Big Data meets Small Data

Introduction to R

Contents

VU Amsterdam

1 Why are we learning R?

This week we'll get you started on working with the **R programming language**. As promised in the first lecture, we'll take this slow and steady, so don't worry if the prospect of learning a programming language seems daunting to you. To successfully finish this course you will not be required to write R code from scratch yourself (but for those who want to, we'll provide optional tutorials). You will be able to finish your assignments by just copy/pasting the code that we give you, and tweaking some specific parts of the code as instructed. In other words, you will learn how to use the **R software**, but you will only need to have a very basic understanding of the **R language**.

So what is the value of learning how to work *with* R if you're not actually required to learn the language? There are two reasons in particular. Fist and foremost, this will give you a glimpse **under-the-hood** of the text analysis techniques that you've now seen in OBI4wan. In the previous practical about **validity** you saw that it cannot be taken for granted that an automated analysis accurately measures what we actually want to measure. In this and the coming lectures, we will deepen this understanding by looking closer at how computers actually perform these measurements. This will help you develop a better intuition for what we can and cannot do with automated analyses of texts.

The second reason is that you'll get first-hand experience with the practice of *data science*, which is a growing specialization among social scientists in both academia and business. This will help you determine *whether* this is a direction that you'd like to pursue. And even if you come to the conclusion that programming is definitely not your cup of tea, it's still valuable to have walked some miles in the shoes of a programmer.

This course thereby follows the recommendation of Van Atteveldt & Peng (2018) to at least "stimulate and facilitate" the learning of computational skills:

> Not everyone can (or should) become a programmer, but modern computer languages, libraries, and toolkits have made it easier than ever to achieve useful results, and with a relatively limited investment in computing skills one can quickly become more productive at data driven research and better at communicating with programmers or computer scientists. We think it is vital that we make computational methods more prominent in our teaching to make sure the new generation of communication scientists and practitioners are stimulated and facilitated to learn computational skills such as data analytics, text processing, or web scraping, as applicable. (p. 87-88)

1.0.1 How to work with the R tutorial documents

As you probably noticed, the format of this tutorial is somewhat different from previous weeks. Before the tutorials were PDF files, but now it's this HTML document. So what's up with that?

In R you will be working with the `R syntax`. That is, you'll be giving all the commands to R `via text`. For this, you will often want to copy/paste the `code` (i.e. text in the R syntax) that we provide in the tutorials into R. This format will make it easier and safer to copy this code. You might not mind the difference between a straight quote " and a curly (or smart) quote ", but for R this is a matter of life and death. Furthermore, this tutorial was created with R to ensures that all the code actually works (and doesn't contain typos).

Simply put, this means that as long as you copy/paste all the code from this tutorial, and execute it in the same order as we showed it in the tutorial, everything *should* work. We say *should*, because you will most definitely run into some errors. That's part of the programming game. However, it does mean that the errors are almost definitely the result of (1) problems due to installing packages or (2) the order in which code is executed. Accordingly, it will be much easier to solve your problem.

In this Google Docs document, we provide general tips for solving problems. We will also update this document for any new problems that pop-up. So whenever you run into errors that you can't solve, please first consult this document, and notify us if your problem isn't addressed.

2 What is R?

We keep mentioning this R thing, but chances are you've never heard of it. R is an open-source statistical software language, that is currently among the most popular software packages for data science. It's popular for several reasons:

- R is a programming language, which makes it very versatile. While R focuses on statistical analysis at heart, it facilitates a wide-range of features, ranging from text analysis to advanced visualizations.
- The range of things you can do with R is constantly being updated. R is open-source, meaning that anyone can contribute to its development. In particular, people can develop new *packages*, that can easily and safely be installed from within R with a single command. Since many scholars and industry professionals use R, it is constantly kept up-to-date.
- R is free. While for students this is not yet a big deal due to university discounts for various statistics software, this can be a big plus in the commercial sector. Especially for small businesses and free-lancers.

