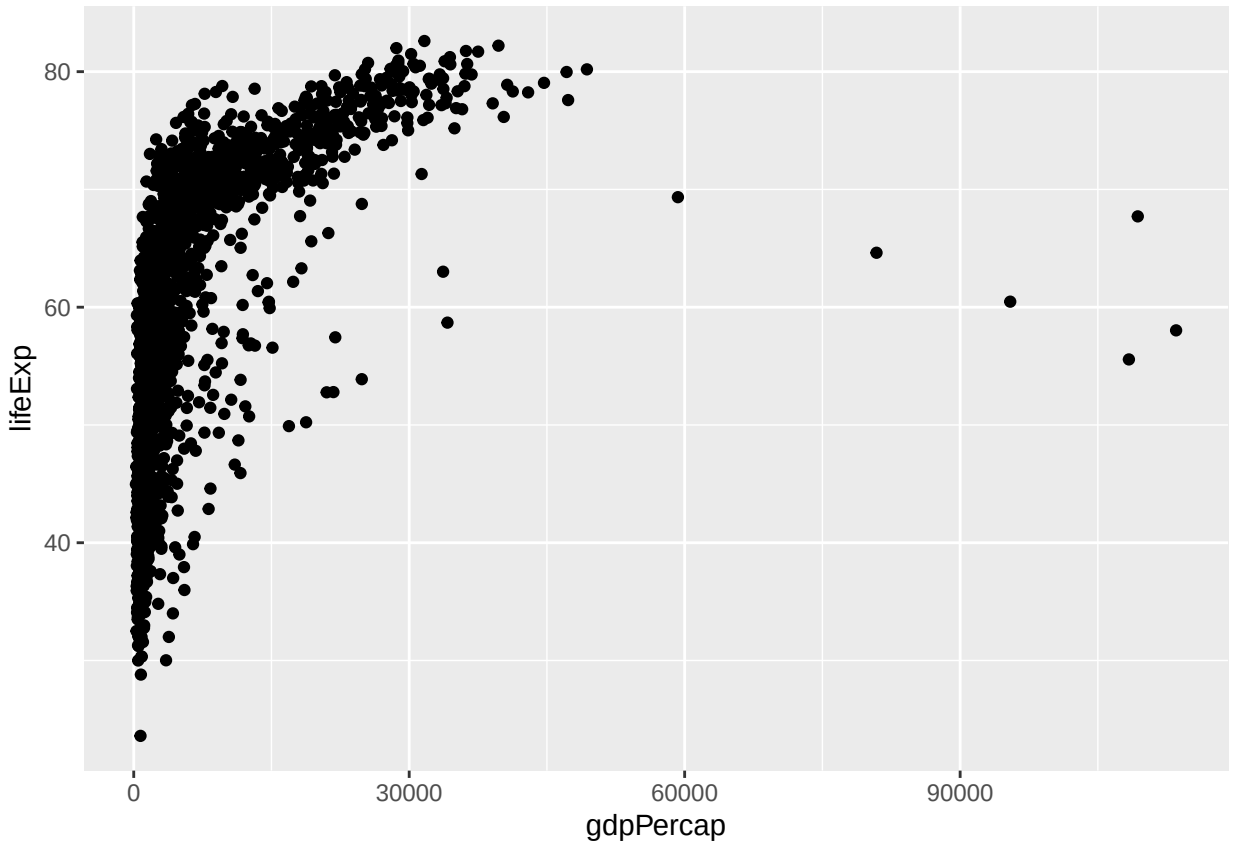



```
## the other way to immediately tell R to show the object is:
## (p <- ggplot(data = gapminder))
```

The `p` object has been created by the `ggplot()` function and has information in it about the mappings we want, together with a lot of other information added by default. If you want to see how much information, check `str(p)`. Yet, what is still not in the object `p` is any information on the layers - i.e. instructions as we have seen last week on what plot to draw.

Let's create a scatterplot:

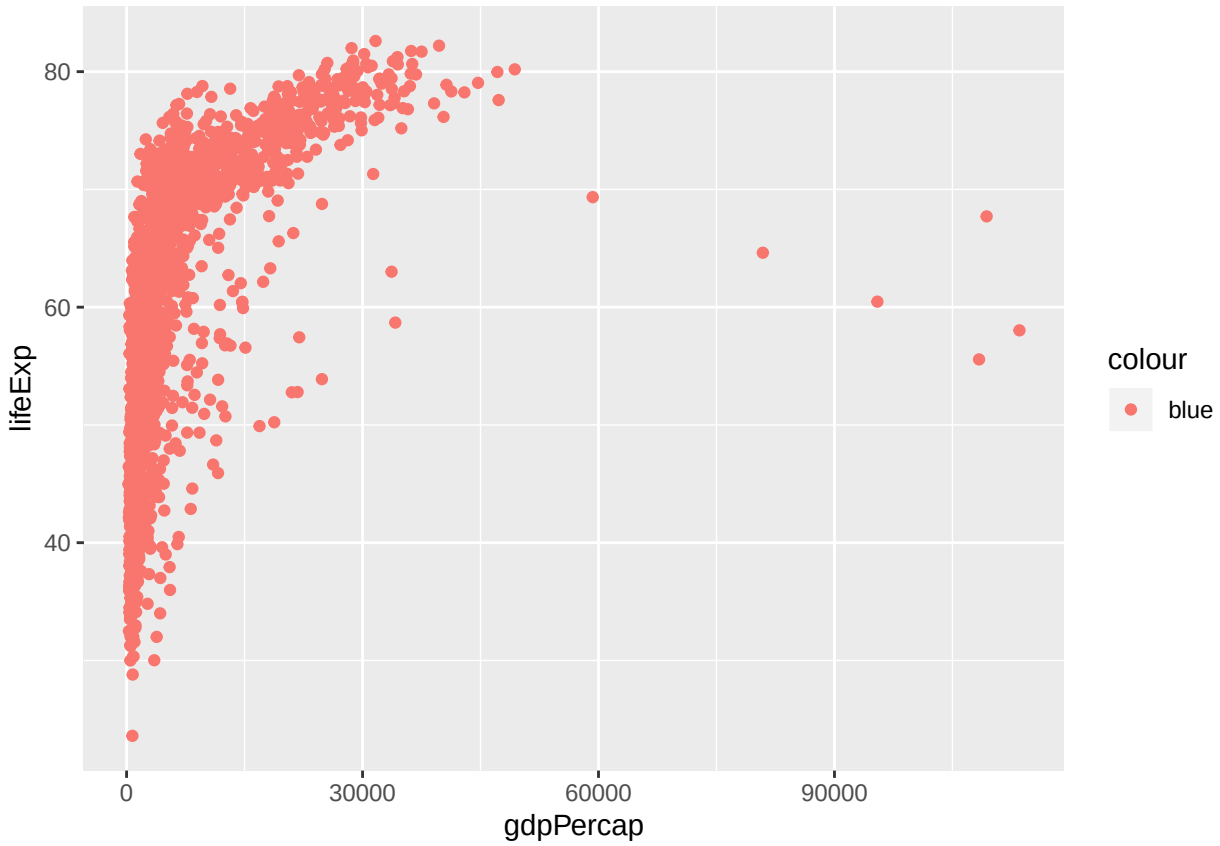
```
p + geom_point()
```



Assignment 1: Mappings & Layers

1a) In Assignment 1, you've practiced with adding colors to the graph. What is going wrong with the following code? Repair the code.

```
library(tidyverse)
ggplot(data = gapminder) +
  geom_point(mapping = aes(x = gdpPercap, y = lifeExp, color = "blue"))
```



1b) Use the same mappings as used in object `p` *but instead use a different* `geom`*. Add a* `geom_smooth` *as a layer.*

1c) What happens when you put the `geom_smooth()` *function before* `geom_point()` *instead of after it? What does this tell you about how the plot is drawn? Think about how this might be useful when drawing plots.*

1d) In the 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 a function called `scale_x_log10()`*. Add this function to the plot and describe what happens.*

1e) Try some alternative scale mappings. Besides `scale_x_log10()` *you can try* `scale_x_sqrt()` *and* `scale_x_reverse()`*. There are corresponding functions for y-axis transformations. Just write y instead of x. Experiment with them to see what sort of effect they have on the plot, and whether they make any sense to use.*

1f) What happens if you map `color` *to* `continent`*? And what happens if you map it to* `year`*?*

1g) Instead of mapping `color = year`*, what happens if you try* `color = factor(year)`*?*

To save plots, you can use `ggsave()` (See section 3.7 of the book).

Show the right numbers

When you write ggplot code in R you are in effect trying to "say"" something visually. It usually takes several iterations to say exactly what you mean. This is more than a metaphor here. The ggplot library is an implementation of the "grammar" of graphics, an idea developed by Wilkinson (2005) (https://www.springer.com/gp/book/9780387245447). The grammar is a set of rules for producing graphics from data, taking pieces of data and mapping them to geometric objects (like points and lines) that have aesthetic

attributes (like position, color and size), together with further rules for transforming the data if needed (e.g. to a smoothed line), adjusting scales (e.g. to a log scale) and projecting the results onto a different coordinate system (usually cartesian).

A key point is that, like other rules of syntax, the grammar limits the structure of what you can say, but it does not automatically make what you say sensible or meaningful. It allows you to produce long "sentences" that begin with mappings of data to visual elements and add clauses about what sort of plot it is, how the axes are scaled, and so on. But these sentences can easily be garbled. Sometimes your code will not produce a plot at all because of some syntax error in R. You will forget a `+` sign between `geom_` functions, or lose a parenthesis somewhere so that your function statement becomes unbalanced. In those cases R will complain (perhaps in an opaque way) that something has gone wrong. At other times, your code will successfully produce a plot, but it will not look the way you expected it to. Sometimes the results will look very weird indeed. In those cases, the chances are you have given ggplot a series of grammatically correct instructions that are either nonsensical in some way, or have accidentally twisted what you meant to say. These problems often arise when ggplot does not have quite all the information it needs in order make your graphic say what you want it to say.

Assignment 2: Which Numbers to Plot

2a) Imagine we wanted to plot the trajectory of life expectancy over time for each country in the data. We map `year` *to x and* `lifeExp` *to y. We take a quick look at the documentation and discover that* `geom_line()` *will draw lines by connecting observations in order of the variable on the x-axis, which seems right. Run the code below. What happened?*

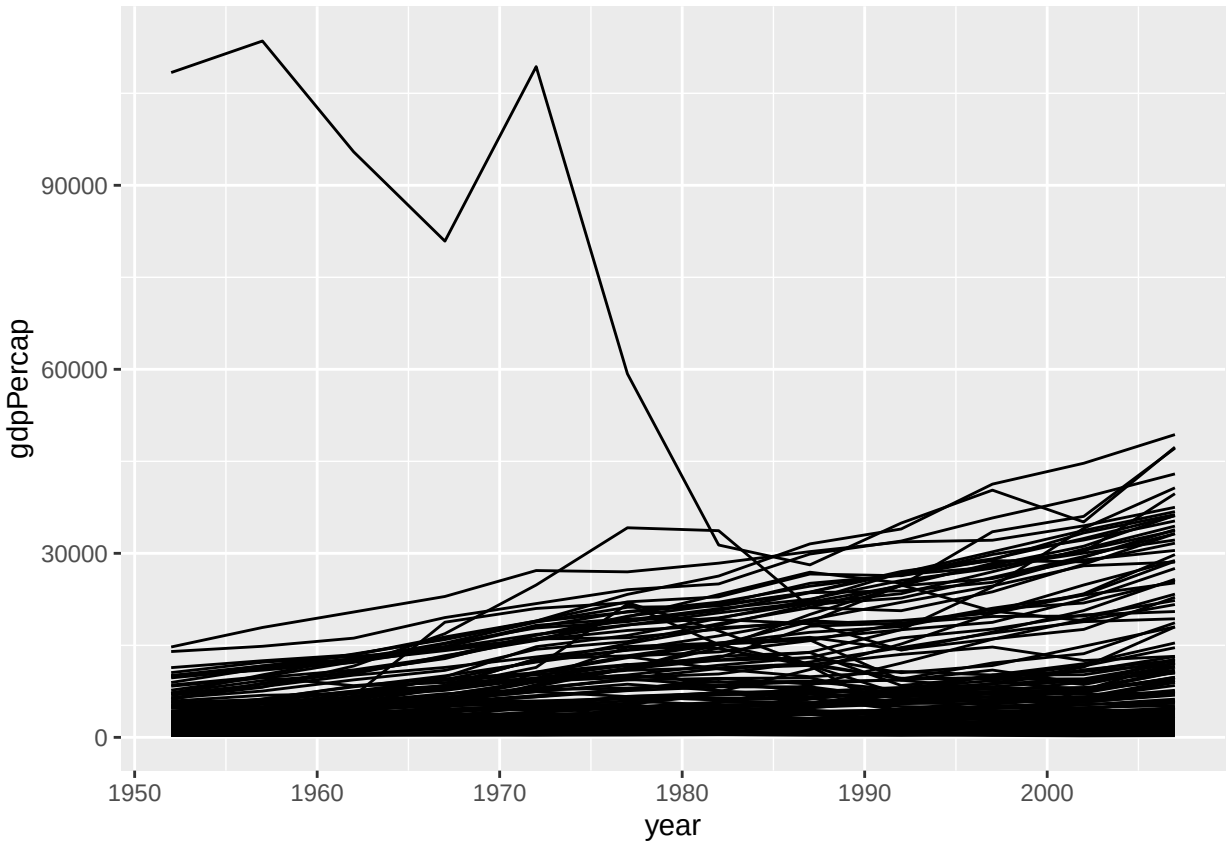
```
(p <- ggplot(data = gapminder,
             mapping = aes(x = year,
                           y = gdpPercap)) +
     geom_line())
```

While ggplot will make a pretty good guess as to the structure of the data, it does not know that the yearly observations in the data are grouped by country. We have to tell it. Because we have not, 'geom_line() gamely tries to join up all the lines for each particular year in the order they appear in the dataset, as promised. It starts with an observation for 1952 in the first row of the data. It doesn't know this belongs to Afghanistan. Instead of going to Afghanistan 1953, it finds there are a series of 1952 observations, so it joins all of those up first, alphabetically by country, all the way down to the 1952 observation that belongs to Zimbabwe. Then it moves to the first observation in the next year, 1957.This would have worked if there were only one country in the dataset.

The result is meaningless when plotted. Bizarre-looking output in ggplot is common enough, because everyone works out their plots one bit at a time, and making mistakes is just a feature of puzzling out how you want the plot to look. When ggplot successfully makes a plot but the result looks insane, the reason is almost always that something has gone wrong in the mapping between the data and aesthetics for the geom being used. This is so common there's even a Twitter account devoted to the "Accidental aRt" that results. So don't despair!

In this case, we can use the `group` aesthetic to tell ggplot explicitly about this country-level structure:

```
(p <- ggplot(data = gapminder,
             mapping = aes(x = year,
                           y = gdpPercap)) +
  geom_line(aes(group=country)))
```



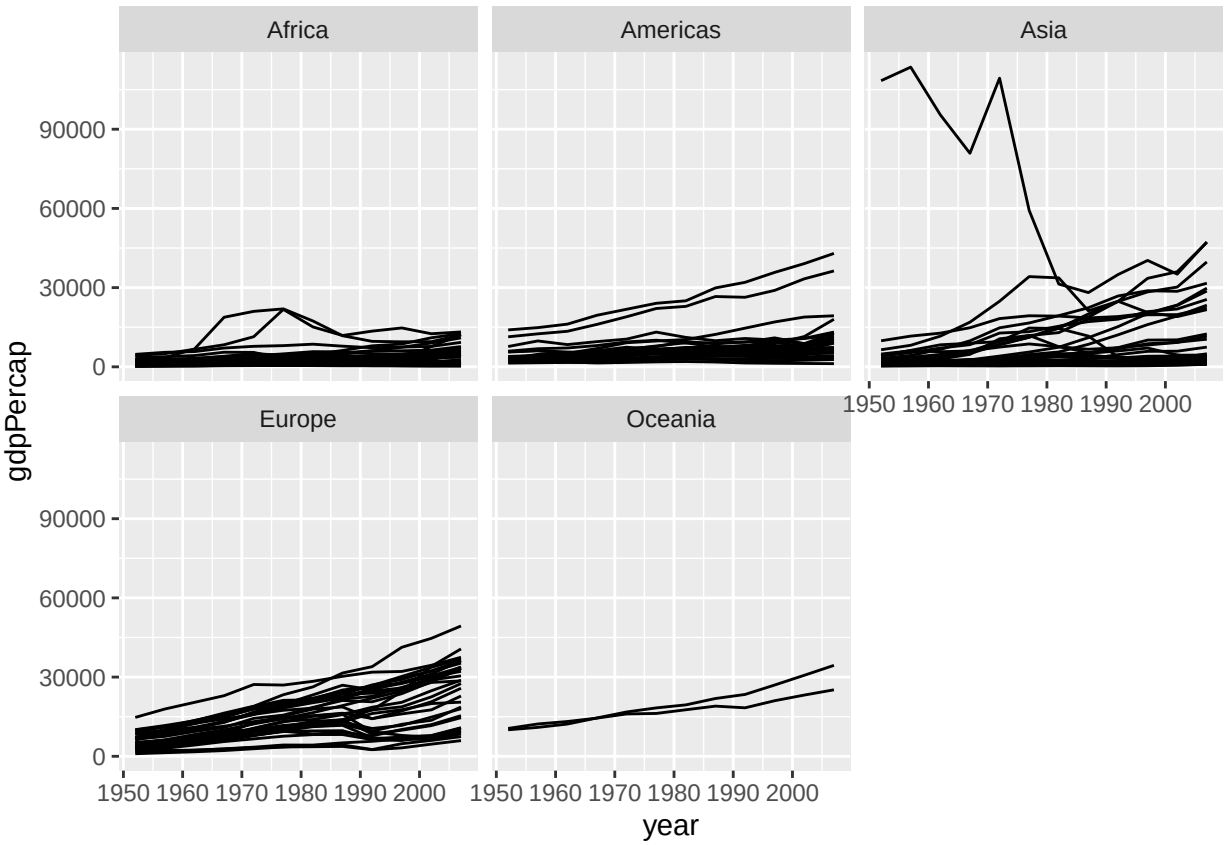
The plot here is still fairly rough, but it is showing the data properly, with each line representing the trajectory of a country over time. The gigantic outlier is Kuwait, in case you are interested.

The `group` aesthetic is usually only needed when the grouping information you need to tell ggplot about is not built-in to the variables being mapped. For example, when we were plotting the points by continent, mapping `color` to `continent` was enough to get the right answer, because `continent` is already a categorical variable, so the grouping is clear. When mapping the `x` to `year`, however, there is no information in the `year` variable itself to let ggplot know that it is grouped by country for the purposes of drawing lines with it. So we need to say that explicitly.

The plot we just made has a lot of lines on it. While the overall trend is more or less clear, it looks a little messy. One option is to *facet* the data by some third variable, making a "small multiple" plot. This is a very powerful technique that allows a lot of information to be presented compactly, and in a consistently comparable way. A separate panel is drawn for each value of the faceting variable. Facets are not a geom, but rather a way of organizing a series of geoms. In this case we have the `continent` variable available to us. We will use `facet_wrap()` to split our plot by `continent`.

The `facet_wrap()` function can take a series of arguments, but the most important is the first one, which is specified using R's "formula" syntax, which uses the tilde character, `~`. Facets are usually a one-sided formula. Most of the time you will just want a single variable on the right side of the formula. But faceting is powerful enough to accommodate what are in effect the graphical equivalent of multi-way contingency tables, if your data is complex enough to require that. For our first example, we will just use a single term in our formula, which is the variable we want the data broken up by: `facet_wrap(~ continent)`.

```
p <- ggplot(data = gapminder,
            mapping = aes(x = year,
                          y = gdpPercap))
p + geom_line(aes(group = country)) +
  facet_wrap(~ continent)
```



2b) Work with ggplot's `mpg` *data - to inspect the data set, type* `?mpg` *in the Console. What plots does the following code make? What does* `.` *do?*

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy)) +
  facet_grid(drv ~ .)

ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy)) +
  facet_grid(. ~ cyl)
```

2c) Read `?facet_wrap`*. What does* `nrow` *do? What does* `ncol` *do? What other options control the layout of the individual panels? Why doesn't* `facet_grid()` *have* `nrow` *and 'ncol arguments?*

2d) When using `facet_grid()` *you should usually put the variable with more unique levels in the columns. Why?*

Geoms Can Transform Data

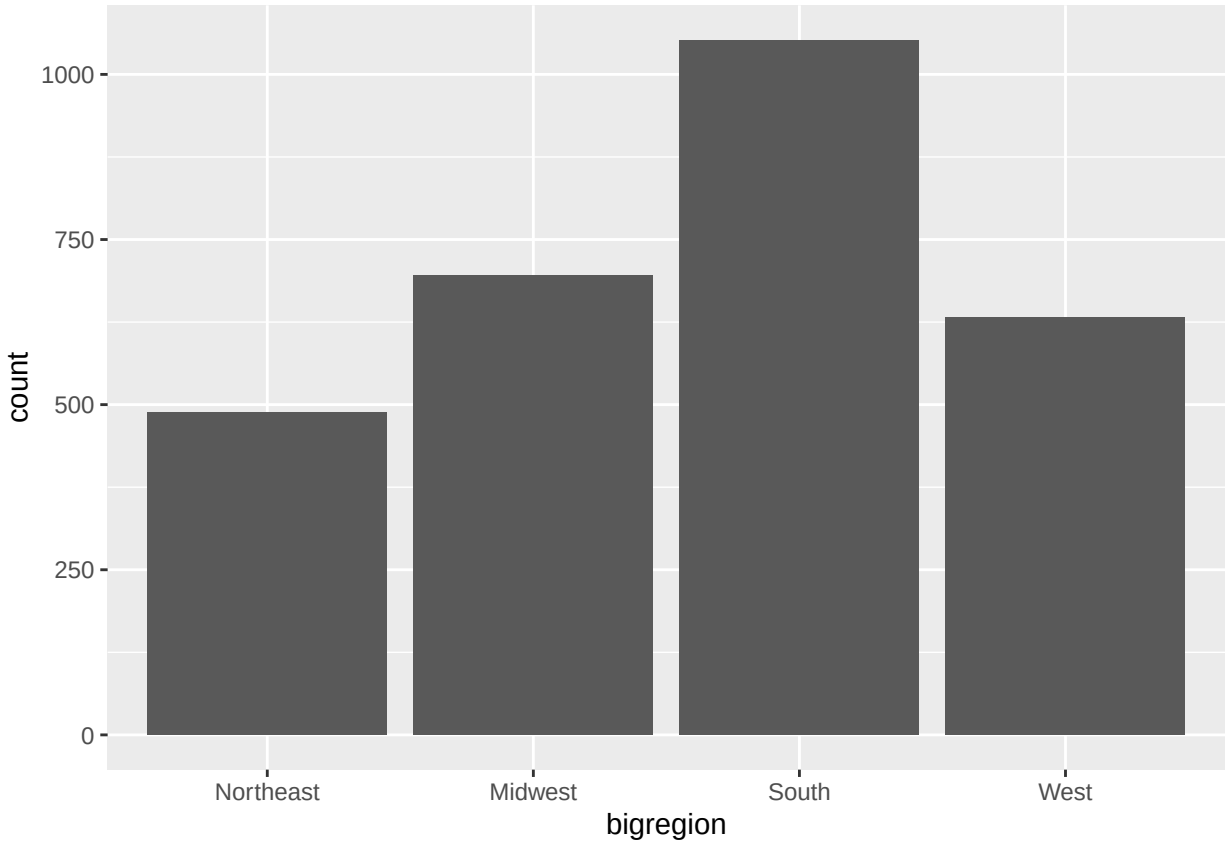
We have already seen several examples where `geom_smooth()` was included as a way to add a trend line to the figure. Sometimes we plotted a LOESS line, sometimes a straight line from an OLS regression, and sometimes the result of a Generalized Additive Model. We did not have to have any strong idea of the differences between these methods. Neither did we have to write any code to specify the underlying models, beyond telling the method argument in `geom_smooth()` which one we wanted to use. The `geom_smooth()` function did the rest.

Thus, some geoms plot our data directly on the figure, as is the case with `geom_point()`, which takes variables designated as x and y and plots the points on a grid. But other geoms clearly do more work on the data

before it gets plotted.Try `p + stat_smooth()`, for example. Every `geom_` function has an associated `stat_` function that it uses by default. The reverse is also the case: every `stat_` function has an associated `geom_` function that it will plot by default if you ask it to. This is not particularly important to know by itself, but as we will see in the next section, we sometimes want to calculate a different statistic for the geom from the default.

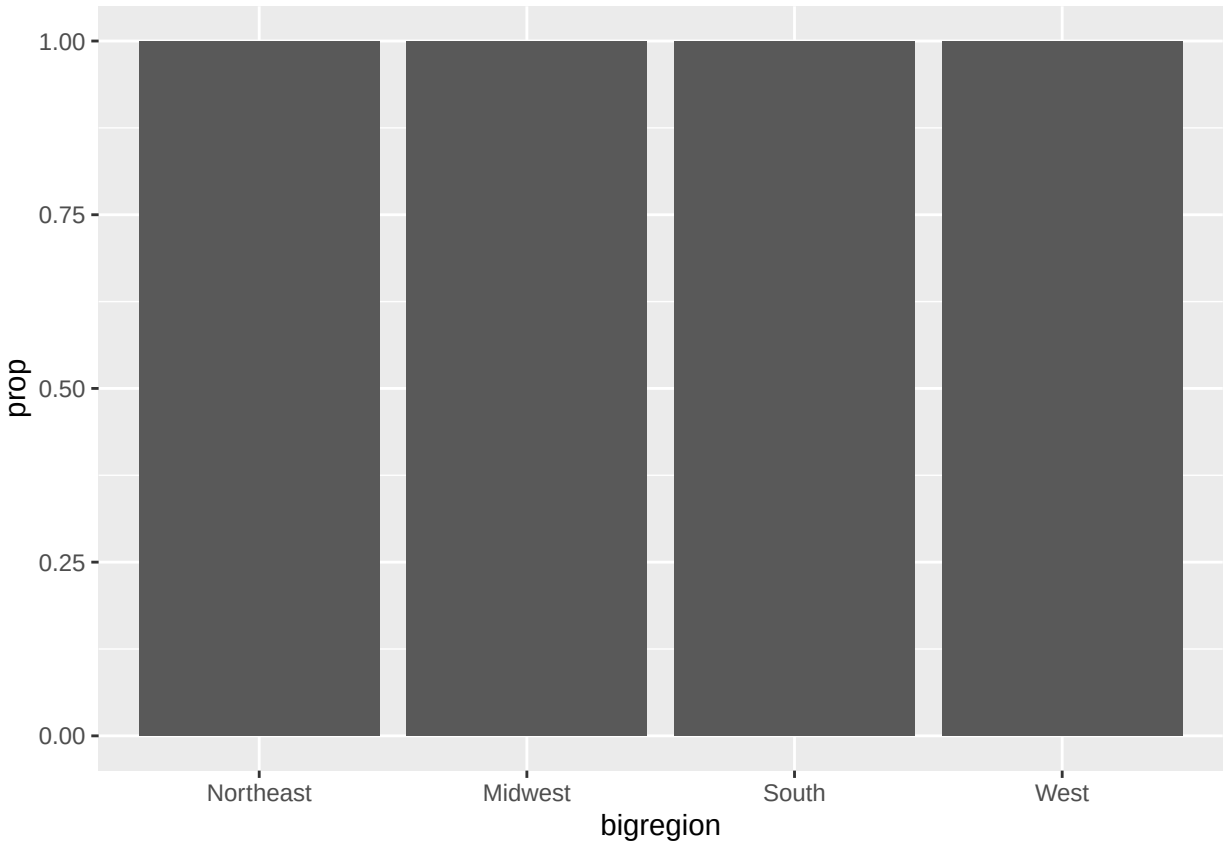
Sometimes the calculations being done by the `stat_` functions that work together with the `geom_` functions might not be immediately obvious. For example, consider this figure produced by a new geom, `geom_bar()`.

```
#install.packages("remotes")
#remotes::install_github("kjhealy/socviz")
library(socviz)
p <- ggplot(data = gss_sm,
            mapping = aes(x = bigregion))
p + geom_bar()
```



Here we specified just one mapping, `aes(x = bigregion)`. The bar chart produced gives us a count of the number of (individual) observations in the data set by region of the United States. This seems sensible. But there is a y-axis variable here, `count`, that is not in the data. It has been calculated for us. Behind the scenes, `geom_bar` called the default `stat_` function associated with it, `stat_count()`. This function computes two new variables, `count`, and `prop` (short for proportion). The `count` statistic is the one 'geom_bar() uses by default.

```
p <- ggplot(data = gss_sm,
            mapping = aes(x = bigregion))
p + geom_bar(mapping = aes(y = ..prop..))
```



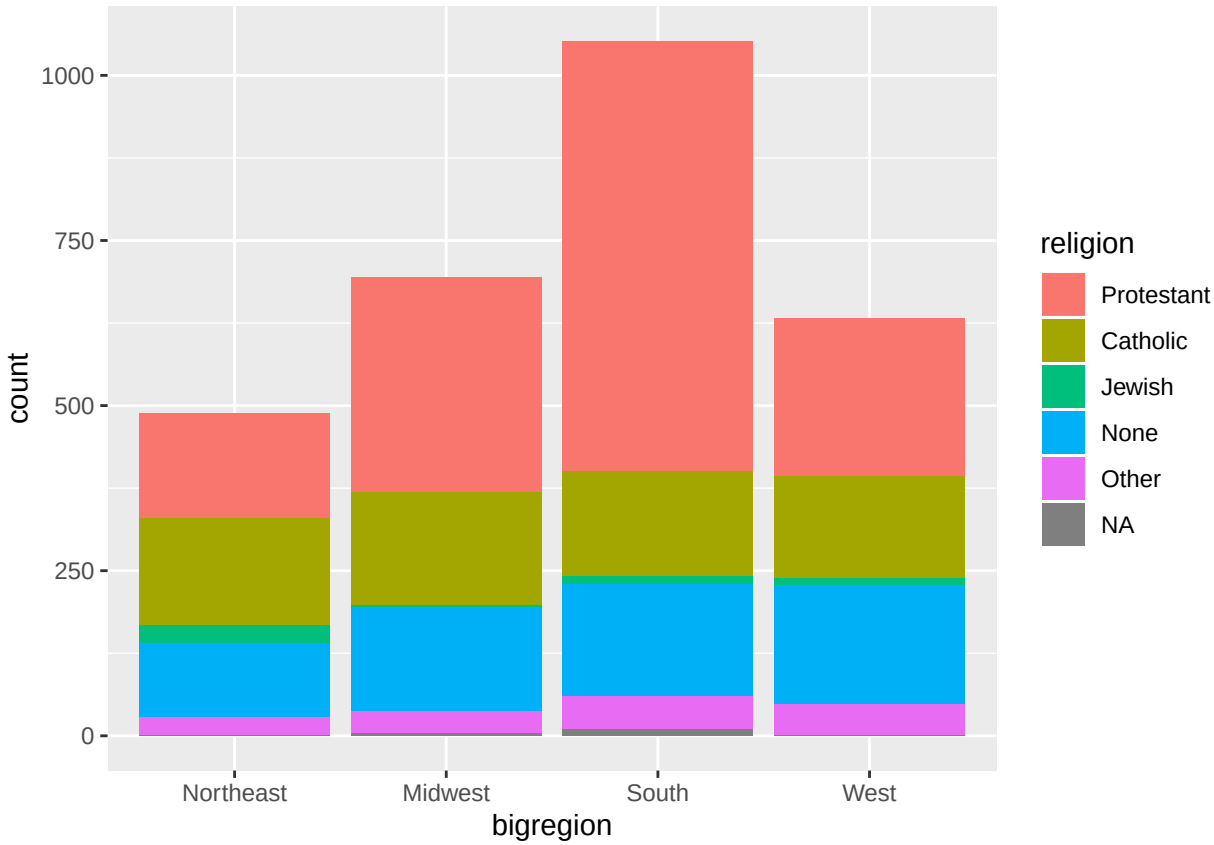
Assignment 3: Geoms and Transformations

Let's look at another question from the survey. The `gss_sm` *data contains a* `religion` *variable derived from a question asking "What is your religious preference? Is it Protestant, Catholic, Jewish, some other religion, or no religion?"*

3a) Run the code below. What is the difference in outcome?

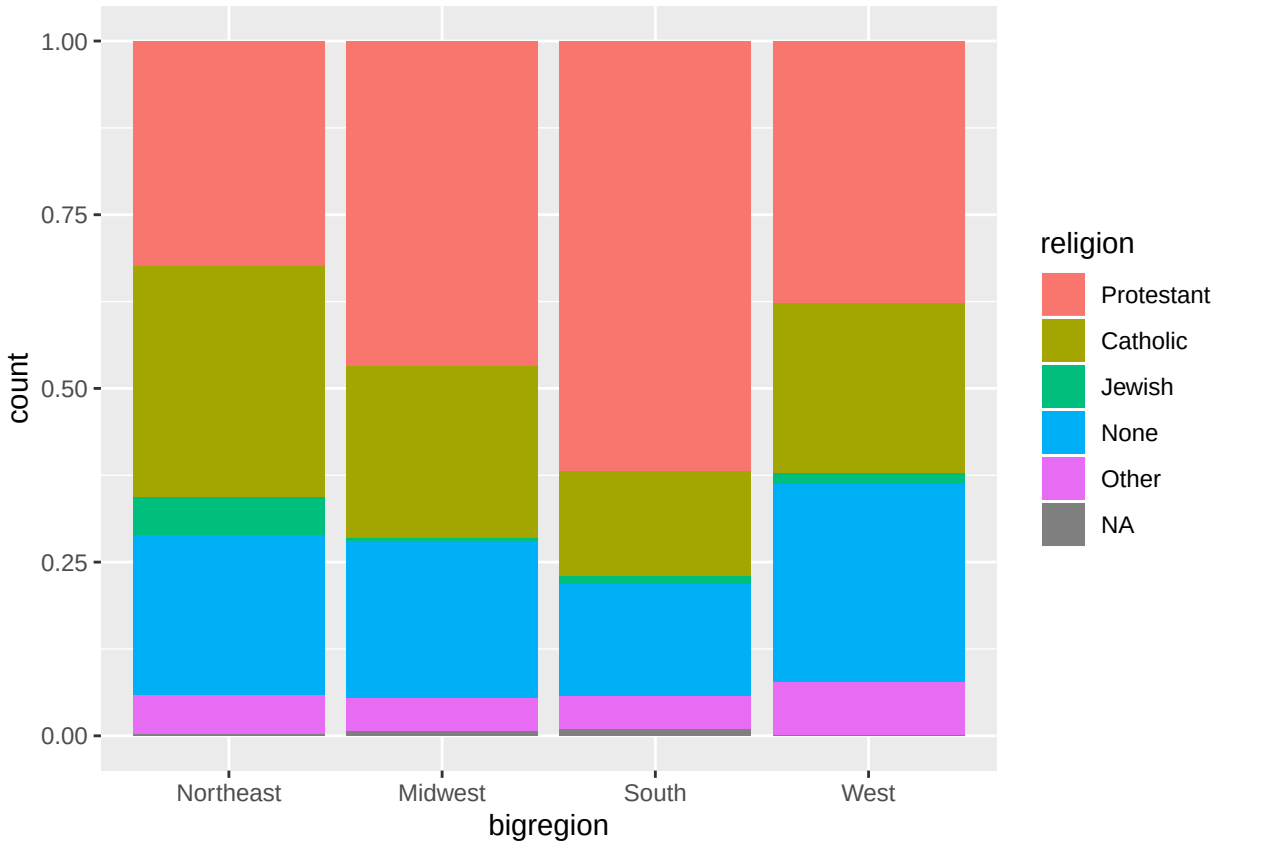
```
p <- ggplot(data = gss_sm,
            mapping = aes(x = religion, color = religion))
p + geom_bar()

p <- ggplot(data = gss_sm,
            mapping = aes(x = religion, fill = religion))
p + geom_bar() + guides(fill = FALSE)
```

3b) A more appropriate use of the `fill` *aesthetic with* `geom_bar()` *is to cross-classify two categorical variables. This is the graphical equivalent of a frequency table of counts or proportions. Using the GSS data, for instance, we might want to examine the distribution of religious preferences within different regions of the United States. Let's say we want to look at religious preference by census region. That is, we want the* `religion` *variable broken down proportionally within* `bigregion`*. When we cross-classify categories in bar charts, there are several ways to display the results. With* `geom_bar()` *the output is controlled by the* `position` *argument. Let's begin by mapping* `fill` *to*`religion`*. Replicate the following graph:*



The default output of `geom_bar()` is a stacked bar chart, with counts on the y-axis (and hence counts within the stacked segments of the bars also). Region of the country is on the x-axis, and counts of religious preference are stacked within the bars. It is somewhat difficult for readers of the chart to compare lengths an areas on an unaligned scale. So while the relative position of the bottom categories are quite clear (thanks to them all being aligned on the x-axis), the relative positions of say, the "Catholic" category is harder to assess. An alternative choice is to set the `position` argument to "`fill`". (This is different from the `fill` aesthetic.) Now the bars are all the same height, which makes it easier to compare proportions across groups:

```
p <- ggplot(data = gss_sm,
            mapping = aes(x = bigregion, fill = religion))
p + geom_bar(position = "fill")
```



Different geoms transform data in different ways, but ggplot's vocabulary for them is consistent. We can see similar transformations at work when summarizing a continuous variable using a histogram, for example. A histogram is a way of summarizing a continuous variable by chopping it up into segments or "bins" and counting how many observations are found within each bin. In a bar chart, the categories are given to us going in (e.g., regions of the country, or religious affiliation). With a histogram, we have to decide how finely to bin the data.

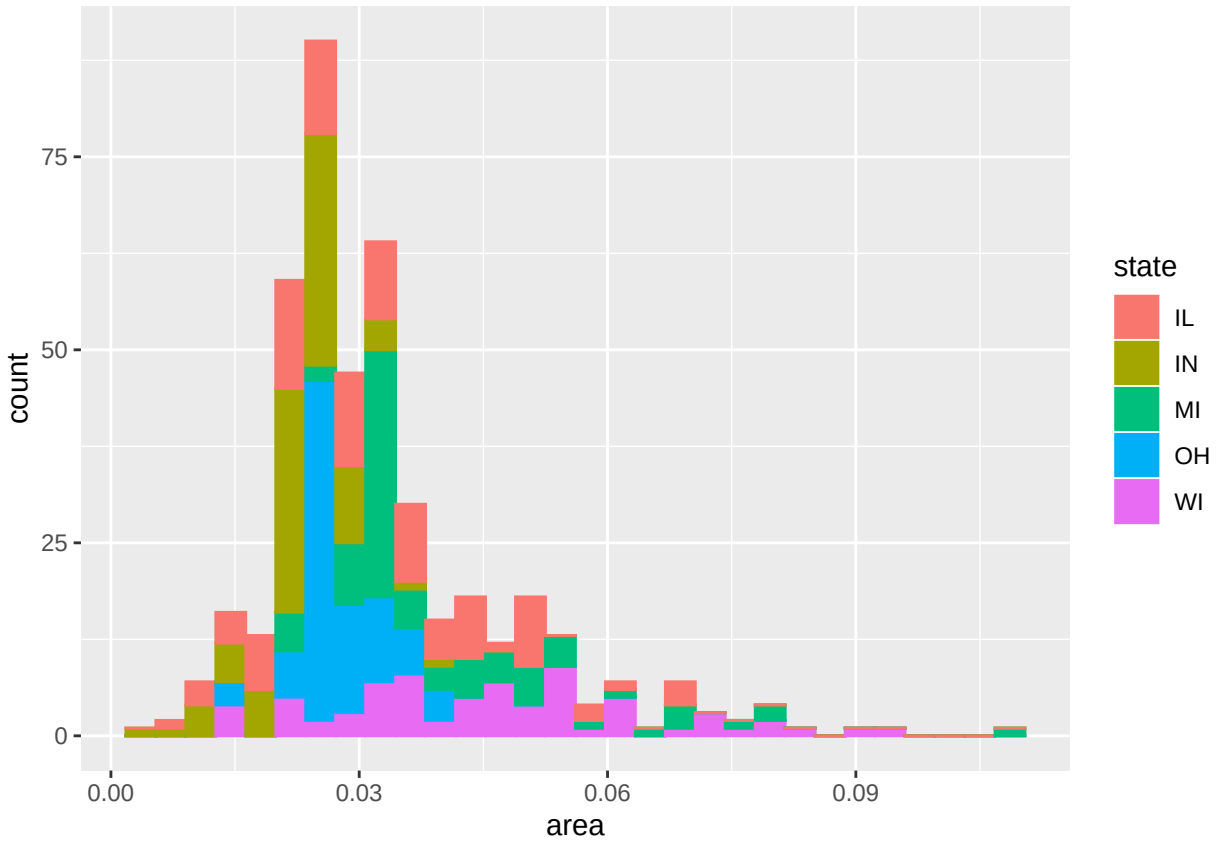
For example, ggplot comes with a dataset, `midwest`, containing information on counties in several midwestern states of the USA. Counties vary in size, so we can make a histogram showing the distribution of their geographical areas. Area is measured in square miles. Because we are summarizing a continuous variable using a series of bars, we need to divide the observations into groups, or bins, and count how many are in each one. By default, the `geom_histogram()` function will choose a bin size for us based on a rule of thumb.

3c) Pick a value for `stat_bin()` to improve the graph below.