The trade-off is that the R language has a relatively steep learning curve compared to software such as Excel or SPSS. This is because R primarily works with *syntax*. That is, rather than clicking through menus, you'll give comments to R by typing them. For example, the following lines of code would create a pretty lame wordcloud. *(You don't have to run this code yourself)*

```r
library(wordcloud)
some_text = "This is a super lame wordcloud"
wordcloud(some_text, colors='blue')
```

If you have not worked with a programming language before, it might feel weird to give commands via text. After all, how would you know what commands the computer will understand? That being said, even if this is the first time seeing R syntax, you might already get the basic gist of what's happening here. You might have guessed that we create a wordcloud of the text "This is a super lame wordcloud". You might also understand, that if you would want to change the color of the wordcloud, you'd need to change `colors = 'blue'` to some other color. So while writing R code from scratch takes time to learn, you would be able to copy/paste this code, and make some changes to it.

And that's the level of understanding we're aiming for in this course. You'll see various examples of R code, to perform different types of computational (text) analyses. In the tutorials, we'll tell you more about how R code works, so that you can change parts of it to create different outcomes. But in the assignments, we'll mainly test you on your interpretation of the outcomes, and on your understanding of the analysis. So if you find R stupid and/or hard to learn, don't worry! Conversely, if you like R and want to learn more, we've got you covered with references to optional material. Although this won't be tested in the assignments, you will be able to use this knowledge in the project in period 6.

3 Getting started with R and RStudio

To get started with R, you will need to install two (free) pieces of software.

- *R* is the actual R software that is used to run R code. Technically, this is all you need, but by itself R is not very nice to work with. Rather than using R by itself, it's therefore common to also install a `graphical user interface` (GUI).
- *RStudio* is the most popular GUI for R. It makes working with R a lot easier and more pleasant.

You won't need to run these two pieces of software side-by-side. When you open RStudio, it will automatically also open R. Both programs can be downloaded for free, and are available for all main operating systems (Windows, macOS and Linux).

3.0.1 Installing R

To install R, you can download it from the CRAN (comprehensive R Archive Network) website. **Do not be alarmed** by the website's 90's aesthetics. R itself is cold, dry, no-nonsense software, and the website kind of reflects that. When you're working with R, you'll actually be using RStudio (as we'll install next), which makes everything much prettier and nicer. In previous years we also noticed that some students worry about how safe the website is, since it looks so... old. But yes, this is the official R website, and it can definitely be trusted.

If you have trouble with the download instructions, you can use this link for additional instructions with pictures.

3.0.2 Installing RStudio

The RStudio website contains download links and installing instructions. You will need to install the free *RStudio Desktop Open Source License*. This website is decidedly more user friendly, but if you have trouble figuring out which links to click, this guide again can help.

Note that while you can also get paid licenses for RStudio, these are not required for *using* RStudio. You are also allowed to use the open-source license outside of university for commercial purposes.

3.0.3 What if I can't install R?

Although R works on all operating systems (Windows, macOS, Linux), there are rare cases where it can be difficult to install R, or to install certain packages. Whenever you run into such installation problems, please notify your teacher, and we'll try to make it work. Although R works on almost every system, in the current circumstances it might be difficult to solve everyone's problems remotely. But even in the unlikely case that we can't get R to work on your computer, you don't have to worry. You can also run R from the cloud, which is free for up to 25-hours per month. This should mainly be used for practice, as we recommend doing the assignments on a personal (teammate's) computer. If there are many people in the team who can't get R to work, you can contact the course coordinator (Kasper Welbers) for an alternative solution.

3.1 Using R and RStudio

Once you have installed R and RStudio, you only need to open the RStudio software. If everything was installed correctly, RStudio will automatically launch R as well.

The first time you open RStudio, you will likely see three separate windows. The first thing you want to do is open an R Script to work in. To do so, go to the toolbar and select File -> New File -> R Script.

You will now see four windows split evenly over the four corners of your screen:

- In the **top-left** you have the text editor for the file that you are working in. This will most of the time be an R script. In this script you type all the code for your analysis. You can then select and execute parts of this code (or execute everything at once). Don't worry if this doesn't make sense right away. You will understand it by the end of this practical.
- In the **top-right** you can see the data and values that you are currently working with (environment). So if you load data into R, you will see it here.
- In the **bottom-left** you have the R console, which is where you see the output when you execute lines of code from your script.
- In the **bottom-right** you can browse through files on your computer, view help for functions, or view visualizations. When we create some wordclouds later on, this is where they'll show up.

4 Basics of working with R

As promised, we will not go in-depth in the R language, and you will mainly need to make some minor changes to R code for this course. Still, in order to do so, we need to talk a little bit about the very basics of the R language. In particular, we will discuss 3 things:

- Running / executing R code
- Assigning values to names
- packages and functions

4.0.1 Running code from the R script

We communicate with R by giving it code. There are two steps here. First, we paste (or type) our code into our R script (the text editor that we openened in the top-left corner). Second, we **run** (or execute) this code, or pieces of this code.

So let's first practice this a bit. Copy and paste the following example code into your R Script. For now, don't bother understanding the syntax itself. We'll just focus on how to run it.

```r
3 + 3
2 * 5
6 / 2
"some text"
"some more text"
sum(1,2,3,4,5)
```

You can **run** (i.e. execute) parts of the code in an R script by pressing Ctrl + Enter (on mac this is command/apple-key + Enter). This can be done in two ways:

- If you select a piece of text (so that it is highlighted) you can press Ctrl + Enter to run the selection. For example, select the first three lines (the three mathematical operations) and press Ctrl + Enter.
- If you haven't made a selection, but your text cursor is in the editor, you can press Ctrl + Enter to run the line where the cursor is at. This will also move the cursor to the next line, so you can *walk*

through the code from top to bottom, running each line. Try starting on the first line, and pressing Ctrl + Enter six times, to run each line separately.

4.0.2 Assigning values to names

When you ran the code above, you saw that R **evaluates** expressions. The calculation 3+3 evaluates to the value 6, and 2*5 evaluates to the value 10. You also saw that the **function** *sum(1,2,3,4,5)* evaluates to the value 15 (the sum of the numbers). In this example, R printed the results to our console (the bottom-left window). This is nice if we would just want to do a simple calculation, but often instead of printing values, we often want to **store values** so that we can use them later on in our analysis.

To **store** values, we **assign** them to **names**. In plain terms, **assignment** is how you make R remember things by assigning them to a name. We basically say: "this name shall from now on refer to this value". There are two ways to say this: using the equal sign `=` or an arrow `<-`. In our tutorials we'll use the `equal` sign, but just remember that they are identical.

```
x = 2
y <- "some text"
```

Here we have **assigned** the **value** `2` to the **name** `x` and the value `"some text"` to the name `y`. If you are working in RStudio (which you should), you can now also see these names and values in the top-right window, under the "Environment" tab. Whenever we now use these names in our code, they will represent the values that we assigned to them. In the following line of code, we ask R to show us the value that `x` refers to.

```
x
```

```
## [1] 2
```

We can also immediately use `x` to do calculations.

```
x * 10
```

```
## [1] 20
```

Now, an important thing to remember, is that R can assign basically any type of data to a name. In this case we assigned single numbers (`2`), or single pieces of text (`"some text"`). But we can also assign an entire dataset to a name. Later on in this practical, you'll see that a name can refer to an entire corpus (i.e. collection of texts). But whatever data we assign, it all works in the same way. We **assign** it to a **name**, and from that point on we can refer to the entire dataset by just that name.

4.0.3 Functions and packages

Earlier we showed that we can make wordclouds in R. To do so, we used the following syntax:

```
wordcloud("A very lame wordcloud", colors="blue")
```

If you tried to run this code yourself, you probably got an error saying: `could not find function "wordcloud"`. Don't worry, you'll be able to do this in a minute, but let's first talk a bit about what this error is saying. Two things require clarification:

- What is a function?
- Why can't I find the `wordcloud` function, but you can?