```
p <- ggplot(data = midwest,
            mapping = aes(x = area))
p + geom_histogram()
```

3d) While histograms summarize single variables, it's also possible to use several at once to compare distributions. We can facet histograms by some variable of interest, or as here we can compare them in the same plot using the `fill` mapping. Yet, the code below doesn't give us a great overview of the data:

```
p <- ggplot(data = midwest,
            mapping = aes(x = area, fill = state, color = state))
p + geom_histogram()
```



It's better to use `geom_density()`. *Replicate the following graph. (Note: To make the colors more opaque, use the* `alpha` *statement, you can check this in* `?geom_density`*)*

```
p <- ggplot(data = midwest,
            mapping = aes(x = area, fill = state, color = state))
p + geom_density(alpha = 0.3)
```

3e) Frequency polygons are closely related to histograms. Instead of displaying the count of observations using bars, they display it with a series of connected lines instead. Try Figure 4.15 of the book using `geom_freqpoly()` *instead.*

3f) Density estimates can also be drawn in two dimensions. The `geom_density_2d()` *function draws contour lines estimating the joint distribution of two variables. Try it with the* `midwest` *data, for example, plotting percent below the poverty line (*`percbelowpoverty`*) against percent college-educated (*`percollege`*). Try it with and without a 'geom_point() layer.*

Good Luck

The deadline for the assignment is **November 14, 10am**!

References

Healy, K. (2018). *Data visualization: a practical introduction.* Princeton University Press.

Data Science: Visualisations in R - Assignment 3

Healy (2018) Data visualization, Chapters 5 and 8

dr. Mariken A.C.G. van der Velden

November 15, 2019

Instructions for the tutorial

Read the *entire* document that describes the assignment for this tutorial. Follow the indicated steps. Each assignment in the tutorial contains several exercises, divided into numbers and letters (1a, 1b, 1c, 2a, 2b, etc.).

Write down your answers with **the same number-letter combination** in a separate document (e.g., Google Doc), and submit this document to Canvas in a PDF format.

Note: this is the first time that this course has ever been taught. Because of rapidly changing techniques, the course tries to keep up/ reflect those changes. As a result, the assignments may contain errors or ambiguities. If in doubt, ask for clarification during the hall lectures or tutorials. All feedback on the assignments is greatly appreciated!

Refining Your Graphs

Today, we will build upon the knowledge gathered in the last two weeks in three ways:

1. We will learn about how to transform data *before* we send it to ggplot to be turned into a figure. As we saw in last week's assignment, ggplot's geoms will often summarize data for us. While convenient, this can sometimes be awkward or even a little opaque. Often, it's better to get things into the right shape before we send anything to ggplot. This is a job for another tidyverse component, the `dplyr` library. We will learn how to use some of its "action verbs" to select, group, summarize and transform our data.
2. We will expand the number of geoms, and learn more about how to choose between them. The more we learn about ggplot's geoms, the easier it will be to pick the right one given the data we have and the visualization we want. As we learn about new geoms, we will also get a little more adventurous and depart from some of ggplot's default arguments and settings. We will learn how to reorder the variables displayed in our figures, and how to subset the data we use before we display it.
3. The process of gradual customization will give us the opportunity to learn a little more about the `scale`, `guide`, and `theme` functions that we have mostly taken for granted until now. These will give us even more control over the content and appearance of our graphs. Together, these techniques can be used to make plots much more legible to readers. They allow us to present our data in a more structured and easily comprehensible way, and to pick out the elements of it that are of particular interest. We will begin to use these techniques to layer geoms on top of one another, a technique that will allow us to produce very sophisticated graphs in a systematic, comprehensible way.

Still, no matter how complex our plots get, or how many individual steps we take to layer and tweak their features, underneath we will always be doing the same thing. We want a table of tidy data (see example below), a mapping of variables to aesthetic elements, and a particular type of graph. If you can keep sight of this, it will make it easier to confidently approach the job of getting any particular graph to look just right!

```
library(tidyverse)
library(socviz)

table(gss_sm$bigregion, gss_sm$religion, useNA = "ifany")
```

```
##
##             Protestant Catholic Jewish None Other <NA>
##   Northeast        158      162     27  112    28    1
##   Midwest          325      172      3  157    33    5
##   South            650      160     11  170    50   11
##   West             238      155     10  180    48    1
```

Summarize & Transform Data

In Chapter 4 we began making plots of the distributions and relative frequencies of variables. Cross-classifying one measure by another is one of the basic descriptive tasks in data analysis. Tables 5.1 and 5.2 show two common ways of summarizing our GSS data on the distribution of religious affiliation and region. Table 5.1 shows the column marginals, where the numbers sum to a hundred by column and show, e.g., the distribution of Protestants across regions. Meanwhile in Table 5.2 the numbers sum to a hundred across the rows, showing for example the distribution of religious affiliations within any particular region.

As shown schematically in Figure 5.1, we will start with our individual-level table of about 2,500 GSS respondents. Then we want to summarize them into a new table that shows a count of each religious preference, grouped by region. Finally, we will turn these within-region counts into percentages, where the denominator is the total number of respondents within each region. The `dplyr` library provides a few tools to make this easy and clear to read. We will use a special operator,‘%>%, to do our work. This is the *pipe* operator. It plays the role of the yellow triangle in Figure 5.1, in that it helps us perform the actions that get us from one table to the next.

We have been building our plots in an *additive* fashion, starting with a `ggplot` object and layering on new elements. By analogy, think of the `%>%` operator as allowing us to start with a data frame and perform a *sequence* or *pipeline* of operations to turn it into another, usually smaller and more aggregated table. Data goes in one side of the pipe, actions are performed via functions, and results come out the other. A pipeline is typically a series of operations that do one or more of four things:

- *Group* the data into the nested structure we want for our summary, such as “Religion by Region” or “Authors by Publications by Year”. This can be done with the function `group_by()`.
- *Filter* or *select* pieces of the data by row, column, or both. This gets us the piece of the table we want to work on. This can be done with the functions `filter()` (for rows) and `select()` (for columns).
- *Mutate* the data by creating new variables at the current level of grouping. This adds new columns to the table without aggregating it. This can be done with the function `mutate()`.
- *Summarize* or *aggregate* the grouped data. This creates new variables at a higher level of grouping. For example we might calculate means with `mean()` or counts with `n()`. This results in a smaller, summary table, which we might do more things on if we want. This can be done within the function `summarize()`.

We use the `dplyr` functions `group_by()`, `filter()`, `select()`, `mutate()`, and `summarize()` to carry out these tasks within our *pipeline*. They are written in a way that allows them to be easily piped. That is, they understand how to take inputs from the left side of a pipe operator and pass results along through the right side of one. The dplyr documentation has some useful vignettes that introduce these grouping, filtering, selection, and transformation functions. There is also a more detailed discussion of these tools, along with many more examples, in Wickham & Grolemund (https://r4ds.had.co.nz/).

We will create a new table called `rel_by_region`. Here’s the code:

```
(rel_by_region <- gss_sm %>% #Create a new object, rel_by_region from the gss_sm data
    group_by(bigregion, religion) %>% # Subsequently group the rows by bigregion and,
  #within that, by religion
    summarize(N = n()) %>% #Summarize this table to create a new, much smaller table,
  #with three columns: bigregion, religion, and a new summary variable, N, that is a
```

```
  #count of the number of observations within each religious group for each region.
    mutate(freq = N / sum(N), # With this new table, use the N variable to calculate
  #the relative proportion (freq)
           pct = round((freq*100), 0))) #and percentage (pct) for each religious
```

```
## # A tibble: 24 x 5
## # Groups:   bigregion [4]
##    bigregion religion       N    freq   pct
##    <fct>     <fct>      <int>   <dbl> <dbl>
##  1 Northeast Protestant   158 0.324      32
##  2 Northeast Catholic     162 0.332      33
##  3 Northeast Jewish        27 0.0553      6
##  4 Northeast None         112 0.230      23
##  5 Northeast Other         28 0.0574      6
##  6 Northeast <NA>           1 0.00205     0
##  7 Midwest   Protestant   325 0.468      47
##  8 Midwest   Catholic     172 0.247      25
##  9 Midwest   Jewish         3 0.00432     0
## 10 Midwest   None         157 0.226      23
## # ... with 14 more rows
```

```
  #category, still grouped by region. Round the results to the nearest percentage
  #point

### Putting brackets () around a statement with a data set, makes R visualize the
### first 10 rows
```

In this way of doing things, objects passed along the pipeline and the functions acting on them carry some assumptions about their context. For one thing, you don't have to keep specifying the name of the underlying data frame object you are working from. Everything is implicitly carried forward from `gss_sm`. Within the pipeline, the transient or implicit objects created from your summaries and other transformations are carried through, too.

When trying to grasp what each additive step in a `ggplot()` sequence does, it can be helpful to work backwards, removing one piece at a time to see what the plot looks like when that step is not included. In the same way, when looking at pipelined code it can be helpful to start from the end of the line, and then remove one `%>%` step at a time to see what the resulting intermediate object looks like.

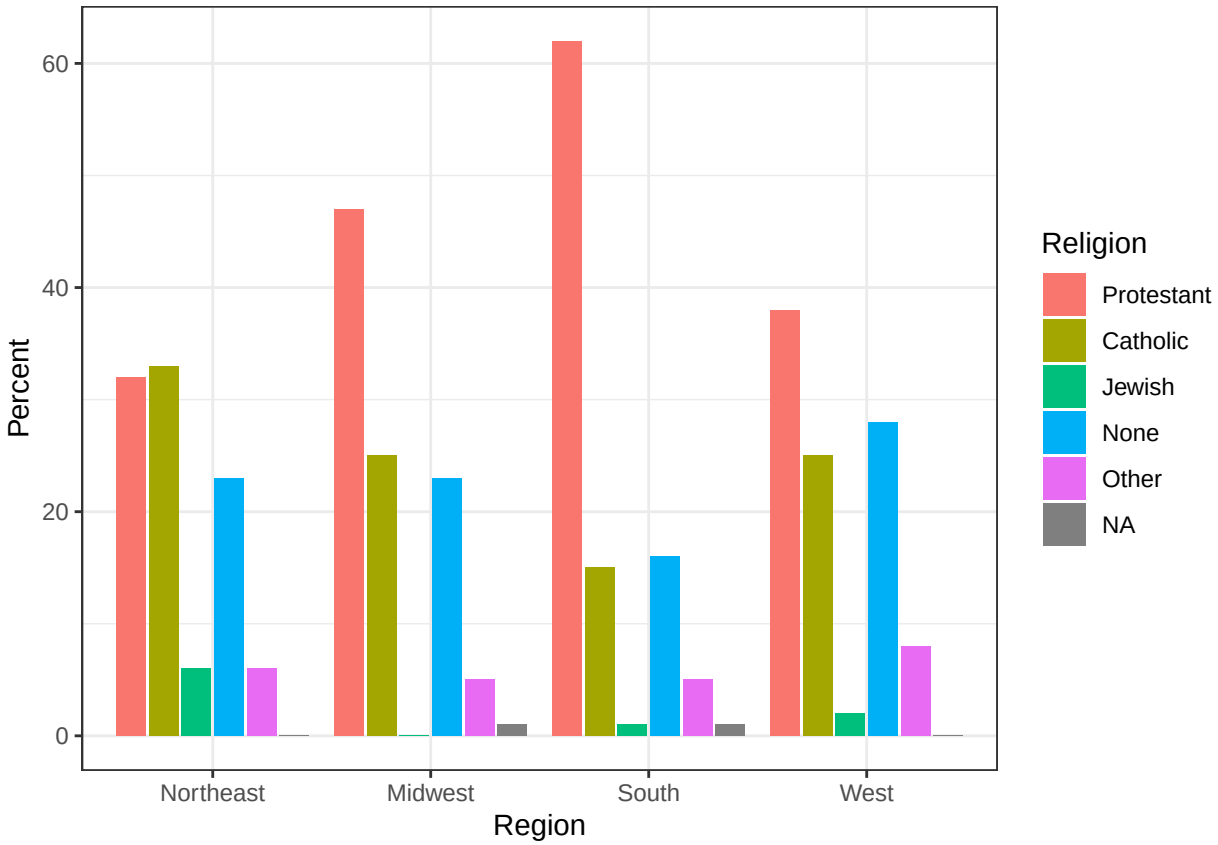
Assignment 1: Deconstructing a Pipeline

1a) What happens if we remove the `mutate()` step from the code above? Provide the script and a table in your assignment.

1b) Subsequently, what happens if you also remove `summarize()` step? How big is the table returned at each step? What level of grouping is it at? What variables have been added or removed?

1c) Replicate the code below. `geom_col()` is used instead of `geom_bar()`. What is changed in the plot if you use `geom_bar()`?

```
ggplot(rel_by_region, aes(x = bigregion, y = pct, fill = religion)) +
  geom_col(position = "dodge2") +
    labs(x = "Region",y = "Percent", fill = "Religion") +
    theme(legend.position = "top") +
  theme_bw()
```



1d) Moreover `dodge2` *is used instead of* `dodge` *like in last week's assignment. What is the difference?*

1e) Add the functions `coord_flip()` *and* `facet_grid()` *to the* `ggplot` *object. Explain what happens and whether this aids or impedes the interpretation of the visual.*

Continue with Grouping

Let's move to a new dataset, the `organdata` table. Like `gapminder`, it has a country-year structure. It contains a little more than a decade's worth of information on the donation of organs for transplants in seventeen OECD countries. The organ procurement rate is a measure of the number of human organs obtained from cadaver organ donors for use in transplant operations. Along with this donation data, the dataset has a variety of numerical demographic measures, and several categorical measures of health and welfare policy and law. Unlike the `gapminder` data, some observations are missing. These are designated with a value of `NA`, R's standard code for missing data. The `organdata` table is included in the `socviz` library. Load it up and take a quick look. Instead of using `head()`, for variety this time we will make a short pipeline to select the first six columns of the dataset, and then pick five rows at random using a function called `sample_n()`. This function takes two main arguments. First we provide the table of data we want to sample from. Because we are using a pipeline, this is implicitly passed down from the beginning of the pipe. Then we supply the number of draws we want to make.

```
organdata %>% select(1:6) %>% sample_n(size = 10)
```

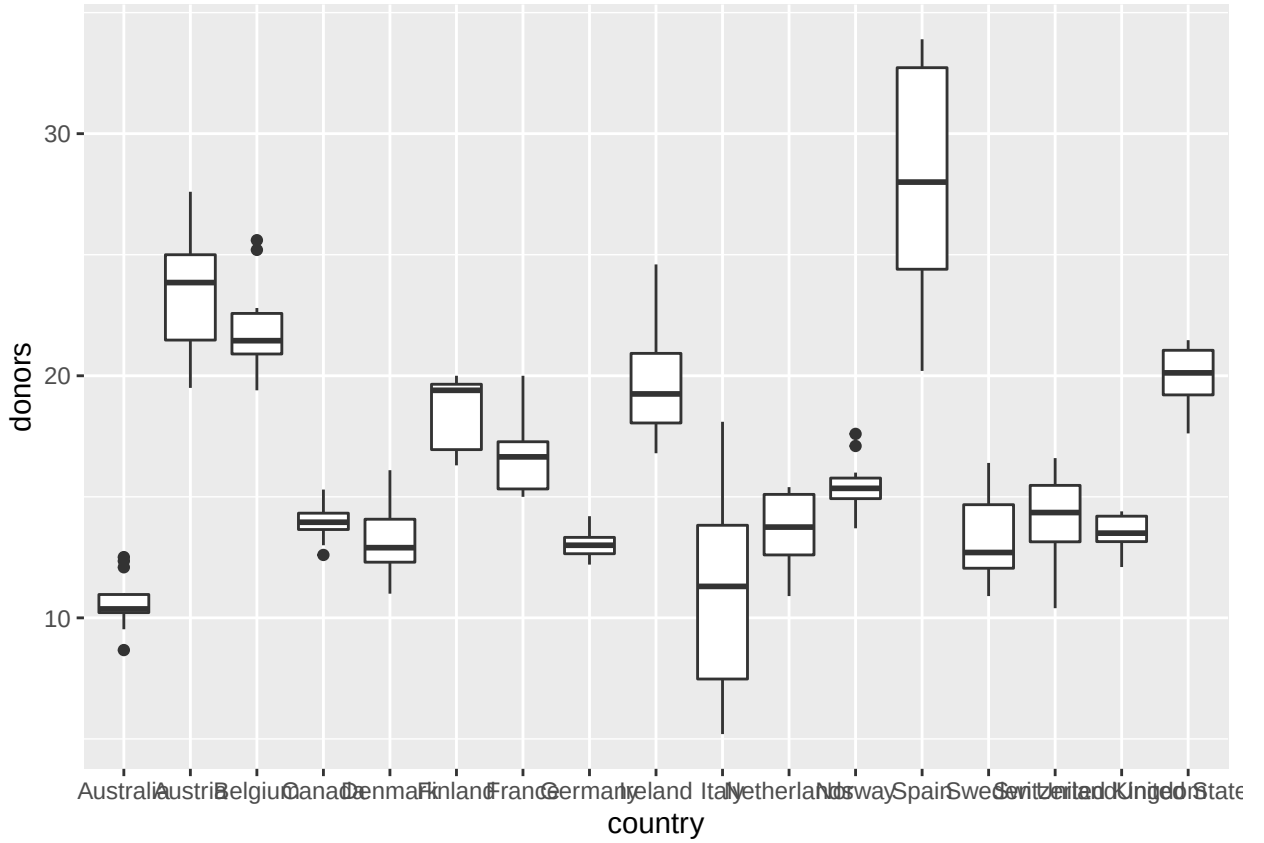
```
## # A tibble: 10 x 6
##    country        year       donors    pop pop_dens   gdp
##    <chr>          <date>      <dbl>  <int>    <dbl> <int>
##  1 Australia      1998-01-01   10.5  18711    0.242 24148
##  2 France         1995-01-01   15.1  57844   10.5   21283
##  3 Switzerland    2002-01-01   10.4   7290   17.7   30725
```

```
##  4 Switzerland    2000-01-01   14     7184   17.4   29837
##  5 Sweden         NA           NA     8559    1.90  18660
##  6 Netherlands    1997-01-01   14.4  15611   37.6   23753
##  7 Australia      2000-01-01   10.2  19153    0.247 26545
##  8 United Kingdom 1995-01-01   14.4  58005   23.9   19998
##  9 Germany        1997-01-01   13.2  82035   23.0   22589
## 10 United States  2000-01-01   21.2 282224    2.93  34590
```

Let's imagine we are interested in country-level variation, without paying attention to the time trend. We can use `geom_boxplot()` to get a picture of variation by year across countries. Just as `geom_bar()` by default calculates a count of observations by the category you map to `x`, the `stat_boxplot()` function that works with `geom_boxplot()` will calculate a number of statistics that allow the box and whiskers to be drawn. We tell `geom_boxplot()` the variable we want to categorize by (here: `country`) and the continuous variable we want summarized (here: `donors`).