For now, we'll settle for a very simple, practical definition of functions. We can think of a function simply as a tool that given some `input` creates some `output`. The `wordcloud` function takes as input any piece of text, and as output creates an image of a wordcloud. Above we also saw the `sum` function, which takes as input some numbers, and as output gives the sum of these numbers. You can recognize a function as a name followed by parentheses, e.g., `wordcloud("some text", colors="blue")`, `sum(1,2,3)`. The parts between the parentheses contains the input to the function.

These parts are also called the `arguments` of the function, and if there are multiple arguments then they are always separated by commas.

```
function_name(argument1 = ..., argument2 = ...)
```

When we used the `wordcloud` function above we used two arguments. The first argument is the text of which we want to make a wordcloud. This argument is mandatory (we can't really make a wordcloud without a text, after all). The second argument is `colors='blue'`. which tells R that we want to use the color "blue". This is an optional argument, and if we don't provide it a default color ('black') will be used. For this course you mainly need to be able to recognize when a function is used, and what the arguments are. If you want to learn more, or have trouble understanding function, we provide additional instructions here (see Functions section).

Working with functions is at the hearth of any analysis in R. We start with some data as input, and then use all sorts of functions to do stuff with this data. So you can imagine that functions are super useful. As long as we have a function to do what we want, we can just plug our data into this function as input, and obtain our results as output!

So where do we get these super useful functions? Surely, R doesn't just contain millions of functions? Indeed, R itself does not, but there does exist A HUGE COLLECTION of functions that we can quite easily obtain. These functions are available in `packages` that we can download and install into R (kind of like an app-store).

4.0.3.1 Installing packages

To use the `wordcloud` function, we first need to install the `tm` and `wordcloud` packages. We can do this with the following lines of code. Note that you **need** to put `"tm"` and `"wordcloud"` between quotation marks!

```
install.packages("tm")
install.packages("wordcloud")
```

This should have installed the tm and wordcloud package (if you got an error, consult your teacher). It is now stored on your computer in a special `library` for your R packages. Note that you do not need to run this line of code again next time that you want to use the wordcloud package. So whenever you installed a package, you can remove the line of code from your script.

Now that you have the package, we can *almost* use the wordcloud function. The package is in your `library`, but we still need to tell R to retrieve the package from your library to use it in your current session. To do so, we need to say `library(package name)`. Note that this time you **SHOULD NOT** put `wordcloud` between quotation marks.

```
library(wordcloud)
```

And now we can finally use the function.

```
wordcloud("Now we can finally, finally use this function")
```

One final thing to note is that a package often contains multiple functions. The `wordcloud` package also provides functions like `textplot`. So remember that functions and packages are not the same thing. A single package can provide many functions. Whenever you open this package with `library(package)`, all these functions become available.

5 Assignment 1: R syntax basics

In this first assignment we'll practice with how you'll mainly be working with R code in this course.

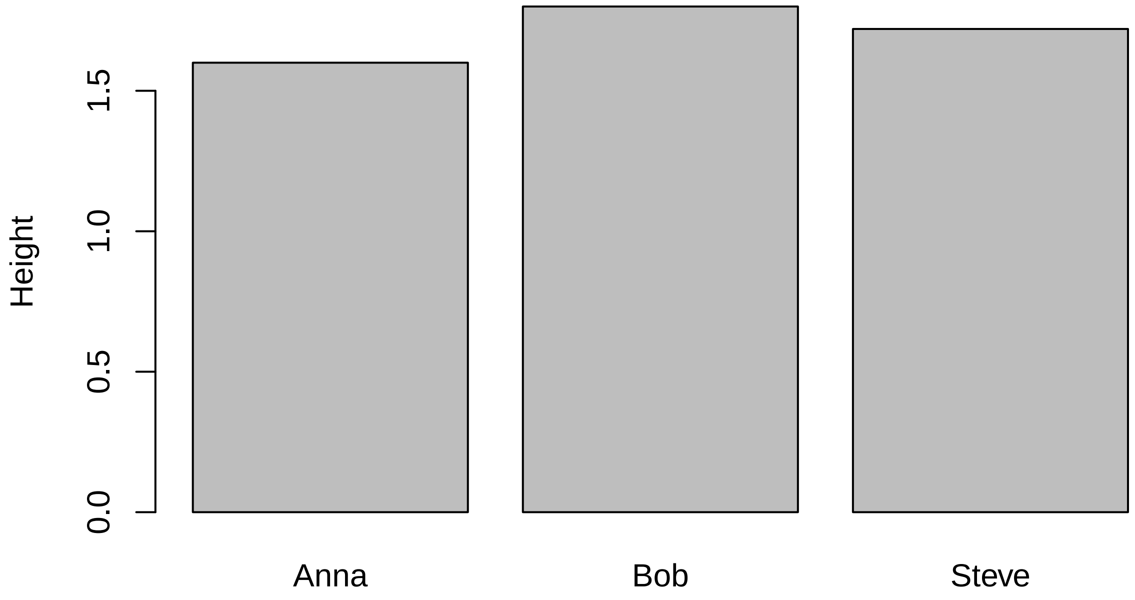
- We provide some code that you copy to your script.
- Then we'll provide some explanation of how the code works, and ask you to make some changes.
- You then run the code, and report the results.