```
ggplot(data = organdata,
       mapping = aes(x = country, y = donors)) +
  geom_boxplot()
```

```
## Warning: Removed 34 rows containing non-finite values (stat_boxplot).
```

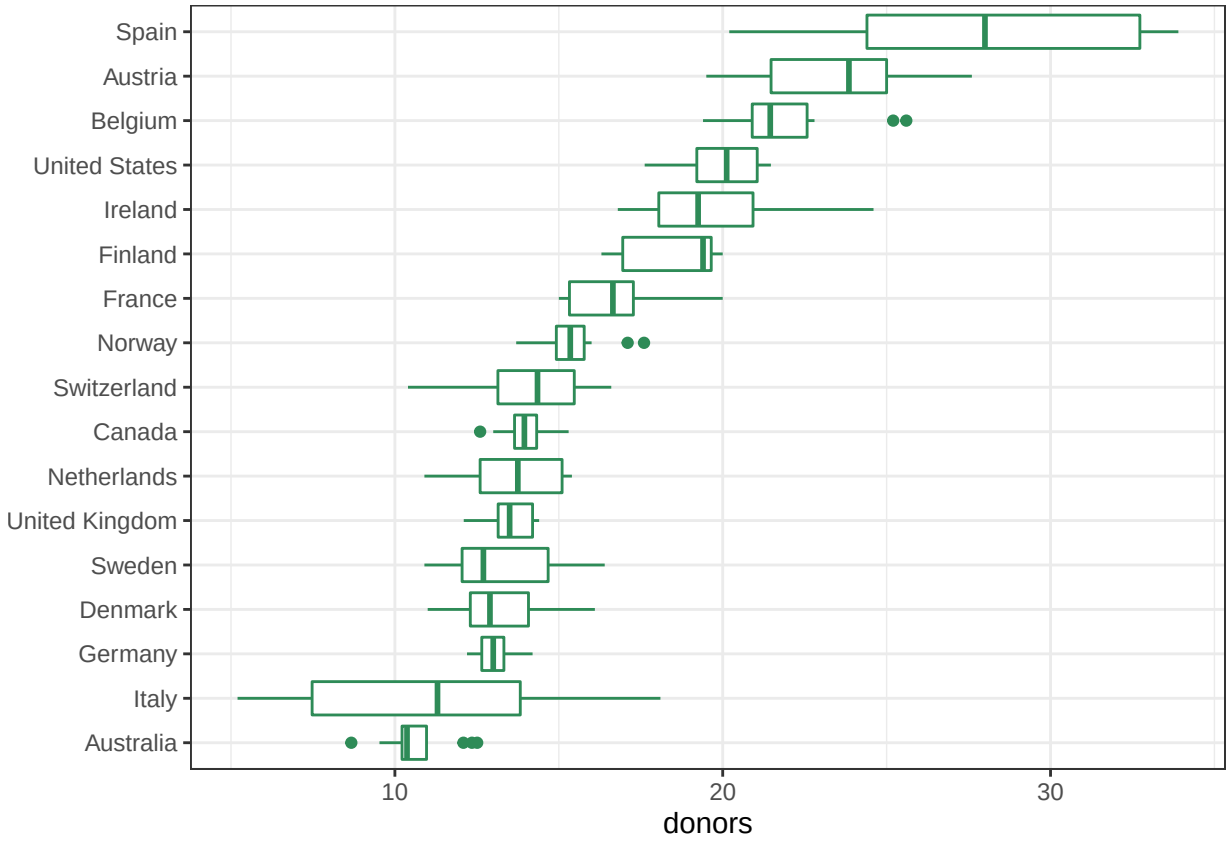


The boxplots look interesting but two issues could be addressed. First, as we saw in the previous chapter, it is awkward to have the country names on the x-axis because the labels will overlap. We should use `coord_flip()` again to switch the axes (but not the mappings). While that will make the graph more legible, it's still not ideal. We generally want our plots to present data in some meaningful *order*. An obvious way is to have the countries listed from high to low average donation rate. We accomplish this by reordering the country variable by the mean of donors. The `reorder()` function will do this for us. It takes two required arguments:

1. The categorical variable or factor that we want to reorder. In this case, that's country.
2. The variable we want to reorder it by. Here that is the donation rate, donors.

The third and optional argument to `reorder()` is the function you want to use as a summary statistic. If you only give `reorder()` the first two required arguments, then by default it will reorder the categories of your first variable by the mean value of the second. You can name any sensible function you like to reorder the categorical variable (e.g., `median`, or `sd`). There is one additional wrinkle. In R, the default `mean()` function will fail with an error if there are missing values in the variable you are trying to take the average of. You must say that it is OK to remove the missing values when calculating the mean. This is done by supplying the `na.rm=TRUE` argument to `reorder()`, which internally passes that argument on to `mean()`. We are reordering the variable we are mapping to the `x` aesthetic, so we use `reorder()` at that point in our code:

```
ggplot(data = organdata,
       mapping = aes(x = reorder(country, donors, na.rm=TRUE),
                     y = donors)) +
  geom_boxplot(colour = "seagreen") +
  labs(x=NULL) +
  coord_flip() +
  theme_bw()
```



Assignment 2: Apply Transformations & Different Ways of Visualizing Data

2a) Replicate the code above, but used `geom_violin()` *instead of* `geom_boxplot()`

2b) What is the difference with the `geom_boxplot()`*? What do the shapes of the* `geom_violin()` *tell you?*

2c) The `subset()` *function is very useful when used in conjunction with a series of layered geoms. Go back to the book's code for the* Presidential Elections plot *(Figure 5.18) and redo it so that it shows all the data*

points but only labels elections since 1992. You might need to look again at the `elections_historic` *data to see what variables are available to you.*

2d) Continue with the plot created in 2c and colour the data points to reflect the winning party.

2e) Use `geom_point()` *and* `reorder()` *to make a Cleveland dot plot (e.g. Figure 5.13) of all Presidential elections, ordered by share of the popular vote.*

2f) Install the packages `ggthemes` *and replicate the graph of 2e with various themes. What kind of backgrounds aid interpretation? Why? Include the code and plots in the assignment.*

Good Luck

The deadline for the assignment is **November 21, 10am**!

References

Healy, K. (2018). *Data visualization: a practical introduction.* Princeton University Press.

Data Science: Visualisations in R - Assignment 4

Healy (2018) Data visualization, Chapter 6

dr. Mariken A.C.G. van der Velden

November 22, 2019

Instructions for the tutorial

Read the *entire* document that describes the assignment for this tutorial. Follow the indicated steps. Each assignment in the tutorial contains several exercises, divided into numbers and letters (1a, 1b, 1c, 2a, 2b, etc.).

Write down your answers with **the same number-letter combination** in a separate document (e.g., Google Doc), and submit this document to Canvas in a PDF format.

Note: this is the first time that this course has ever been taught. Because of rapidly changing techniques, the course tries to keep up/ reflect those changes. As a result, the assignments may contain errors or ambiguities. If in doubt, ask for clarification during the hall lectures or tutorials. All feedback on the assignments is greatly appreciated!

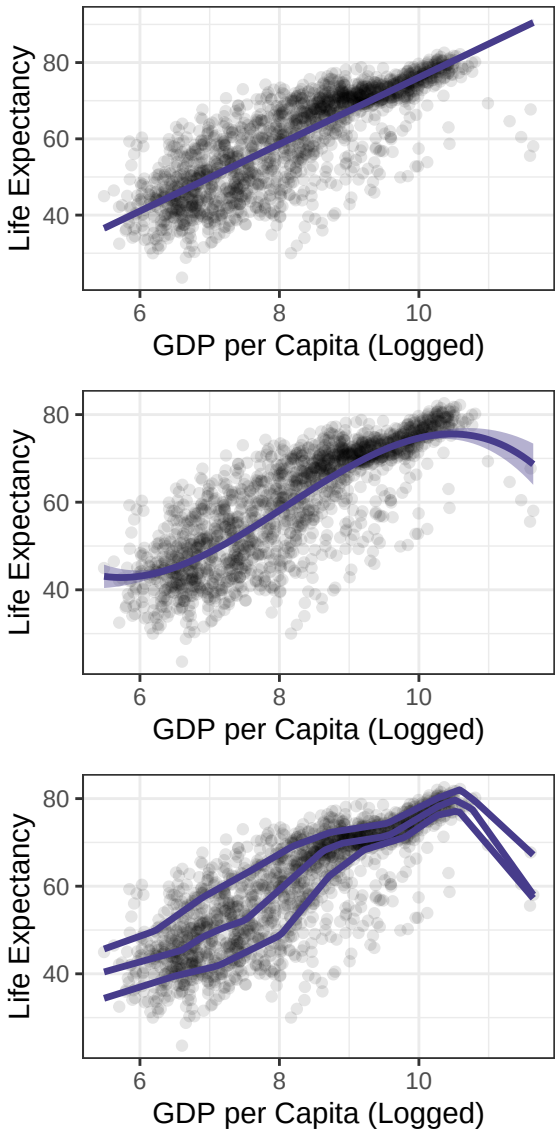
Uncertainty in the Social Science

Data visualization is about more than generating figures that display the raw numbers from a table of data. As we have seen in the last three weeks, it can involve summarizing or transforming parts of the data and plot the results thereof. Statistical models are a central part of the process. The goal of a statistical model is to provide a simple low-dimensional summary of a dataset. Ideally, the model will capture true "*signals*" (i.e. patterns generated by the phenomenon of interest), and ignore "*noise*" (i.e. random variation that you're not interested in).

Today, we will explore how ggplot can be used for various modeling techniques directly with geoms. We will use the `broom` and `margins` libraries to tidily extract and plot estimates from models that we fit ourselves. For this tutorial, it's not necessary to have a solid background or understanding of statistics.[1]

The below displayed graphs demonstrate different ways to show signals and noise in data. For all three graphs, the `geom_point()` displays the relation between the variables `lifeExp` (life expectancy) and `gdpPercap` (GDP per capita). The relation is positive and linear, meaning that the richer a country is (i.e. higher GDP per capita), the higher the citizens are expected to live (i.e. higher levels of life expectancy). This relationship could already be observed by 'eye-balling' the data - i.e. just looking at a simple dotplot. Yet, when we want model the 'exact' relation between the two variables, there are various ways to do that. *First*, you can add a `geom_smooth()` layer. This line tries to adequately capture the underlying pattern in the data - i.e. the signal. This layer gives you many options, the upper and middle plot both have a smoothener added. Still, those two lines differ. The upper plot shows a linear display of the relation, while the middle plot shows a non-linear version. The lighter area around the solid line of the middle plot displays the standard errors around the estimated relation between `gdpPercap` and `lifeExp`. The standard error (SE) of a statistic (i.e. an estimate of a parameter) is an estimate of that standard deviation, it displays the variation around the estimation: The narrower the standard error around an estimate, the more certain we can be that the estimate reflects some pattern in the real world. *Second*, to visualize the relation between `gdpPercap` and `lifeExp`, one could choose to display the different *quantiles*. In the lower plot, three lines are visualized: The lower one displaying the 20th percentile, the middle one the 50th percentile and the upper one the 85th percentile. That means that 65% of the observations are between the lower and upper line.

[1] For a comprehensive, modern introduction to that topic you should work your way through Gelman & Hill (2018). Harrell (2016) is also very good on the many practical connections between modeling and graphing data. Similarly, Gelman (2004) provides a detailed discussion of the use of graphics as a tool in model-checking and validation.



Today, we will discuss some ways to take the models that you fit and extract information that is easy to work with in ggplot. Our goal, as always, is to get from however the object is stored to a tidy table of numbers that we can plot. Most classes of statistical models in R will contain the information we need, or will have a special set of functions, or methods, designed to extract it. We can start by learning a little more about how the output of models is stored in R. Remember, we are always working with objects, and objects have an internal structure consisting of named pieces. Sometimes these are single numbers, sometimes vectors, and sometimes lists of things like vectors, matrices, or formulas. We have been mainly working with tibbles and data frames, like e.g. the `gapminder` data:

```
head(gapminder)
```

```
## # A tibble: 6 x 6
##   country     continent  year lifeExp      pop gdpPercap
##   <fct>       <fct>     <int>   <dbl>    <int>     <dbl>
## 1 Afghanistan Asia       1952    28.8  8425333      779.
## 2 Afghanistan Asia       1957    30.3  9240934      821.
## 3 Afghanistan Asia       1962    32.0 10267083      853.
## 4 Afghanistan Asia       1967    34.0 11537966      836.
## 5 Afghanistan Asia       1972    36.1 13079460      740.
```

```
## 6 Afghanistan Asia       1977    38.4 14880372      786.
```

Tibbles and data frames store tables of data with named columns, perhaps consisting of different classes of variable, such as integers, characters, dates, or factors. We can use `str()` to learn more about the internal structure of any object. For example, we can get some information on what class (or classes) of object `gapminder` is, how large it is, and what components it has. The output from `str(gapminder)` is somewhat dense:

```
str(gapminder)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    1704 obs. of  6 variables:
##  $ country  : Factor w/ 142 levels "Afghanistan",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ continent: Factor w/ 5 levels "Africa","Americas",..: 3 3 3 3 3 3 3 3 3 3 ...
##  $ year     : int  1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 ...
##  $ lifeExp  : num  28.8 30.3 32 34 36.1 ...
##  $ pop      : int  8425333 9240934 10267083 11537966 13079460 14880372 12881816 13867957 16317921 22
##  $ gdpPercap: num  779 821 853 836 740 ...
```

There is a lot of information here about the object as a whole and each variable in it. In the same way, statistical models in R have an internal structure. Yet, because models are more complex entities than data tables, their structure is correspondingly more complicated. There are more *pieces* of information, and more *kinds* of information, that we might want to use. All of this information is generally stored in or is computable from parts of a model object.

We can create a linear model, an ordinary OLS regression, using the `gapminder` data. As a side note, if you would work with this data on a paper, you have to be aware that this dataset has a country-year structure, which makes an OLS specification like this the wrong one to use. Never mind that for now. We use the `lm()` function to run the model, and store it in an object called `out`:

```
(out <- lm(formula = lifeExp ~ gdpPercap + pop + continent,
           data = gapminder))
```

The first argument is the formula for the model. `lifeExp` is the dependent variable and the tilde `~` operator is used to designate the left- and right-hand sides of a model (including in cases, as we saw with `facet_wrap()` where the model just has a right-hand side.)

Assignment 1: Models & Their Structure

1a) Try out `summary(out)` and look at the summary. Add a screenshot to your assignment.

1b) When we use `summary()` on `out`, we are not getting a simple feed of what's in the model object. Instead, like any function, `summary()` takes its input, performs some actions, and produces output. What is printed to the console is partly information that is stored inside the model object, and partly information that the `summary()` function has calculated and formatted for display on the screen. Behind the scenes, `summary()` gets help from other functions. Objects of different classes have default methods associated with them. Try `summary(gapminder)`, what type of information is given? What are the similarities and differences with `summary(out)`'?

1c) The output from `summary(out)` gives a precis of the model, but we can't really do any further analysis with it directly. For example, what if we want to plot something from the model? The information necessary to make plots is inside the `out` object, but it is not obvious how to use it. The figure below displays the structure of the `out` object. In this list of items, elements are single values, some are data frames, and some are additional lists of simpler items. List objects could be thought of as being organized like a filing system: cabinets contain drawers, and drawers may contain folders, which may contain pages of information, whole documents, or groups of folders with more documents inside. Try `out$coefficients` and `out$fitted.values` in the console. What do you get?

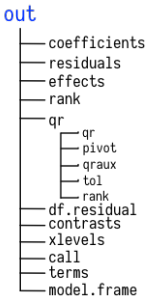


Figure 1: Structure of a object for a linear model

Visually Presenting Models

Figures based on statistical models face all the ordinary challenges of effective data visualization, and then some. This is because model results usually carry a considerable extra burden of interpretation and necessary background knowledge. The more complex the model, the trickier it becomes to convey this information effectively, and the easier it becomes to lead one's audience or oneself into error. Within the social sciences, our ability to clearly and honestly present model-based graphics has greatly improved over the past ten or fifteen years. Over the same period, it has become clearer that some kinds of models are quite tricky to understand, even ones that had previously been seen as straightforward elements of the modeling toolkit (Ai & Norton, 2003; Brambor, Clark, & Golder, 2006).

Plotting model estimates is closely connected to properly estimating models in the first place. This means there is no substitute for learning the statistics! You should not use graphical methods as a substitute for understanding the model used to produce them. While we cannot teach you that material in the scope of the course, we can make a few general points about what good model-based graphics look like, and work through some examples of how `ggplot` and some additional libraries can make it easier to get good results.

1. Useful model-based plots show results in ways that are substantively meaningful and directly interpretable with respect to the questions the analysis is trying to answer. This means showing results in a context where other variables in the analysis are held at sensible values, such as their means or medians. With continuous variables, it can often be useful to generate predicted values that cover some substantively meaningful move across the distribution, such as from the 25th to the 75th percentile, rather than a single-unit increment in the variable of interest. For unordered categorical variables, predicted values might be presented with respect to the modal category in the data, or for a particular category of theoretical interest. Presenting substantively interpretable findings often also means using (and sometimes converting to) a scale that readers can easily understand. There is nothing distinctively graphical about putting the focus on the substantive meaning of your findings.

2. Much the same applies to presenting the degree of uncertainty or confidence you have in your results. Model estimates come with various measures of precision, confidence, credence, or significance. Presenting and interpreting these measures is notoriously prone to misinterpretation, or over-interpretation, as researchers and audiences both demand more from things like confidence intervals and p-values than these statistics can deliver. At a minimum, having decided on an appropriate measure of model fit or the right assessment of confidence, you should show their range when you present your results. A family of related `ggplot` geoms allow you to show a range or interval defined by position on the x-axis and then a `ymin` and `ymax` range on the y-axis. These geoms include `geom_pointrange()` and `geom_errorbar()`, which we will see in action shortly. A related geom, `geom_ribbon()` uses the same arguments to draw filled areas, and is useful for plotting ranges of y-axis values along some continuously varying x-axis.

3. Plotting the results from a model with multiple variable (so-called multivariate model) generally means one of two things. First, we can show what is in effect a table of coefficients with associated measures of confidence, perhaps organizing the coefficients into meaningful groups, or by the size of the predicted

association, or both. Second, we can show the predicted values of some variables (rather than just a model's coefficients) across some range of interest. The latter approach lets us show the original data points if we wish. The way `ggplot` builds graphics layer by layer allows us to easily combine model estimates (e.g. a regression line and an associated range) and the underlying data. In effect these are manually-constructed versions of the automatically-generated plots that we have been producing with `geom_smooth()`.

Having fitted a model, then, we might want to get a picture of the estimates it produces over the range of some particular variable, holding other covariates constant at some sensible values. The `predict()` function is a generic way of using model objects to produce this kind of prediction. In R, "generic" functions take their inputs and pass them along to more specific functions behind the scenes, ones that are suited to working with the particular kind of model object we have. The details of getting predicted values from a OLS model, for instance, will be somewhat different from getting predictions out of a different type of regression, such as a logistic regression. But in each case we can use the same `predict()` function, taking care to check the documentation to see what form the results are returned in for the kind of model we are working with. Many of the most commonly-used functions in R are generic in this way. The `summary()` function, for example, works on objects of many different classes, from vectors to data frames and statistical models, producing appropriate output in each case by way of a class-specific function in the background.

For `predict()` to calculate the new values for us, it needs some new data to fit the model to. We will generate a new data frame whose columns have the same names as the variables in the model's original data, but where the rows have new values. A very useful function called `expand.grid()` will help us do this. We will give it a list of variables, specifying the range of values we want each variable to take. Then `expand.grid()` will generate the will multiply out the full range of values for all combinations of the values we give it, thus creating a new data frame with the new data we need.

In the following bit of code, we use `min()` and `max()` to get the minimum and maximum values for GDP per capita, and then create a vector with one hundred evenly-spaced elements between the minimum and the maximum. We hold population constant at its median, and we let continent take all of its five available values:

```
(min_gdp <- min(gapminder$gdpPercap))
(max_gdp <- max(gapminder$gdpPercap))
(med_pop <- median(gapminder$pop))

(pred_df <- expand.grid(gdpPercap = (seq(from = min_gdp,
                                         to = max_gdp,
                                         length.out = 100)),
                        pop = med_pop,
                        continent = c("Africa", "Americas",
                                      "Asia", "Europe", "Oceania")))
```

Now we can use `predict()`. If we give the function our new data and model, without any further argument, it will calculate the fitted values for every row in the data frame. If we specify `interval = 'predict'` as an argument, it will calculate 95% prediction intervals in addition to the point estimate.

```
(pred_out <- predict(object = out,
                     newdata = pred_df,
                     interval = "predict"))
```

Because we know that, by construction, the cases in `pred_df` and `pred_out` correspond row for row, we can bind the two data frames together by column. Note: this method of joining or merging tables is definitely not recommended when you are dealing with data!