Part of the challenge is that you might have to try different things. Off course, if you get stuck, you can ask your teacher, but please don't give up too early.

Copy the following code to your script:

```r
teammembers = c("Anna", "Bob", "Steve")
heights = c(1.60, 1.80, 1.72)

barplot(heights, names.arg = teammembers, ylab='Height')
```



If you run the code, you see that it creates a Bar graph showing the heights of team members Anna, Bob and Steve. To represent the names and heights of the team members, we created two `vectors`: **teammembers** and **heights**. A vector in R is a collection of multiple values. With the `c` (= combine) function we combine multiple values, separated by commas, into a vector: `c(value, value, value)`. So with `teammembers = c("Anna", "Bob", "Steve")` we created a vector with the names of these three people. The order of values in a vector matters. So we know that the first value in `heights` (1.60) corresponds to the first value in `teammembers` (Anna).

> **Question 1.a**. Change the code so that it contains the names and heights of all your team members. Report your R code, and add a screenshot of the Bar graph.

In the code `barplot(heights, names.arg = teammembers, ylab='Height')`, we also specified that the label of the y-axis should be "Height". We did this by giving an `argument` to the function. Here the argument was `ylab` (short for y-axis label), and with `ylab="Height"` we said that the y-axis label should be "Height".

> **Question 1.b**. Next to the y-axis, we can also provide a label for the x-axis in the plot, using the `xlab` argument. Please specify that the x-axis label should be "Team members". Report the R code (you don't need to provide another screenshot)

Now, you figured this is a pretty boring Bar chart, so you wanted to add some colors. You found out that there is an argument for the `barplot` function, named `col`, that allows you to give colors to bars, using any one of these color names. If the value of the `col` argument is a vector of color names, it will give specific colors to each bar.

You decide to give every bar the favourite color of the respective teammember. We're going to do this in two steps:

> **Question 1.c**. First create (assign) a new vector named `colornames` that contains these colors. (hint: this is just like how we made the `teammembers` vector above). Report your code.

Question 1.d. Second, give the `colornames` vector to the `col` argument in the barplot function. Provide a screenshot of the new colourful Bar chart as your answer.

6 Text Analysis in R

Now we're going to use a popular package for text analysis in R, called `quanteda`. As the first step, we're going to be installing this package.

In general, most R packages can be installed without any issues. However, some of the more complicated packages might require some additional steps. If you just installed the most recent version of R, you should be fine with installing quanteda. But if you run into any issues, please have a look at the instructions here, and consult your teacher if you can't sort it out.

> Troubleshoot: Sometimes R will also *ask* you a question when you try to install packages, and present you certain *answers* that you can give. To *give* an answer, you need to type it in your console. So if it asks you a yes/no question, you type yes or no. In general, when R asks a yes/no question, you should say yes, because it's often to confirm that you want to use the default option. There is one exception, where R asks you if you want to `install packages from source`. Some computers cannot do this (without installing other stuff first), so here you might have to say no. Note that you can simply try again if an option doesn't work. Another question to watch out for is that R might suggest to install some updates, and gives some options. In this case you should either choose all packages, or "CRAN only" packages.

```r
install.packages('quanteda')
```

If successful, **quanteda** has now been installed on your computer. Remember that you only have to install a package once, so you can (and should) remove this code from your script to prevent you from accidentally running it again (and wasting time).

If not successful, first look at the (error) messages. These sometimes do contain easy solutions. If you can't figure it out, ask your teacher. Please note, however, that it's possible that the solution is not easy to find, and the teacher won't be able to help you immediately. If so, please continue the tutorial by looking along with a teammate via Zoom screenshare, and after class send an email to the course coordinator (Kasper Welbers) in which you specify what type of computer you're using, and what your specific error message is.

Now in addition to the general `quanteda` package, we'll also need to install two add-ons. These provide additional text analysis functions that we'll use in this document.

```r
install.packages('quanteda.textstats')
install.packages('quanteda.textplots')
```

6.1 Quanteda

To start working with quanteda, we first need to run `library(quanteda)` to tell R that we want to use this package in our current session. We'll do the same for the two add-ons.

```r
library(quanteda)
library(quanteda.textstats)
library(quanteda.textplots)
```

6.2 Preparing the inaugural speeches Corpus

We're now going to prepare data for performing some text analysis techniques. We'll be skipping over some details about how this works, but don't worry, we'll get back to those in the next practical meeting.

> It's important to also excute all the code in the following steps yourself, because you will need some of it in your assignment!

In text analysis, the term **corpus** is often used to refer to a collection of texts. For this tutorial, we'll use a demo corpus that is included in the **quanteda** package. The corpus is called `data_corpus_inaugural`, and contains the inaugural speeches of US presidents. For convenience, we'll assign the corpus to the name **corp**

```
corp = data_corpus_inaugural
```

> If you get the error message `Object 'data_corpus_inaugural' not found`, then you forgot to run `library(quanteda)`

Now, if we run just the name `corp`, R will print (a sample of) the texts in the corpus.

```
corp
```

```
## Corpus consisting of 59 documents and 4 docvars.
## 1789-Washington :
## "Fellow-Citizens of the Senate and of the House of Representa..."
##
## 1793-Washington :
## "Fellow citizens, I am again called upon by the voice of my c..."
##
## 1797-Adams :
## "When it was first perceived, in early times, that no middle ..."
##
## 1801-Jefferson :
## "Friends and Fellow Citizens: Called upon to undertake the du..."
##
## 1805-Jefferson :
## "Proceeding, fellow citizens, to that qualification which the..."
##
## 1809-Madison :
## "Unwilling to depart from examples of the most revered author..."
##
## [ reached max_ndoc ... 53 more documents ]
```

Here **quanteda** lets us know that the corpus contains 58 documents, and 3 docvars. The docvars are **variables** about the documents, in this case the first and last name of the president, and the year of the speech. We can view them with the *docvars()* function.

```
docvars(corp)  ## (only the first lines of output are shown here)
```

Year	President	FirstName	Party
1789	Washington	George	none
1793	Washington	George	none
1797	Adams	John	Federalist
1801	Jefferson	Thomas	Democratic-Republican
1805	Jefferson	Thomas	Democratic-Republican
1809	Madison	James	Democratic-Republican

Now as a final step, we're going to create a Document-Term Matrix (DFM). This is a matrix in which the rows are documents, and the columns are all the unique terms (aka features) that occurred in these documents. The cells in this matrix tell us how often each term occurred in each document. In the second lecture we'll talk more about what this DFM is and what we can (and can't) do with it. For now, just try to understand that by representing our corpus as a matrix with term frequencies per document, we have transformed texts into data that we can do calculations with. This is what enables us to apply the following text analysis techniques.

```r
tokenized = tokens(corp, remove_punct=TRUE)
m = dfm(tokenized)
m = dfm_remove(m, stopwords('en'))
```

Here we used the `tokens` function to tokenize the texts. Note that the main input for the `tokens` function is the corpus that we created above. With `remove_punct=TRUE` we state that we only want to get the words, and ignore punctuation. With the `dfm` function we then created the DFM, which we assigned to the simple name `m` (short for matrix).