```
pred_df <- cbind(pred_df, pred_out)
```

The end result is a tidy data frame, containing the predicted values from the model for the range of values we specified. Now we can plot the results. Because we produced a full range of predicted values, we can decide

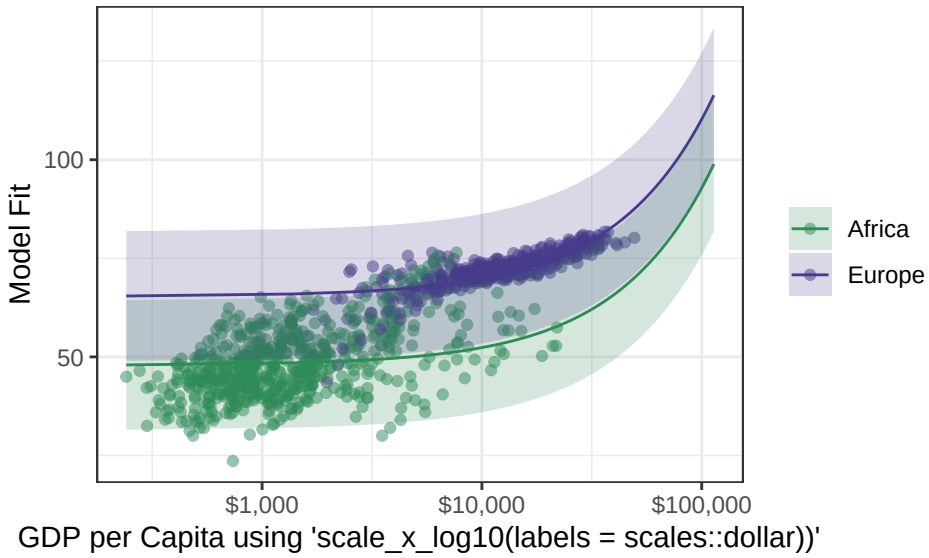
whether or not to use all of them.

Assignment 2: Plot a fitted model

2a) How many columns does the new data set `pred_df` *have? And how many rows? Use* `dim(pred_df)`.

2b) And how many columns does the new data set `pred_out` *have? And how many rows? Use* `dim(pred_out)`.

2c) Look at the plot displayed below. This is based on the predicted data (`pred_df`*) - displayed by the line and the confidence intervals around the line using* `geom_ribbon()` *- as well as on the real data from* `gapminder` *- displayed with* `geom_point()`. *The data is filtered, which data is included and which is not? Tip: see* `?table` *to find out information about a variable (e.g.* `table(gapminder$continent)`.



2d) Describe the layers (i.e. geoms) that you see in the graph.

2e) Replicate the graph. I've done some filtering using `data = subset(NAME DATA, continent %in% c("Europe", "Africa"))`. *Moreover, I manually picked the colors for the two continents. I've used "seagreen" and "slateblue4", but have a look at http://www.stat.columbia.edu/~tzheng/files/Rcolor.pdf and pick your own favorite colors. Additionally, I set a theme (*`theme_bw()`*), and removed the title of the legend with* `theme(legend.title=element_blank())`.

Good Luck

The deadline for the assignment is **November 28, 10am**!

References

Healy, K. (2018). *Data visualization: a practical introduction*. Princeton University Press.

Data Science: Visualisations in R - Assignment 5

Github of Alex Hanna and Robin Love Lace, Healy (2018) Data visualization, Chapter 7

dr. Mariken A.C.G. van der Velden

November 29, 2019

Instructions for the tutorial

Read the *entire* document that describes the assignment for this tutorial. Follow the indicated steps. Each assignment in the tutorial contains several exercises, divided into numbers and letters (1a, 1b, 1c, 2a, 2b, etc.).

Write down your answers with **the same number-letter combination** in a separate document (e.g., Google Doc), and submit this document to Canvas in a PDF format.

Note: this is the first time that this course has ever been taught. Because of rapidly changing techniques, the course tries to keep up/ reflect those changes. As a result, the assignments may contain errors or ambiguities. If in doubt, ask for clarification during the hall lectures or tutorials. All feedback on the assignments is greatly appreciated!

Relational Data & Networks

It's rare that a data analysis involves only a single table of data. Typically you have many tables of data, and you must combine them to answer the questions that you're interested in. Collectively, multiple tables of data are called *relational data* because it is the relations, not just the individual datasets, that are important.

Relations are always defined between a pair of tables. All other relations are built up from this simple idea: the relations of three or more tables are always a property of the relations between each pair. Sometimes both elements of a pair can be the same table! This is needed if, for example, you have a table of people, and each person has a reference to their parents.

Social network analysis (SNA) is a body of methods that deals with relational data. SNA, also called network analysis, and the related fields of network science and graph theory, is a body of methods used to study the relationship between entities. Entities can be individuals, organizations, Twitter users, countries, or a mix of any of the above. Network analysis distinguishes itself from other types of analysis because the focus is on the relationship between entities rather than focusing solely on the properties of the entity itself.

Typically, we want to use network analysis if we have a sense that the relationships or transactions between actors/entities are the most critical part of the story. If you have a sense that the interlocking connection between entities is more important than their individual attributes, or if networked connections between entities forces an endogeneity problem which cannot be resolved by more conventional quantitative regression techniques, then network analysis may be a good approach for you.

Although network analysis as is presented today is sometimes synonymous with computational social science or "big data" methods, network analysis has a long history within sociology and political science. Some of the more innovative uses of network analysis focus on uses that work with archival and historical data.

In a classic article, Padgett and Ansell discuss the rise of the Medici family in 14th-century Florence, Italy. While many families in Florence were attempting to accumulate power and influence within this Renaissance state, the Medici family was able to do so in such a way that outpaced all others. Padgett and Ansell illustrate how they were able to do this using network methods. They used a method called blockmodeling to simplify multiple network relationships and positions to get at an underlying network structure. Padgett and Ansell found that the Medicis were able to consolidate power by positioning themselves as a critical

broker in marriage and economic (trade, partnership, and real estate) networks within the larger group of elite Florentine families.

It illustrates a highly complicated network based on data which are over six centuries old. The **actor or entity** here is the elite family. Second, there is not a single type of relationship in this network – this is a **multiplex** network, meaning there are multiple types of relationships which exist between entities. Relationships are both **undirected** and **directed**, which means they are not all mutual. More on that below. Furthermore, what this means is that the **medium** of what travels across the network tie is different for each network. It is a combination of trust, cooperation, and resources. Third, the data are archival and based upon historical research. The network must be constructed and operationalized explicitly, rather than something like a Twitter retweet, which tends to be accepted somewhat uncritically.

Compare that to this classic network from Adamic and Glance on the political blogosphere circa 2005. This may have been the first of many articles which illustrated the growing political polarization in US political life. The actor here is a political blog. The relationship here is the hyperlink between one blog and another. Because links by their nature go from one blog to another, this is a directed network. The links are not necessarily mutual.

In this network visualization, **colors** and **size** take on a particular type of meaning. Red actors signify conservative blogs, while blue actors signify liberal ones. Furtermore, red lines represent conservative-to-conservative links, and blue liberal-to-liberal. Orange lines, however, represent cross-ideological relationships. Lastly, size indicates how many links are coming into a particular blog. Those colors and sizes indicate something about the actors themselves; they are **attributes** of the actors.

In the former example, we used network analysis to focus on the *centrality of a particular actor* – the Medici family. However, in the latter example, we used network analysis to focus on the *structure of the network as a whole* – namely how between-ideology linkage is much less common than cross-ideology linkage. Both of these could not be achieved by looking at each entity on its own or entities as an aggregate.

The theoretical groundings of network analysis are scattered around various disciplines of social science, but Emirbayer argues most forcefully for its necessary in social science research. Emirbayer poses *relational* or *transactional* analysis as more ontologically sound than *variable-based* or *substantialist* analysis in the social sciences. The focus should be on the interaction between entities, rather than their properties. Individuals may be said to contain attributes (e.g. gender, race, sexual orientation), but a transactional view would indicate that those attributes are all relational (e.g. Desmond and Emirbayer's discussion of racial domination).

Today, we will be covering the basic and intermediate topics in network analysis. We will cover basic network terminology and concepts, network visualization, and descriptive metrics which describe whole networks, individual nodes, and subgroups. We will also cover statistical inference on determinants of network structure.

To work with SNA in R, we're going to use two packages:

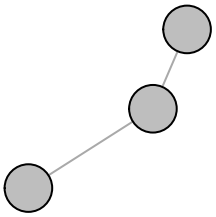
```
library(igraph)
library(repr) #for resizing plots
```

Let's get to defining some terms before we go on, so that we're on the same page. So far I've been using the terms entities or actor to talk about individuals in network analysis. From now on, I will more often use the term node to discuss the entity who is part of the relationship. The terms actor or vertex are synonyms for this.

The connection between two nodes is called an edge (or arc, link, or relation). A network with two nodes and a single edge is called a dyad. A network with three nodes, with any type of configuration of edges between them, is called a triad.

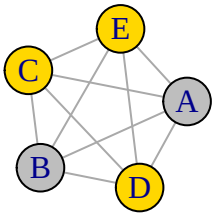
```
## a dyad
## using lgl layout so that the dyad lays flat
plot(graph_from_literal(A-B), vertex.color = "gray",
     vertex.label = NA, vertex.size = 60,
     layout=layout_with_lgl)
```

```
## a triad
plot(graph_from_literal(A-B-C), vertex.color = "gray",
     vertex.label = NA, vertex.size = 60,
     layout=layout_nicely)
```



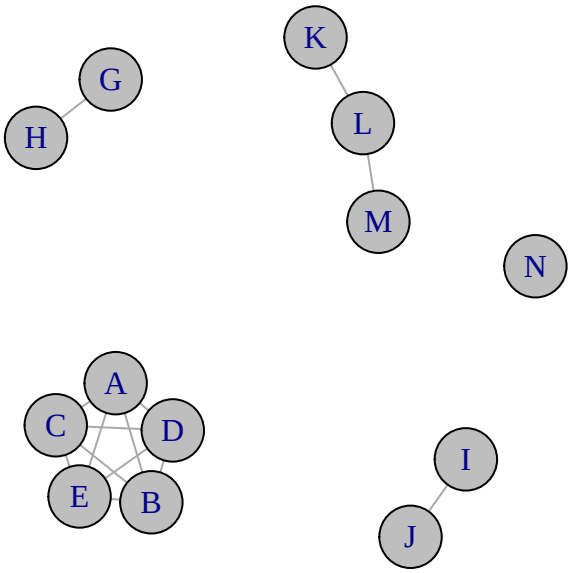
You'll see that we're using a funny little function to draw these networks above called `graph_from_literal`. This lets me literally draw some basic networks using a rudimentary simple syntax. What I want to draw attention to is the name of the function, namely the first word: **graph**. A **graph** is another name for a network and is a term much more common on computer science. A **subgraph** is any subset of a graph. In the network below, the nodes B, C, and D (highlighted in gold) form a subgraph of the larger graph.

```
net <- graph_from_literal(A:B:C:D:E - A:B:C:D:E)
V(net)$color <- c('gray', 'gray', 'gold', 'gold', 'gold')
plot(net, vertex.size = 60)
```



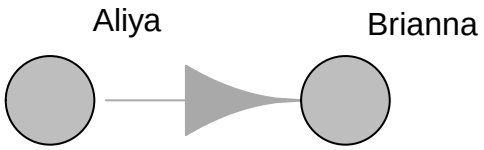
A **component** is a subgraph which is connected together. In the plot below, nodes A through E are a component. Nodes G-H, I-J, K-L-M, and N are components. The largest component is called the **major component** while the others are called **minor components**. N is a special kind of component which is by itself and thus called an **isolate**.

```
options(repr.plot.width = 8, repr.plot.height = 4)
net <- graph_from_literal(A:B:C:D:E - A:B:C:D:E,
                          G-H, I-J, K-L-M, N)
plot(net, vertex.color = 'gray', vertex.size = 25)
```



The edges of a network tend to have a set of common properties. Let's begin with properties of edges. As noted in the Medici example, edges in a network can be either **directed** or **undirected**. The most readily available example we can draw on is from social media – on Facebook, your friendships are mutual. This is an undirected network. Aliya is friends with Brianna and vice versa. On Twitter, however, follower networks are asymmetrical. Aliya may follow Brianna, but Brianna doesn't follow Aliya. This is a directed network. Directed networks are denoted with an arrow in a visualization.

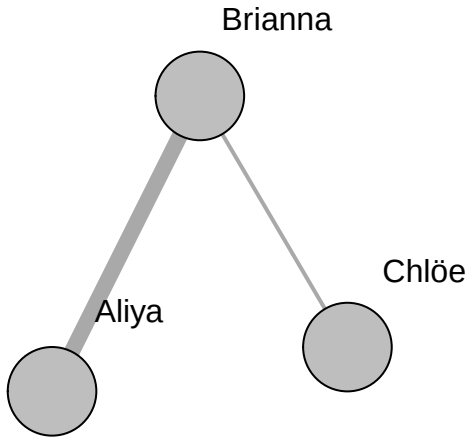
```
# shrink plot size
options(repr.plot.width = 4, repr.plot.height = 3)

net <- graph_from_literal(Aliya --+ Brianna)
plot(net,
     vertex.color = "gray",
     vertex.size = 60,
     vertex.label.color = "black",
     vertex.label.family = "Helvetica",
     vertex.label.dist = 10,
     edge.arrow.size = 3,
     layout=layout_with_lgl)
```



Edges can be **weighted** (valued) or **unweighted** (unvalued), which means they can denote some kind of attribute, such as strength or distance. A Facebook friend in which there's a lot of interaction a good deal of interaction may have a higher weight than one without. Visually, this can be represented in a number of ways. Below, it's denoted with edge width. We'll get into that more in the next module.

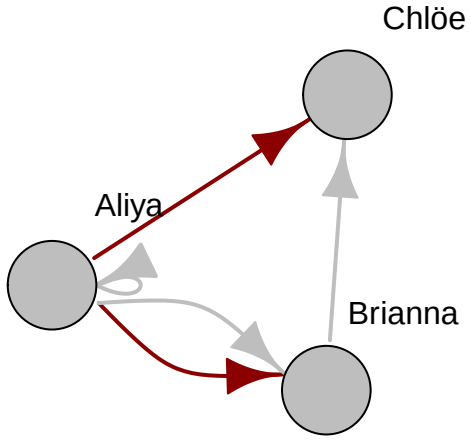
```
net <- graph_from_literal(Aliya - Brianna - Chlöe)
E(net)$width <- c(8, 2)
plot(net,
     vertex.color = "gray",
     vertex.size = 60,
     vertex.label.color = "black",
     vertex.label.family = "Helvetica",
```

```
     vertex.label.dist = 10)
```



Networks can also be **multiplex**, meaning they can denote multiple relationships. Again, this was the case with the Medici example. Furthermore, nodes can link to themselves, what is called a **self-loop**. Think about retweets – people on Twitter can retweet others as well as retweeting themselves. In the graph below, imagine that retweets are the gray network and @replies are dark red network. Aliya is retweeting and replying to Brianna and Chlöe. Aliya also retweets herself. Brianna retweets Chlöe.

```
net <- graph_from_literal(Aliya -+ Brianna -+ Chlöe,
                          Aliya -+ Brianna, Aliya -+ Chlöe,
                          Aliya-+Aliya, simplify = FALSE)
plot(net,
     edge.color = c("dark red", "grey", "grey", "dark red", "grey"),
     vertex.color = "gray",
     vertex.size = 60,
     edge.width = 2,
     edge.arrow.size = 1.5,
     vertex.label.color = "black",
     vertex.label.family = "Helvetica",
     vertex.label.dist = 10)
```

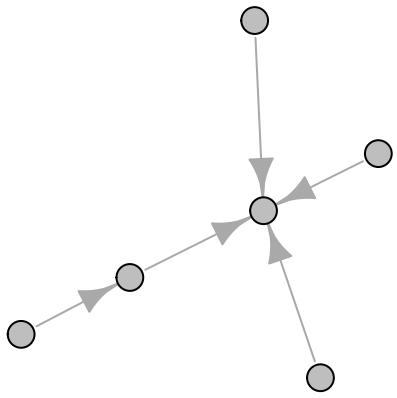


Lastly, networks can change over time. They can be either static or dynamic. We won't really touch on these much in this workshop, so we'll avoid the plotting.

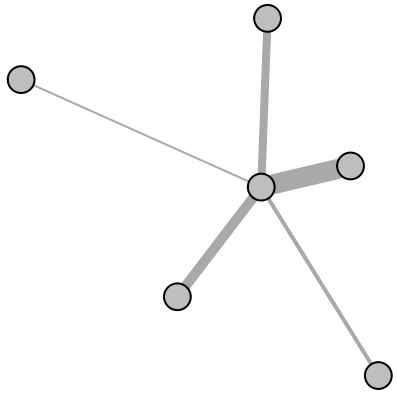
Assignment 1: Basic Understanding

1a) The following code generates two random networks (called a preferential attachment network). Based on these visualization, can you determine whether the networks are directed or undirected? Weighted or unweighted? Singular or multiplex?

```
net1 <- sample_pa(6)
plot(net1, vertex.label = NA, vertex.color = "gray")
```



```
net2 <- sample_pa(6, directed = FALSE)
E(net2)$weight <- c(10,4,1,2,5)
E(net2)$width <- E(net2)$weight
plot(net2, vertex.label = NA, vertex.color = "gray")
```



So far we've only discussed networks in which there's a node of a particular type. However, in political and social networks we often observe networks of more than one type of entity. The simplest advancement of this is the **two-mode network** or **bipartite network**, that is, a network where there's two types of nodes.

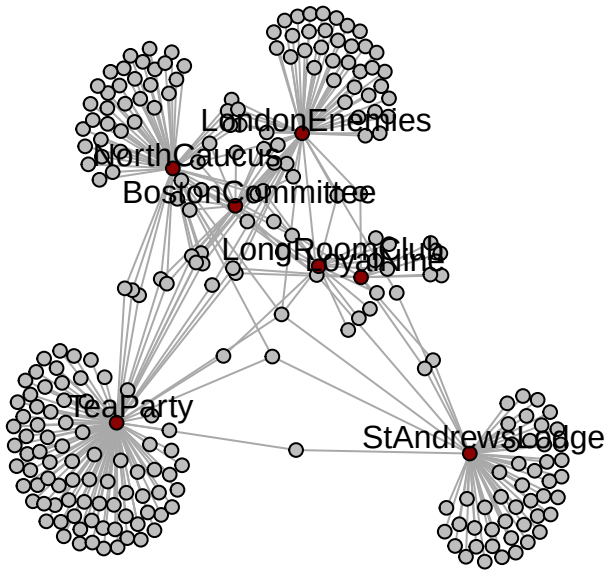
Two-mode network analysis is most common in the case of individuals and groups, or individuals and events. Take this example from Kieran Healy and using metadata to identify Paul Revere. The data here is a matrix which has on its rows individuals and on its columns whether or not the person was a member of a particular American revolutionary organization.

```
(df<- read.csv("PaulRevereAppD.csv", row.names = 1))
```

The visualization of bipartite graph can be helpful in illustrating common membership in organizations.

```
bipartite_revere <- graph.incidence(df)
plot(bipartite_revere,
     vertex.color = ifelse(V(bipartite_revere)$type, "dark red", "gray"),
     vertex.size = 5,
```

```
     vertex.label = ifelse(V(bipartite_revere)$type, V(bipartite_revere)$name, NA),
     vertex.label.dist = 1,
     vertex.label.family = "Helvetica",
     vertex.label.color = "black")
```



However, a major insight of these types of networks highlighted by Ron Breiger is the two-mode networks have a duality – they can be transformed into one-mode networks (e.g. networks with only one type of node) such that we can highlight the importance of one mode in terms of the other, or vice versa. In terms of the Paul Revere dataset, we can see the importance of the individuals based on their group memberships, or we can see the importance of the groups based on the individual memberships.

The transformation is more or less this: if we have an matrix like the dataset above, if we want to obtain the one-mode network of the first mode, we multiple the matrix with its transpose:

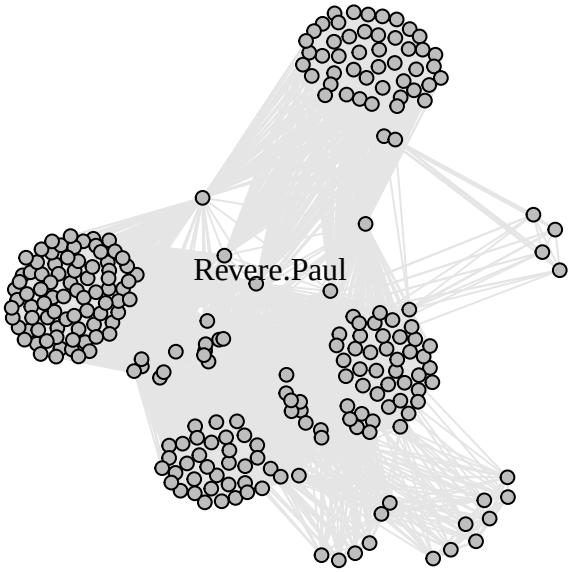
$$G_1 = A(A^T)$$

Conversely, to obtain the one-mode network of the second mode, we multiple the transpose of the matrix with the matrix:

$$G_2 = (A^T)A$$

```
df <- as.matrix(df) ## only matrices can be transposed

person_net <- df %*% t(df) # A*A^T
diag(person_net) <- NA
revere_person_g <- graph.adjacency(person_net,mode="undirected", weighted=NULL, diag=FALSE)

## gets rid of multiplex links and loops
revere_person_g <- simplify(revere_person_g)
plot(revere_person_g,
     vertex.color = "grey",
     vertex.size = 5,
     vertex.label = ifelse(V(revere_person_g)$name == "Revere.Paul", V(revere_person_g)$name, ""),
     vertex.label.dist = 1,
```

```
     edge.color = "grey90",
     vertex.label.color = "black")
```



Paul Revere himself is highlighted to illustrate his importance to the connectivity of this network. While we lose some information in terms of group membership, we gain a good deal in highlighting the importance of relationships in the mode we may be more interested in.

1b) The syntax for creating undirected graphs with `graph_from_literal` *is* `A - B`*. Create an undirected graph in which Amsterdam is connected to Rotterdam and Eindhoven, and Rotterdam is connected to Eindhoven.*

1c) Generate and plot the one-mode network for the groups by filling out the blanks below. What do you notice?