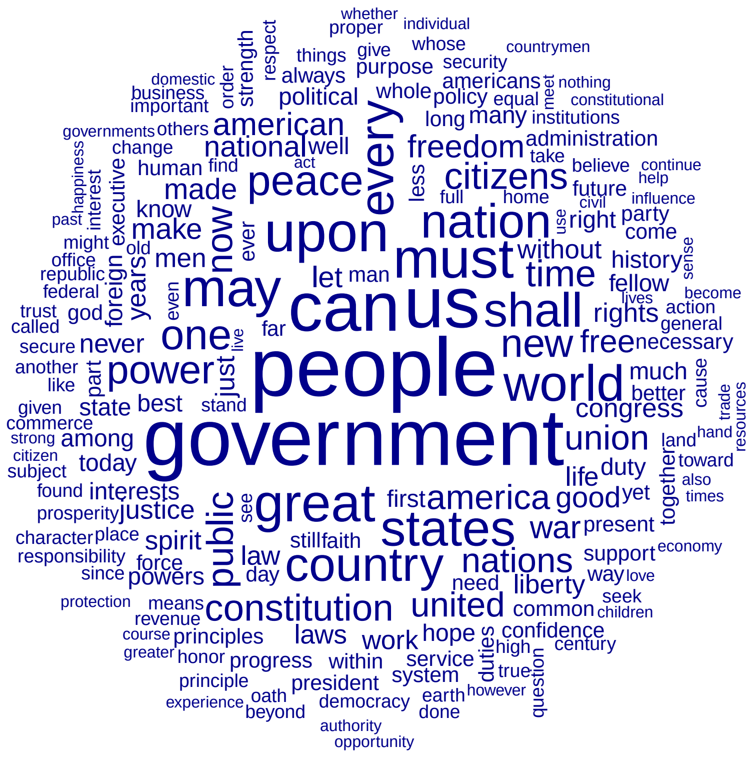
If you want to get a glimpse at what this matrix looks like, you can click on the `m` object in the rop-right window in RStudio. This will show you only a few of the many columns (because otherwise it might crash you computer)

6.3 Word clouds

You have seen wordclouds in OBI4wan. Although wordclouds are pretty simplistic, they can be a nice way to get a quick overview of the most common words in a corpus.

To get a basic idea of what presidents talk about, we can create a wordcloud with quanteda's `textplot_wordcloud()` function. The main input for this function is the DFM that you created in the previous step. As an additional argument we set min_count (the minimum wordcount) to 50 to ignore all words that occurred less than 50 times.

```r
textplot_wordcloud(m, min_count = 50)
```



OK, that's a decent start. Note that R will automatically shape the image based on the width and height of you plottign window (the bottom-right window). If your wordcloud doesn't look nice, try making the window wider or taller and re-reun the `textplot_wordcloud` function.

A wordcloud of all speeches can be interesting, but we can also focus on specific presidents. For example, we can say that we only want speeches by Obama. Remember that above we saw that for each text we have a document variable (docvar) with a column `President` that holds the last name of the president. So we can say that we only want speeches where this name is "Obama"

```
m_obama = dfm_subset(m, President=="Obama")
```

Let's look at what's happening here. We use a function called `dfm_subset`, which let's us create a subset (i.e. a smaller part of) a DFM. As input we give it our DFM called `m`, and we also provide a condition for creating our subset: `President == "Obama"`. Notice the double equal sign `==`. This is a common way in programming languages to say that the values before and after `==` should be equal. So you should read this code as: We make a subset of the data where the value of the `President` column is equal to `"Obama"`.