```
revere.group.net <- ____ %*% ____
diag(revere.group.net) <- NA
revere.group.g <- graph.adjacency(____,
                                  weighted = TRUE,
                                  mode="undirected",
                                  diag=FALSE)

plot(revere.group.g,
     vertex.color = "grey",
     vertex.size = 5,
     vertex.label.dist = 1,
     edge.color = "grey",
     edge.width = E(revere.group.g)$weight,
     vertex.label.color = "black")
```

We turn our attention to the various methods of gathering network data. Our minds may go automatically to web scraping or various APIs for network data. But there are many different methods of gathering network data. The first is **digital** – accessing network data through web scraping (such as Adamic and Glance's hyperlink network), the Facebook Graph API (which has graph in the name), and the Twitter API. These data appear to be "born networked" – it doesn't take much of a stretch of the imagination to ask how they are relational. However, one should definitely ask what the nature of that network relation is. Boyd, Golder, and Lotan, for instance, ask about the different conversational aspects of retweets and what that relationship means to different people.

The next most popular which we tend to see is **survey-based**, which ask about networks of individuals

with whom individuals correspond or interact. Much of the research in the past 40 years on networks has been based on surveys which ask about core discussion networks – close networks of people with whom we discuss important topics. The General Social Survey in the US has asked about this for years and has been a subject of contention.

Thirdly, network ties can be inferred via qualitative methods, namely, **interviews and ethnographies**. These methods attempt to gather information on social ties by interacting and/or observing people. A study by Matt Desmond discusses "disposible ties" – intensely strong, yet temporary ties amongst the urban poor. Ethnographical methods often discuss these ties in network analysis language without formalizing the relations between individuals.

Lastly, **archival** methods draw networks from historical and archival sources. We have already seen an example about with the Medici network. There are many examples from within historical sociology, including Bearman et al.'s methodological article on historical casing from first-person narratives and Mohr and Duquenne's exploration of the turn-of-the-century language around the deserving and undeserving poor.

Before moving on, I want to say a word about **explicit vs. implicit networks**. There are some things which seem to be explicitly referred to as networks in the above methods – Twitter retweets, survey name generators. However, there's a good deal of data which has an implicit network structure which has to be specified by the researcher. Examples of these include textual networks – which is an alternative way of performing text analysis – hashtag cooccurrence analysis, and legislative cosponsorship data. While these do not appear to be data sources where you can apply network data, they turn out to be places where thinking about things in a relational manner ends up being very helpful.

There's three main ways that networks are represented and can be loaded into R. There are several other properitary formats which I will not discuss here. These, however, are the main ways which we can read in networks to `igraph` and other R packages.

```
## incidence matrix
head(df)
```

```
##                   StAndrewsLodge LoyalNine NorthCaucus LongRoomClub
## Adams.John                     0         0           1            1
## Adams.Samuel                   0         0           1            1
## Allen.Dr                       0         0           1            0
## Appleton.Nathaniel             0         0           1            0
## Ash.Gilbert                    1         0           0            0
## Austin.Benjamin                0         0           0            0
##                   TeaParty BostonCommittee LondonEnemies
## Adams.John               0               0             0
## Adams.Samuel             0               1             1
## Allen.Dr                 0               0             0
## Appleton.Nathaniel       0               1             0
## Ash.Gilbert              0               0             0
## Austin.Benjamin          0               0             1
```

Incidence matrix - The incidence matrix makes the most sense as a representation for two-mode networks, since it is matching one class of entities to another. In the Paul Revere example, the people are on the rows and the organizations are on the columns. If the network is unweighted, then for all nodes which have a connection, the corresponding cell is 1. For all other cells, the value is 0. If it network is weighted, then instead of 1, the value is the weight.

```
## adjacency matrix
revere_person_df <- data.frame(as.matrix(as_adjacency_matrix(revere_person_g)))

revere_person_df[1:5,1:5] #try out head(revere_person_df)
```

```
##                   Adams.John Adams.Samuel Allen.Dr Appleton.Nathaniel
```

```
## Adams.John                  0            1        1                  1
## Adams.Samuel                1            0        1                  1
## Allen.Dr                    1            1        0                  1
## Appleton.Nathaniel          1            1        1                  0
## Ash.Gilbert                 0            0        0                  0
##                     Ash.Gilbert
## Adams.John                    0
## Adams.Samuel                  0
## Allen.Dr                      0
## Appleton.Nathaniel            0
## Ash.Gilbert                   0
```

Adjacency matrix - An adjacency matrix operates on effectively the same principle as the incidence matrix for one-mode networks. The adjacency matrix is symmetric across the diagonal for undirected networks but not for directed networks. The code above converts it to a data frame for ease of presentation.

```
## edgelist
head(revere_person_el <- data.frame(as_edgelist(revere_person_g)))
```

```
##           X1                 X2
## 1 Adams.John       Adams.Samuel
## 2 Adams.John           Allen.Dr
## 3 Adams.John Appleton.Nathaniel
## 4 Adams.John       Ballard.John
## 5 Adams.John   Barber.Nathaniel
## 6 Adams.John         Bass.Henry
```

Edgelist - This is the simplest data format. It is a simple list of edges in the graph. Weights will often be denoted by a third column. Again, the code above converts the edgelist to a data frame for ease of use.

`igraph` (documentation) is a robust and fast network package developed by Gabor Csardi. We are going to deal with for the majority of this workshop, although there are a number of supplemental packages which are great for plotting and doing inference. We'll introduce them as needed. The good news is that `igraph` plays nicely with all the other network packages.

`igraph` has a class of method which can create an network object from many different formats. We've already seen one in `graph_from_literal`. We also have the following:

- `graph_from_edgelist`
- `graph_from_adjacency_matrix`
- `graph_from_incidence_matrix`
- `graph_from_data_frame` - creates a graph from an R data frame type

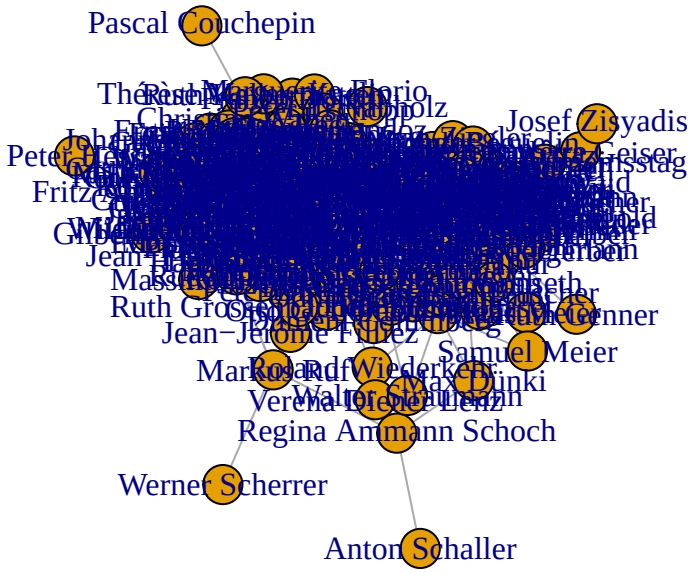
1d) Consider a data set of tweets with information on retweets and @mentions. How do these relationships differ from each other? How are they the same?

1e) Let's load up a new dataset – the cosponsorship network for Conseil National of the Swiss Parliament, from 1995-1999. This data is part of a larger database of cosponsorship data created by Françoise Briatte. This data has been converted from gexf format, and it has been converted from a directed to an undirected graph. Use `head()` to answer the question what kind of data source this is? What kind of data format is this?

```
data_conseil = read.csv('net_ch_cn1995-1999.csv', header = TRUE, as.is = TRUE)
```

1f) Complete the following code to create the network which is appropriate to this data format.

```
plot(graph_from_edgelist(as.matrix(data_conseil), directed = FALSE))
```



Other type of Relations: Spatial Sata

R has a huge and growing number of spatial data packages. We recommend taking a quick browse on R's main website to see the spatial packages available.

In this tutorial we will use the following packages:

- **ggmap**: extends the plotting package `ggplot2` for maps
- **rgdal**: R's interface to the popular C/C++ spatial data processing library `gdal`
- **rgeos**: R's interface to the powerful vector processing library `geos`
- **maptools**: provides various mapping functions
- **dplyr** and **tidyr**: fast and concise data manipulation packages
- **tmap**: a new packages for rapidly creating beautiful maps

```
x <- c("ggmap", "rgdal", "rgeos", "maptools", "tidyverse", "tmap")
#install.packages(x) # warning: uncommenting this may take a number of minutes
lapply(x, library, character.only = TRUE) # load the required packages
```

The first file we are going to load into R Studio is the `london_sport` shapefile. The data can be downloaded from Robin Love Lace's Github page. It is worth looking at this input dataset in your file browser before opening it in R. You will notice that there are several files named "london_sport"", all with different file extensions. This is because a shapefile is actually made up of a number of different files, such as .prj, .dbf and .shp. You could also try opening the file"london_sport.shp" file in a conventional GIS such as QGIS to see what a shapefile contains. Once you think you understand the input data, it's time to open it in R. There are a number of ways to do this, the most commonly used and versatile of which is readOGR. This function, from the `rgdal` package, automatically extracts the information regarding the data. `rgdal` is R's interface to the "Geospatial Abstraction Library (GDAL)" which is used by other open source GIS packages such as QGIS and enables R to handle a broader range of spatial data formats.

```
lnd <- readOGR(dsn = "data", layer = "london_sport")
```

```
## OGR data source with driver: ESRI Shapefile
## Source: "/Users/velden/surfdrive/Shared/Teaching/Data Science_Visualization and Analytics/Assignments
## with 33 features
```

```
## It has 4 fields
## Integer64 fields read as strings:  Pop_2001
```

`readOGR` is a function which accepts two arguments: `dsn` which stands for "data source name" and specifies the directory in which the file is stored, and `layer` which specifies the file name (note that there is no need to include the file extention .shp). The arguments are separated by a comma and the order in which they are specified is important. You do not have to explicitly type `dsn =` or `layer =` as R knows which order they appear, so `readOGR("data", "london_sport")` would work just as well. For clarity, it is good practice to include argument names when learning new functions so we will continue to do so. The file we assigned to the `lnd` object contains the population of London Boroughs in 2001 and the percentage of the population participating in sporting activities. This data originates from the Active People Survey. The boundary data is from the Ordnance Survey. For information about how to load different types of spatial data, see the help documentation for `readOGR`. This can be accessed by typing `?readOGR`. For another worked example, in which a GPS trace is loaded, please see Cheshire and Lovelace (2014).

Spatial objects like the `lnd` object are made up of a number of different slots, the key slots being `@data` (non geographic attribute data) and `@polygons` (or `@lines` for line data). The data slot can be thought of as an attribute table and the geometry slot is the polygons that make up the physcial boundaries. Specific slots are accessed using the `@` symbol. Let's now analyse the sport object with some basic commands:

```
head(lnd@data, n = 2)
mean(lnd$Partic_Per)
```

Now we have seen something of the structure of spatial objects in R, let us look at plotting them. Note, that plots use the geometry data, contained primarily in the `@polygons` slot.

```
plot(lnd)
```

`plot` is one of the most useful functions in R, as it changes its behaviour depending on the input data (this is called *polymorphism* by computer scientists). Inputting another object such as `plot(lnd@data)` will generate an entirely different type of plot. Thus R is intelligent at guessing what you want to do with the data you provide it with.

R has powerful subsetting capabilities that can be accessed very concisely using square brackets,as shown in the following example:

```
# select rows of lnd@data where sports participation is less than 15
lnd@data[lnd$Partic_Per < 15, ] # we don't use the tidyverse verb select,
```

```
##    ons_label           name Partic_Per Pop_2001
## 17      00AQ         Harrow       14.8   206822
## 21      00BB         Newham       13.1   243884
## 32      00AA City of London        9.1     7181
```

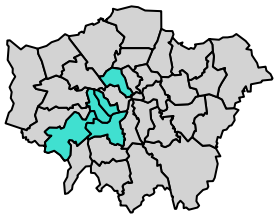
```
# because it doesn't work well with polygons
```

The above line of code asked R to select only the rows from the `lnd` object, where sports participation is lower than 15, in this case rows 17, 21 and 32, which are Harrow, Newham and the city centre respectively. The square brackets work as follows: anything before the comma refers to the rows that will be selected, anything after the comma refers to the number of columns that should be returned. For example if the data frame had 1000 columns and you were only interested in the first two columns you could specify `1:2` after the

Figure 1: Zones in London whose centroid lie within 10 km of the geographic centroid of the City of London. Note the distinction between zones which only touch or 'intersect' with the buffer (light blue) and zones whose centroid is within the buffer (darker blue).

comma. The `:` symbol simply means "to", i.e. columns 1 to 2. Try experimenting with the square brackets notation (e.g. guess the result of `lnd@data[1:2, 1:3]` and test it). So far we have been interrogating only the attribute data slot (`@data`) of the `lnd` object, but the square brackets can also be used to subset spatial objects, i.e. the geometry slot. Using the same logic as before try to plot a subset of zones with high sports participation. Try to replicate the following graph:

```
plot(lnd, col = "lightgrey") # plot the london_sport object
sel <- lnd$Partic_Per > 25
plot(lnd[ sel, ], col = "turquoise", add = TRUE)
```



You have just interrogated and visualised a spatial object: where are areas with high levels of sports participation in London? The map tells us. Do not worry for now about the intricacies of how this was achieved.

As a bonus stage, select and plot only zones that are close to the centre of London. Programming encourages rigorous thinking and it helps to define the problem more specifically:

Assingment 2: Working with Maps

2a) Select all zones whose geographic centroid lies within 10 km of the geographic centroid of inner London.[1]

[1] To see how this map was created, see the code in README.Rmd. This may be loaded by typing `file.edit("README.Rmd")` or online at the underlineREADME page.

The code below should help understand the way spatial data work in R.

```
# Find the centre of the london area
easting_lnd <- coordinates(gCentroid(lnd))[[1]]
northing_lnd <- coordinates(gCentroid(lnd))[[2]]
# arguments to test whether or not a coordinate is east or north of the centre
east <- sapply(coordinates(lnd)[,1], function(x) x > easting_lnd)
north <- sapply(coordinates(lnd)[,2], function(x) x > northing_lnd)
# test if the coordinate is east and north of the centre
lnd$quadrant <- "unknown" # prevent NAs in result
lnd$quadrant[east & north] <- "northeast"
```

2b) Based on the the above code as reference try and find the remaining 3 quadrants and colour them. Hint - you can use the **`llgridlines`** *function in order to overlay the long-lat lines. Try to desolve the quadrants so the map is left with only 4 polygons.*

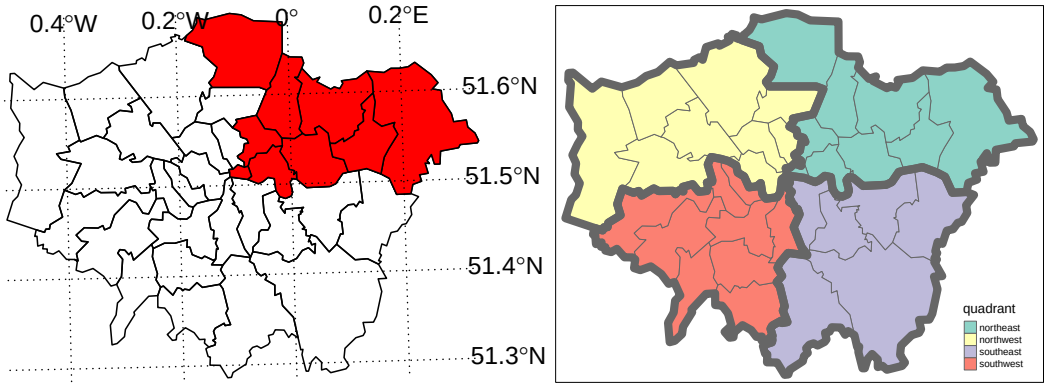


Figure 2: The 4 quadrants of London and dissolved borders. Challenge: recreate a plot that looks like this.

2c) As an alternative to maps, we can consider bins *to visually present spatial data, using the package* `statebins` *(documentation) developed by Bob Rudis. We will use it to look at US state-level election results, (load the* `socviz` *package to access the data set* `election`*.). Statebins is similar to* `ggplot` *but has a slightly different syntax from the one we're used to. It needs several arguments including the basic data frame (the* `state_data()` *argument), a vector of state names (*`state_col`*), and the value being shown (*`value_col`*). In addition, we can optionally tell it the color palette we want to use and the color of the text to label the state boxes. For a continuous variable, like the percentage of votes for Trump or Clinton (*`pct_trump` *and* `pct_clinton`*), we can use* `statebins_continuous()`*. Try to replicate the following plots:*

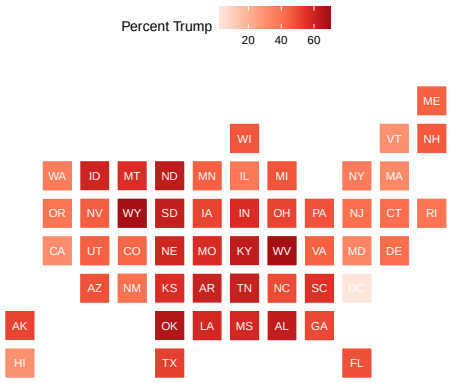


Figure 3: Statebins of the election results. DC is omitted from the Clinton map to prevent the scale becoming unbalanced. Challenge: recreate a plot that looks like this.

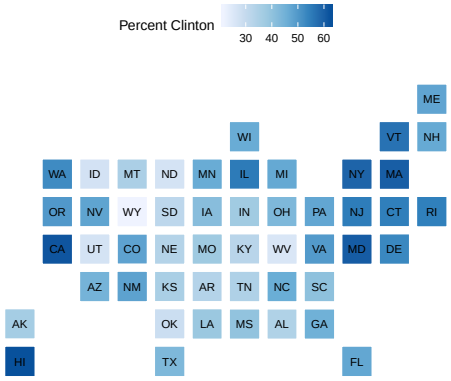


Figure 4: Statebins of the election results. DC is omitted from the Clinton map to prevent the scale becoming unbalanced. Challenge: recreate a plot that looks like this.

Good Luck

The deadline for the assignment is **December 5th, 10am**!

References

Healy, K. (2018). *Data visualization: a practical introduction*. Princeton University Press.

Alex Hana's Github

Alex Hana's Github

Data Science: Visualisations in R - Assignment 6

Github of CCS Amsterdam

dr. Mariken A.C.G. van der Velden

November 29, 2019

Instructions for the tutorial

Read the *entire* document that describes the assignment for this tutorial. Follow the indicated steps. Each assignment in the tutorial contains several exercises, divided into numbers and letters (1a, 1b, 1c, 2a, 2b, etc.).

Write down your answers with **the same number-letter combination** in a separate document (e.g., Google Doc), and submit this document to Canvas in a PDF format.

Note: this is the first time that this course has ever been taught. Because of rapidly changing techniques, the course tries to keep up/ reflect those changes. As a result, the assignments may contain errors or ambiguities. If in doubt, ask for clarification during the hall lectures or tutorials. All feedback on the assignments is greatly appreciated!

Text Analysis with Quanteda

In this tutorial you will learn how to perform text analysis using the quanteda package. The quanteda package is an extensive text analysis suite for R. It covers everything you need to perform a variety of automatic text analysis techniques, and features clear and extensive documentation. Here we'll focus on the main preparatory steps for text analysis, and on learning how to browse the quanteda documentation. The documentation for each function can also be found here.

For a more detailed explanation of the steps discussed here, you can read the paper Text Analysis in R (Welbers, van Atteveldt & Benoit, 2017).

```
library(quanteda)
library(tidyverse)
```

Step 1: Importing text and creating a quanteda corpus: The first step is getting text into R in a proper format. stored in a variety of formats, from plain text and CSV files to HTML and PDF, and with different 'encodings'. There are various packages for reading these file formats, and there is also the convenient readtext that is specialized for reading texts from a variety of formats. For this tutorial, we will be importing text from a csv. For convenience, we're using a csv that's available online, but the process is the same for a csv file on your own computer. The data consists of the State of the Union speeches of US presidents, with each document (i.e. row in the csv) being a paragraph. The data will be imported as a `data.frame`.

```
library(readr)
url <- 'https://bit.ly/2QoqUQS'
(d <- read_csv(url))
```

```
## # A tibble: 23,469 x 5
##    paragraph date       President     Party text
##        <dbl> <date>     <chr>         <chr> <chr>
##  1         1 1790-01-08 George Washi~ Other I embrace with great satisfact~
##  2         2 1790-01-08 George Washi~ Other In resuming your consultations~
##  3         3 1790-01-08 George Washi~ Other Among the many interesting obj~
##  4         4 1790-01-08 George Washi~ Other A free people ought not only t~
##  5         5 1790-01-08 George Washi~ Other The proper establishment of th~
```

```
##  6         6 1790-01-08 George Washi~ Other There was reason to hope that ~
##  7         7 1790-01-08 George Washi~ Other The interests of the United St~
##  8         8 1790-01-08 George Washi~ Other Various considerations also re~
##  9         9 1790-01-08 George Washi~ Other Uniformity in the currency, we~
## 10        10 1790-01-08 George Washi~ Other The advancement of agriculture~
## # ... with 23,459 more rows
```

We can now create a quanteda corpus with the `corpus()` function. If you want to learn more about this function, recall that you can use the question mark to look at the documentation.

```
?corpus
```

Here you see that for a data.frame, we need to specify which column contains the text field. Also, the text column must be a character vector.

```
(corp <- corpus(d, text_field = 'text'))  ## create the corpus
```

```
## Corpus consisting of 23,469 documents and 4 docvars.
```

Step 2: Creating the DTM (or DFM): any text analysis techniques only use the frequencies of words in documents. This is also called the bag-of-words assumption, because texts are then treated as bags of individual words. Despite ignoring much relevant information in the order of words and syntax, this approach has proven to be very powerfull and efficient.

The standard format for representing a bag-of-words is as a `document-term matrix` (DTM). This is a matrix in which rows are documents, columns are terms, and cells indicate how often each term occured in each document. We'll first create a small example DTM from a few lines of text. Here we use quanteda's `dfm()` function, which stands for `document-feature matrix` (DFM), which is a more general form of a DTM.

```
text <-  c(d1 = "Cats are awesome!",
           d2 = "We need more cats!",
           d3 = "This is a soliloquy about a cat.")

(dtm <- dfm(text, tolower=F))
```

```
## Document-feature matrix of: 3 documents, 15 features (64.4% sparse).
## 3 x 15 sparse Matrix of class "dfm"
##     features
## docs Cats are awesome ! We need more cats This is a soliloquy about cat .
##   d1    1   1       1 1  0    0    0    0    0  0 0         0     0   0 0
##   d2    0   0       0 1  1    1    1    1    0  0 0         0     0   0 0
##   d3    0   0       0 0  0    0    0    0    1  1 2         1     1   1 1
```

Here you see, for instance, that the observation (word) `soliloquy` only occurs in the third document. In this matrix format, we can perform calculations with texts, like analyzing different sentiments of frames regarding cats, or the computing the similarity betwee the third sentence and the first two sentences.

However, directly converting a text to a DTM is a bit crude. Note, for instance, that the words `Cats`, `cats`, and `cat` are given different columns. In this DTM, "Cats" and "awesome" are as different as "Cats" and "cats", but for many types of analysis we would be more interested in the fact that both texts are about felines, and not about the specific word that is used. Also, for performance it can be useful (or even necessary) to use fewer columns, and to ignore less interesting words such as `is` or very rare words such as `soliloquy`.

This can be achieved by using additional *preprocessing* steps. In the next example, we'll again create the DTM, but this time we make all text lowercase, ignore stopwords and punctuation, and perform *stemming*. Simply put, stemming removes some parts at the ends of words to ignore different forms of the same word, such as singular versus plural ("gun" or "gun-s") and different verb forms ("walk","walk-ing","walk-s")

```
(dtm <- dfm(text, tolower=T, remove = stopwords('en'), stem = T, remove_punct=T))
```

```
## Document-feature matrix of: 3 documents, 4 features (50.0% sparse).
## 3 x 4 sparse Matrix of class "dfm"
##     features
## docs cat awesom need soliloquy
##   d1   1      1    0         0
##   d2   1      0    1         0
##   d3   1      0    0         1
```

The `tolower` argument determines whether texts are (TRUE) or aren't (FALSE) converted to lowercase. `stem` determines whether stemming is (TRUE) or isn't (FALSE) used. The remove argument is a bit more tricky. If you look at the documentation for the dfm function (`?dfm`) you'll see that `remove` can be used to give "a pattern of user-supplied features to ignore". In this case, we actually used another function, `stopwords()`, to get a list of english stopwords. You can see for yourself.

```
stopwords('en')
```

```
##   [1] "i"          "me"         "my"         "myself"     "we"
##   [6] "our"        "ours"       "ourselves"  "you"        "your"
##  [11] "yours"      "yourself"   "yourselves" "he"         "him"
##  [16] "his"        "himself"    "she"        "her"        "hers"
##  [21] "herself"    "it"         "its"        "itself"     "they"
##  [26] "them"       "their"      "theirs"     "themselves" "what"
##  [31] "which"      "who"        "whom"       "this"       "that"
##  [36] "these"      "those"      "am"         "is"         "are"
##  [41] "was"        "were"       "be"         "been"       "being"
##  [46] "have"       "has"        "had"        "having"     "do"
##  [51] "does"       "did"        "doing"      "would"      "should"
##  [56] "could"      "ought"      "i'm"        "you're"     "he's"
##  [61] "she's"      "it's"       "we're"      "they're"    "i've"
##  [66] "you've"     "we've"      "they've"    "i'd"        "you'd"
##  [71] "he'd"       "she'd"      "we'd"       "they'd"     "i'll"
##  [76] "you'll"     "he'll"      "she'll"     "we'll"      "they'll"
##  [81] "isn't"      "aren't"     "wasn't"     "weren't"    "hasn't"
##  [86] "haven't"    "hadn't"     "doesn't"    "don't"      "didn't"
##  [91] "won't"      "wouldn't"   "shan't"     "shouldn't"  "can't"
##  [96] "cannot"     "couldn't"   "mustn't"    "let's"      "that's"
## [101] "who's"      "what's"     "here's"     "there's"    "when's"
## [106] "where's"    "why's"      "how's"      "a"          "an"
## [111] "the"        "and"        "but"        "if"         "or"
## [116] "because"    "as"         "until"      "while"      "of"
## [121] "at"         "by"         "for"        "with"       "about"
## [126] "against"    "between"    "into"       "through"    "during"
## [131] "before"     "after"      "above"      "below"      "to"
## [136] "from"       "up"         "down"       "in"         "out"
## [141] "on"         "off"        "over"       "under"      "again"
## [146] "further"    "then"       "once"       "here"       "there"
## [151] "when"       "where"      "why"        "how"        "all"
## [156] "any"        "both"       "each"       "few"        "more"
## [161] "most"       "other"      "some"       "such"       "no"
## [166] "nor"        "not"        "only"       "own"        "same"
## [171] "so"         "than"       "too"        "very"       "will"
```

This list of words is thus passed to the `remove` argument in the `dfm()` to ignore these words. If you are using texts in another language, make sure to specify the language, such as `stopwords('nl')` for Dutch or `stopwords('de')` for German.

There are various alternative preprocessing techniques, including more advanced techniques that are not implemented in quanteda. Whether, when and how to use these techniques is a broad topic that we won't cover today.

For this tutorial, we'll use the State of the Union speeches. We already created the corpus above. We can now pass this corpus to the `dfm()` function and set the preprocessing parameters.

```
(dtm <- dfm(corp, tolower=T, stem=T, remove=stopwords('en'), remove_punct=T))
```

```
## Document-feature matrix of: 23,469 documents, 20,431 features (99.8% sparse).
```

This dtm has 23,469 documents and 20,429 features (i.e. terms), and no longer shows the actual matrix because it simply wouldn't fit. Depending on the type of analysis that you want to conduct, we might not need this many words, or might actually run into computational limitations.

Luckily, many of these 20K features are not that informative. The distribution of term frequencies tends to have a very long tail, with many words occuring only once or a few times in our corpus. For many types of bag-of-words analysis it would not harm to remove these words, and it might actually improve results.

We can use the `dfm_trim` function to remove columns based on criteria specified in the arguments. Here we say that we want to remove all terms for which the frequency (i.e. the sum value of the column in the DTM) is below 10.

```
(dtm  <- dfm_trim(dtm, min_termfreq = 10))
```

```
## Document-feature matrix of: 23,469 documents, 5,293 features (99.4% sparse).
```

Now we have about 5000 features left. See `?dfm_trim` for more options.

In many cases, especially for languages with a richer morphology than English, it can be useful to use tools from computational linguistics to preprocess the data. In particular, *lemmatization* often works better for stemming, and *Part of Speech tagging* can be a great way to select e.g. only the names or verbs in a document.

`udpipe` is an R package that can do many preprocessing steps for a variety of languages including English, French, German and Dutch. If you call it for a language you have not previously used, it will automatically download the language model.

For this example, we will use a very short text, as it can take (very) long to process large amounts of text.

```
small_text <- c("Pelosi says Trump is welcome to testify in impeachment inquiry, if he chooses", "House
small_corpus <- corpus(small_text)
```

Now, let's lemmatize and tag this corpus:

```
library(udpipe)
tokens <- udpipe(texts(small_corpus), "english", parser="none")
tokens %>% as_tibble() %>% select(token_id:xpos)
```

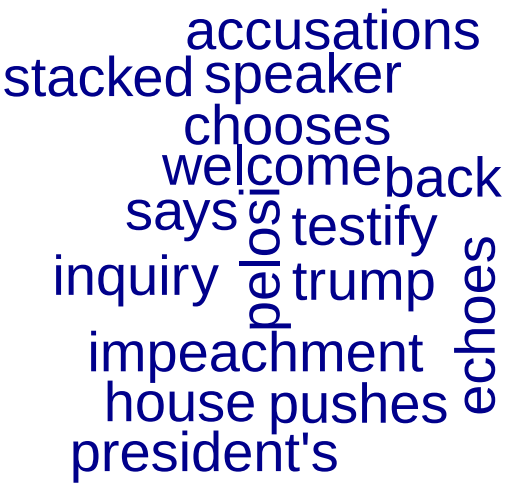
```
## # A tibble: 33 x 5
##    token_id token       lemma       upos  xpos
##    <chr>    <chr>       <chr>       <chr> <chr>
##  1 1        Pelosi      Pelosi      PROPN NNP
##  2 2        says        say         VERB  VBZ
##  3 3        Trump       Trump       PROPN NNP
##  4 4        is          be          AUX   VBZ
##  5 5        welcome     welcome     ADJ   JJ
##  6 6        to          to          PART  TO
##  7 7        testify     testify     VERB  VB
##  8 8        in          in          ADP   IN
##  9 9        impeachment impeachment ADJ   JJ
## 10 10       inquiry     inquiry     NOUN  NN
```

```
## # ... with 23 more rows
```

As you can see, 'is' is lemmatized to 'be', and Pelosi and Trump are both recognized as proper nouns (names).

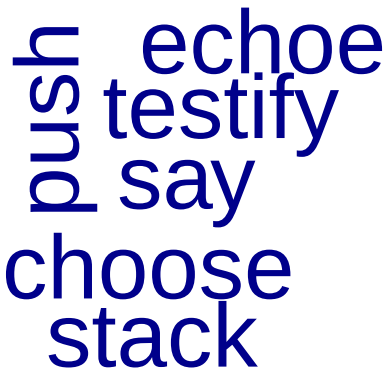
We can create a dfm of all lemmata, assinging to udpipe tokens to the corpus and proceeding as normal:

```
small_corpus$tokens <-  as.tokens(split(tokens$lemma, tokens$doc_id))
d <- dfm(small_corpus, remove=stopwords("english"), remove_punct=T)
textplot_wordcloud(d, min_count = 1)
```



More interesting, however, is to e.g. select only the verbs: Note that here I skip the corpus steps to show how you can also work directly with the texts and tokens:

```
tokens <- udpipe(small_text, "english", parser="none")
d <- tokens %>% filter(upos == "VERB") %>%
  with(split(lemma, doc_id)) %>% as.tokens() %>%
  dfm(remove=stopwords("german")) %>%
textplot_wordcloud(d, min_count=1)
```



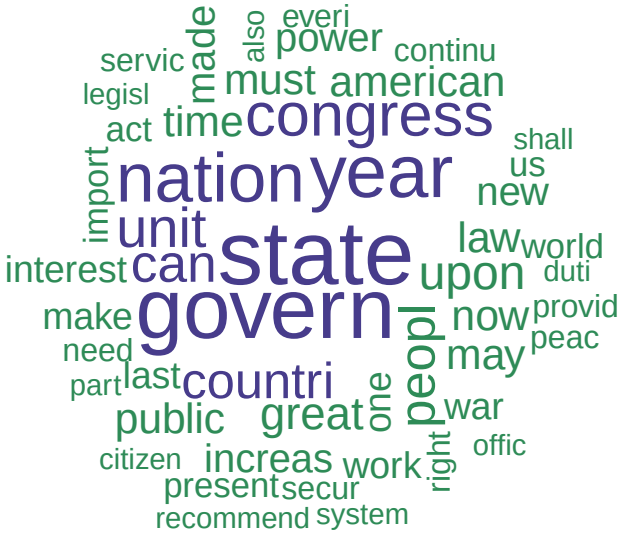
Of course, this is more meaningful if you run it on a larger text.

Step 3: Analysis: Using the dtm we can now employ various techniques.

1. Get most frequent words in corpus.

```
textplot_wordcloud(dtm, max_words = 50)     ## top 50 (most frequent) words
```


```
textplot_wordcloud(dtm, max_words = 50, color = c('seagreen','slateblue4')) ## change colors
```



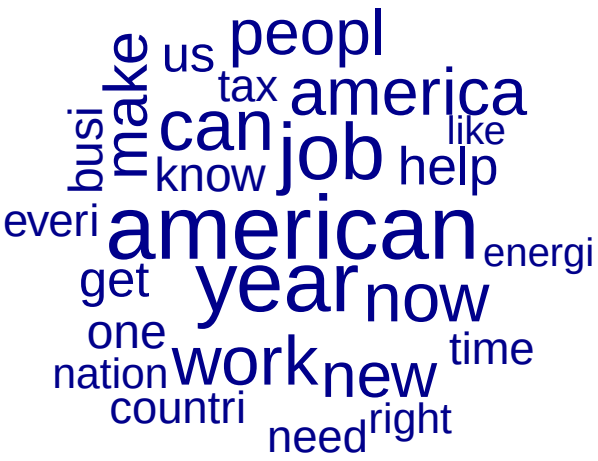

```
textstat_frequency(dtm, n = 10)             ## view the frequencies
```

```
##     feature frequency rank docfreq group
## 1     state      9231    1    5579   all
## 2    govern      8576    2    5549   all
## 3      year      7241    3    4931   all
## 4    nation      6724    4    4843   all
## 5  congress      5685    5    4491   all
## 6      unit      5223    6    3715   all
## 7       can      4727    7    3627   all
## 8   countri      4664    8    3612   all
## 9     peopl      4467    9    3379   all
## 10     upon      4168   10    3004   all
```

You can also inspect a subcorpus. For example, looking only at Obama speeches. To subset the DTM we can use quanteda's `dtm_subset()`, but we can also use the more general R subsetting techniques (as discussed last week). Here we'll use the latter for illustration.

With `docvars(dtm)` we get a data.frame with the document variables. With `docvars(dtm)$President`, we get the character vector with president names. Thus, with `docvars(dtm)$President == 'Barack Obama'` we look for all documents where the president was Obama. To make this more explicit, we store the logical vector, that shows which documents are 'TRUE', as is_obama. We then use this to select these rows from the DTM.

```
is_obama <- docvars(dtm)$President == 'Barack Obama'
obama_dtm <- dtm[is_obama,]
textplot_wordcloud(obama_dtm, max_words = 25)
```



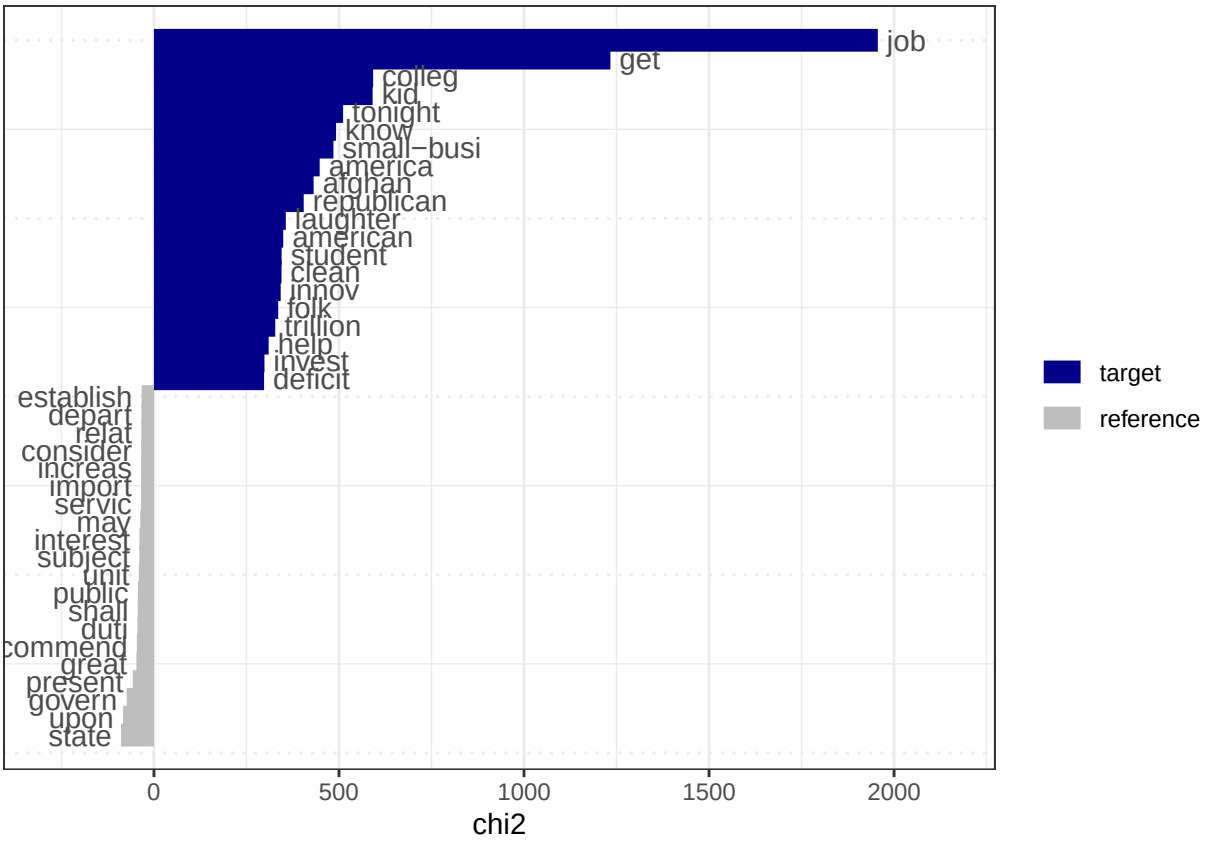
2. Compare word frequencies between two subcorpora. Here we (again) first use a comparison to get the is_obama vector. We then use this in the `textstat_keyness()` function to indicate that we want to compare the Obama documents (where is_obama is TRUE) to all other documents (where is_obama is FALSE).

```
is_obama <- docvars(dtm)$President == 'Barack Obama'
ts <- textstat_keyness(dtm, is_obama)
head(ts, n = 20)
```

```
##       feature      chi2 p n_target n_reference
## 1         job 1956.0965 0      202         628
## 2         get 1233.4504 0      121         355
## 3      colleg  592.3280 0       57         160
## 4         kid  590.8774 0       30          34
## 5     tonight  511.1499 0       82         399
## 6        know  492.1576 0      108         694
## 7  small-busi  485.0449 0       14           3
## 8     america  447.8391 0      169        1644
## 9      afghan  431.5014 0       17          11
## 10 republican  404.6363 0       41         121
## 11   laughter  356.3841 0       28          60
## 12   american  349.3285 0      240        3385
## 13    student  344.9895 0       41         144
## 14      clean  344.6813 0       35         103
## 15      innov  342.4968 0       27          58
## 16       folk  335.7901 0       14          10
## 17   trillion  327.7738 0       16          16
## 18       help  309.9923 0      126        1289
## 19     invest  298.5842 0       72         500
## 20    deficit  297.6741 0       56         315
```

We can visualize these results, stored under the name `ts`, by using the textplot_keyness function

```
textplot_keyness(ts)
```



3. A keyword-in-context listing shows a given keyword in the context of its use. This is a good help for interpreting words from a wordcloud or keyness plot. Since a DTM only knows word frequencies, the `kwic()` function requires the corpus object as input.

```
head(k <- kwic(corp, 'freedom', window = 7))
```

```
##
##  [text126, 62]      without harmony as far as consists with | freedom |
##  [text357, 84]           a wide spread for the blessings of | freedom |
##  [text466, 84] commerce of the United States its legitimate | freedom |
##  [text481, 89]    of cheaper materials and subsistence, the | freedom |
##  [text483, 23]      payment of the public debt whenever the | freedom |
##  [text626, 32]     its progress a force proportioned to its | freedom |
##
##  of sentiment its dignity may be lost
##  and equal laws.
##  . The instructions to our ministers with
##  of labor from taxation with us,
##  and safety of our commerce shall be
##  , and that the union of these
```

The `kwic()` function can also be used to focus an analysis on a specific search term. You can use the output of the kwic function to create a new DTM, in which only the words within the shown window are included in the DTM. With the following code, a DTM is created that only contains words that occur within 10 words from `terror*` (terrorism, terrorist, terror, etc.).

```
terror <- kwic(corp, 'terror*')
terror_corp <- corpus(terror)
terror_dtm <- dfm(terror_corp, tolower=T, remove=stopwords('en'), stem=T, remove_punct=T)
```

Now you can focus an analysis on whether and how Presidents talk about `terror*`.

```
textplot_wordcloud(terror_dtm, max_words = 50)     ## top 50 (most frequent) words
```

4. You can perform a basic dictionary search. Quanteda supports the use of existing dictionaries, for instance for sentiment analysis (but mostly for english dictionaries). An convenient way of using dictionaries is to make a DTM with the columns representing dictionary terms.

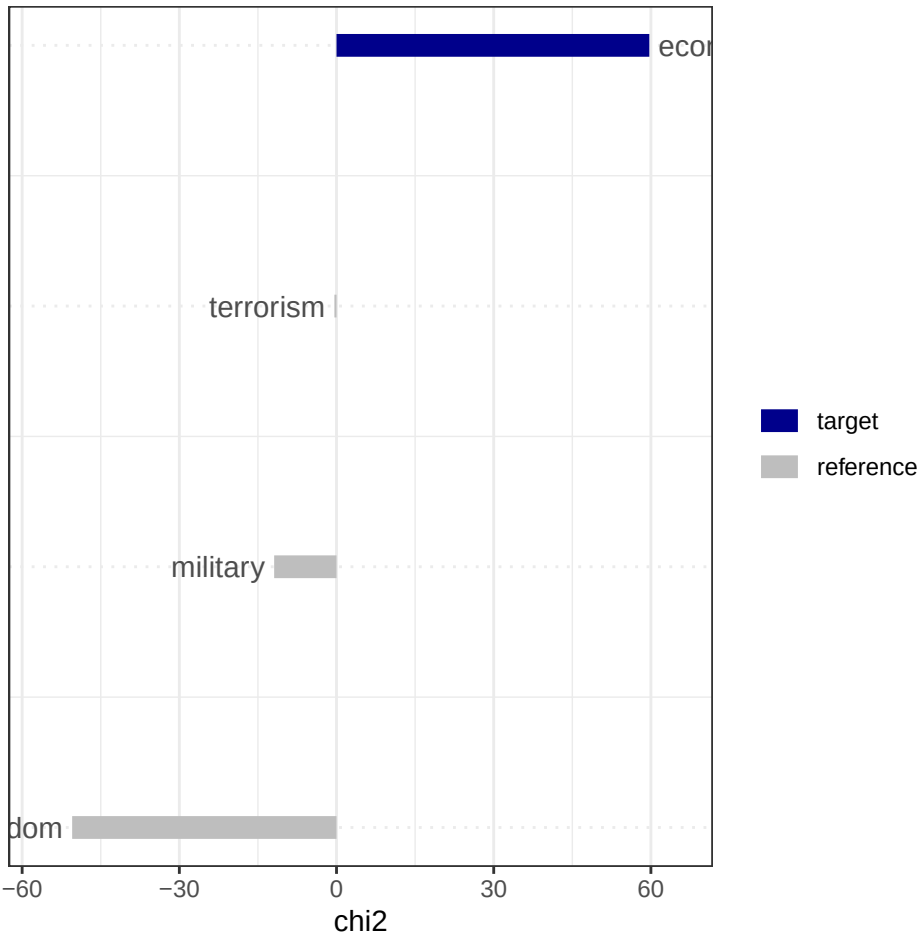
```
dict <- dictionary(list(terrorism = 'terror*',
                        economy = c('econom*', 'tax*', 'job*'),
                        military = c('army','navy','military',
                                     'airforce','soldier'),
                        freedom = c('freedom','liberty')))
(dict_dtm <- dfm_lookup(dtm, dict, exclusive=TRUE))
```

```
## Document-feature matrix of: 23,469 documents, 4 features (95.6% sparse).
```

The "4 features" are the four entries in our dictionary. Now you can perform all the analyses with dictionaries.

```
textplot_wordcloud(dict_dtm)
```


```
tk <- textstat_keyness(dict_dtm, docvars(dict_dtm)$President == 'Barack Obama')
textplot_keyness(tk)
```



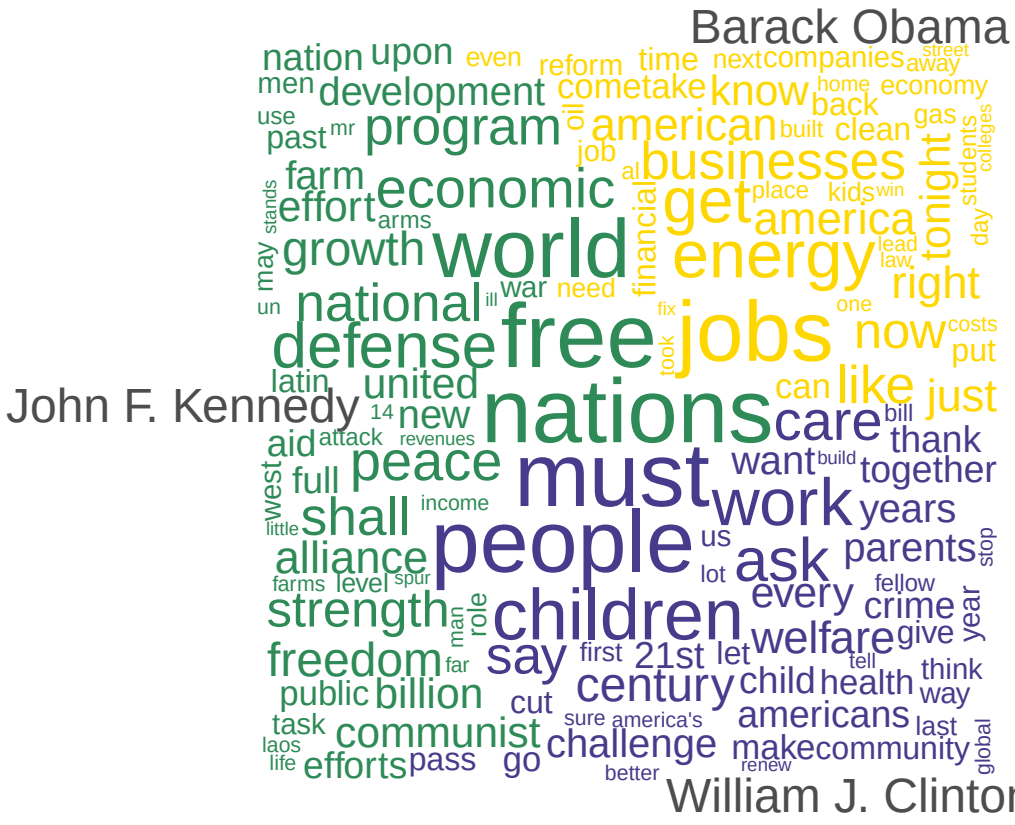
You can also convert the dtm to a data frame to get counts of each concept per document (which you can then match with e.g. survey data).

```
head(df <- convert(dict_dtm, to="data.frame"))
```

```
##   document terrorism economy military freedom
## 1    text1         0       0        0       0
## 2    text2         0       0        0       0
## 3    text3         0       0        0       0
## 4    text4         0       0        0       0
## 5    text5         0       1        1       0
## 6    text6         0       0        0       0
```

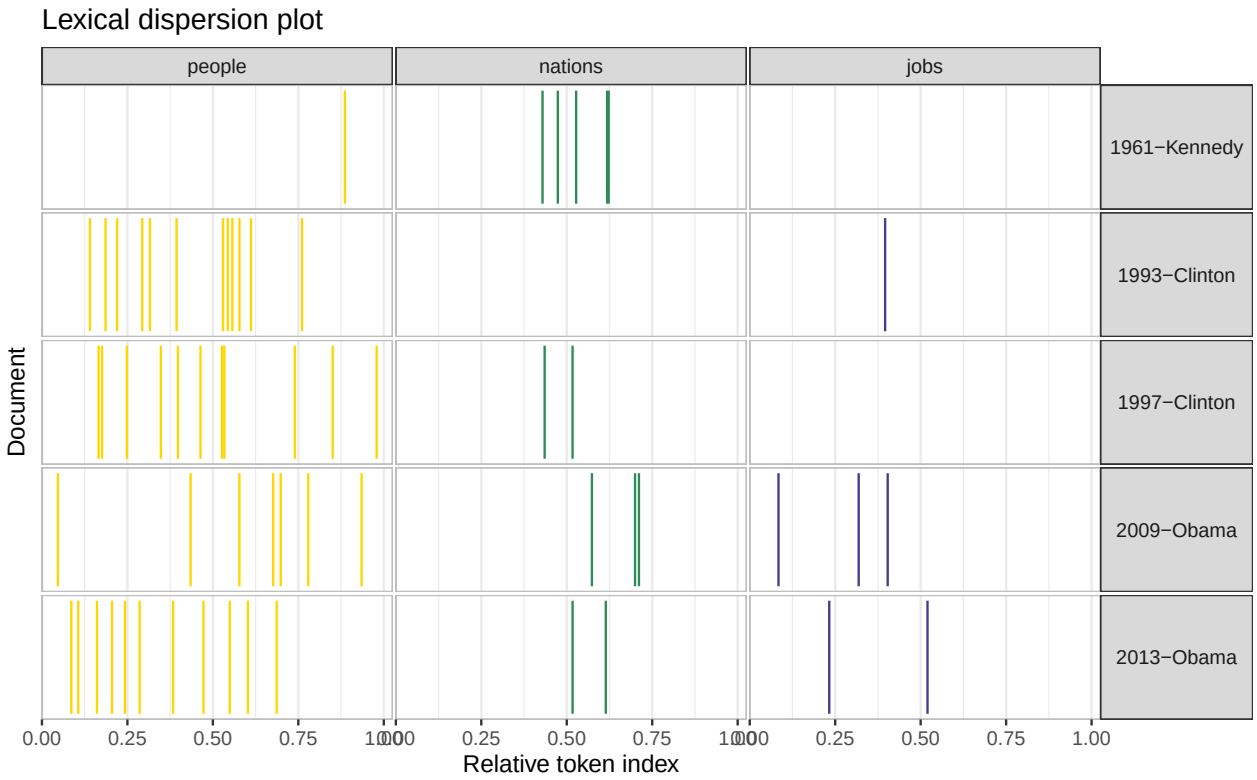
Assignment: Textual Visualization

1a) Use the State of the Union Speeches that we imported during the explanation above. Compare the speeches of Barack Obama to those of John F. Kennedy and Bill Clinton (William J. Clinton) using a word cloud, shown in the visualization below. For help, you can look here



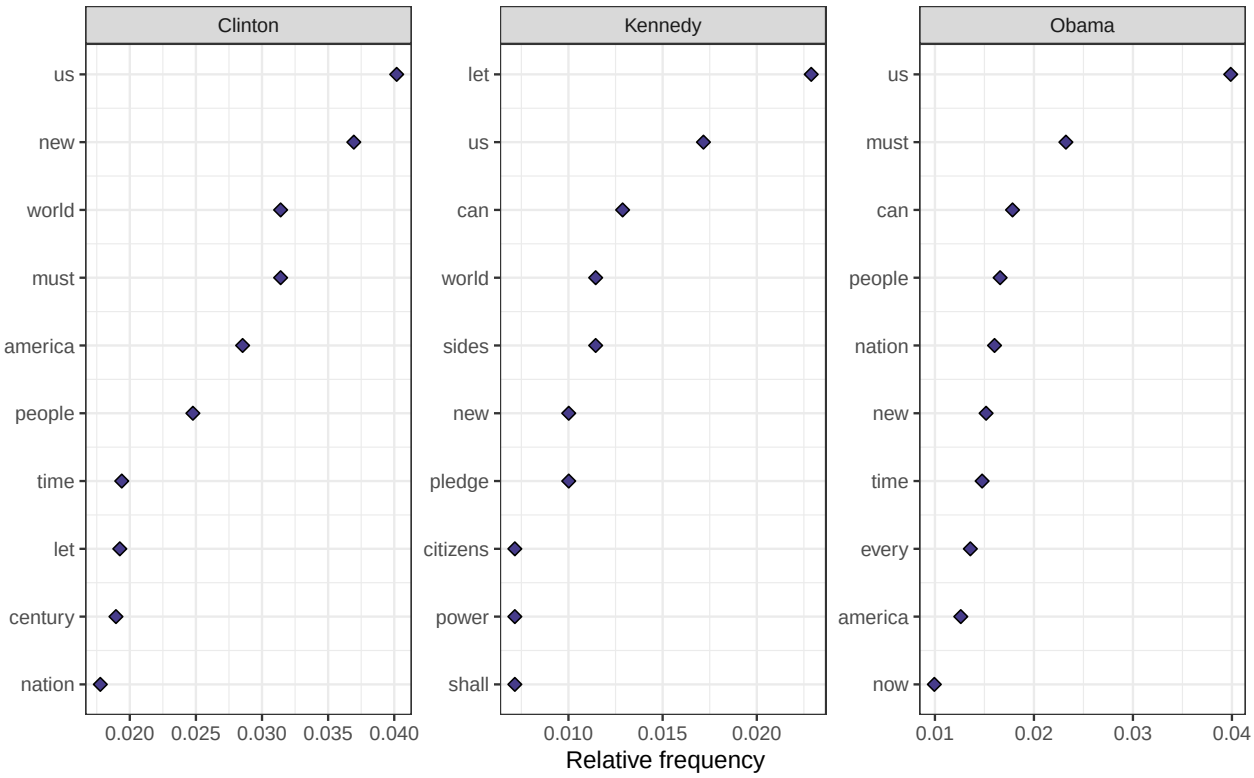
1b) If you superficially compare the speeches given by these three presidents, what can you conclude? To be more precise in your comparison, what type of visualizations could you use? Or what type of annotations to the graph could you make?

1c) Based on the wordcloud above, we see that 'people', 'nations' and 'jobs' are words used a lot by the three presidents in their State of the Union speeches. Would that also hold for their inaugural speeches? Plotting a `kwic` *object produces a lexical dispersion plot which allows us to visualize the occurrences of particular terms throughout the text. We call these "x-ray" plots due to their similarity to the data produced by Amazon's "x-ray" feature for Kindle books. Replicate the following visualization using* `textplot_xray` *and* `kwic` *and the* `data_corpus_inaugural` *corpus from quanteda.*

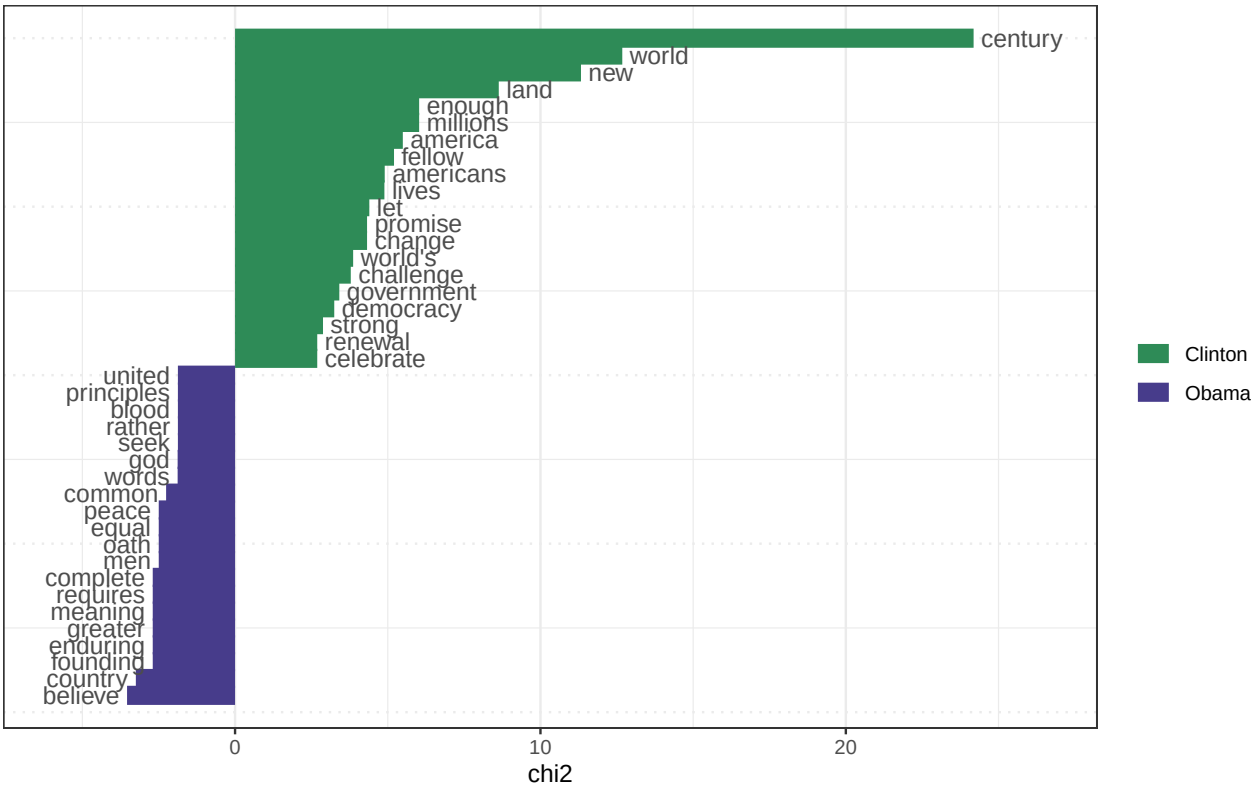


1d) What does the x-ray plot show? Is this easier to compare the three presidents? Why (not)?

1e) Another way to compare text is to look at the frequency of the top features in a text, using `topfeatures`*. Again, use the* `data_corpus_inaugural` *and replicate the following visualization. For some help, have a look here*



1f) If you want to compare the differential associations of keywords in a target and reference group, you can calculate "keyness" which is based on `textstat_keyness`*. Replicate the visualization where Obama is compared to Clinton.*



1g) Make the same visualizations comparing Clinton and Kennedy and Kennedy and Obama. What conclusions can you draw from these comparisons?

Good Luck

The deadline for the assignment is **December 12th, 10am**!

References

Healy, K. (2018). *Data visualization: a practical introduction*. Princeton University Press.

CCS Amsterdam's Github