Alternatively, we could focus on speeches from after the second world war. Remember that we also had a column `Year`. So we can say that Year needs to be higher than 1945

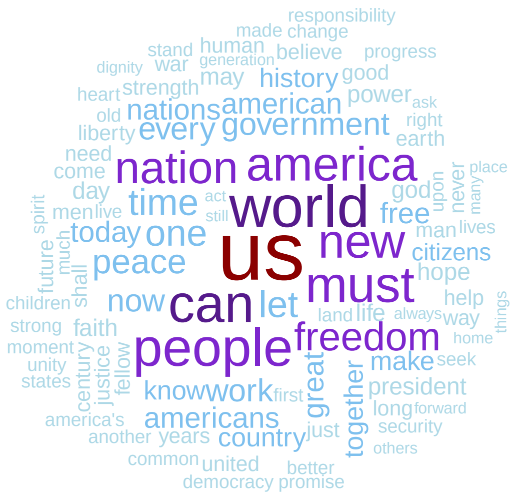
```
m_postwar = dfm_subset(m, Year > 1945)
```

Again, try to understand what this is saying. Here we take a subset of the data where the value of the `Year` column is greater than (`>`) `1945`

Note that this has not deleted other years or presidents from our existing DTM `m`, but created two new DTMs `dtm_obama` and `m_postwar` to contain the subsets. Indeed, we `assigned` this data to new names. If you look in the top-right window under the `Environment` tab, you should now also see that you have multiple "data" rows: `m`, `m_obama` and `m_postwar` (assuming that you also performed all of the above steps yourself).

Let's plot one of these, and let's also use some colors in addition to wordsize to complement the differences in wordfrequency. You can pass multiple colors to the function to achieve this.

```
textplot_wordcloud(m_postwar, max_words = 100,
                   color = c('lightblue', 'skyblue2','purple3', 'purple4','darkred'))
```



Alright, that'll do for now. Try to play around with the code a bit. If you want to use a different color combination, you can get a list of the available colors, or see this page for an overview of colors.

```
colors()      ## output not printed in this document
```

7 Assignment 2: Wordclouds

For this assignment, you'll first make a wordcloud for a different president (i.e. not Obama). You will first want to copy the code that we used to create the DTM for "Obama", and the code to create a wordcloud. This time we'll help a bit:

```
m_obama = dfm_subset(m, President=="Obama")

textplot_wordcloud(m_obama, max_words = 100)
```

For the assignment you'll have to show your wordcloud. Rather than making a screenshot, you'll get a higher quality image by copying the image from R. In the window that shows the visualization, there is an `Export` button at the top. Here you can either save the image to a file, or choose `Copy to clipboard`, where you can directly copy it.

Also, remember that you could look at the `docvars(m)` to see all the **document variables** that you can filter on.

```
docvars(m)
```

> **Question 2.a**. Change the code so that you'll create a wordcloud for a different president. Report the code and wordcloud.
>
> **Question 2.b**. Add some colors to the wordcloud using the `color` argument (as seen above). Report the code and wordcloud.
>
> **Question 2.c**. Interpret the wordcloud. What words are most prominent? Does this help you get a quick idea of what the president talked about?
>
> **Question 2.d**. Now, you're going to make two other wordclouds. One for all "Republican" Presidents and one for all "Democratic" Presidents. (hint: see the Party column in the document variables). You can ignore the other parties and the "Democratic-Republican" candidates like Jefferson. Report the code and wordcloud.
>
> **Question 2.e**. Interpret the wordclouds. What are the differences and similarities in the most prominent words? What would you say, based on these wordclouds, about the differences between Republican and Democratic presidents in what they talk about in their State-of-the-union speeches?

8 From Wordclouds to Keyness

Above we created two wordclouds and manually compared them. In each wordcloud we saw the most common terms used in the subset. This can give us a rough indication of whether two corpora (i.e. plural of corpus) are similar. But what if we want to specifically investigate how they are different? For instance, what if we want to specifically see whether there are certain words that President Obama was more likely to use compared to other presidents?

For this, we can compare the frequencies of word use in two corpora. So instead of looking at the most common words, we compare how much more likely a word is to occur in one corpus. In `quanteda` this is called the `keyness`, and we can compute it as follows.

```
tk = textstat_keyness(m, docvars(m, 'President') == "Obama")
```

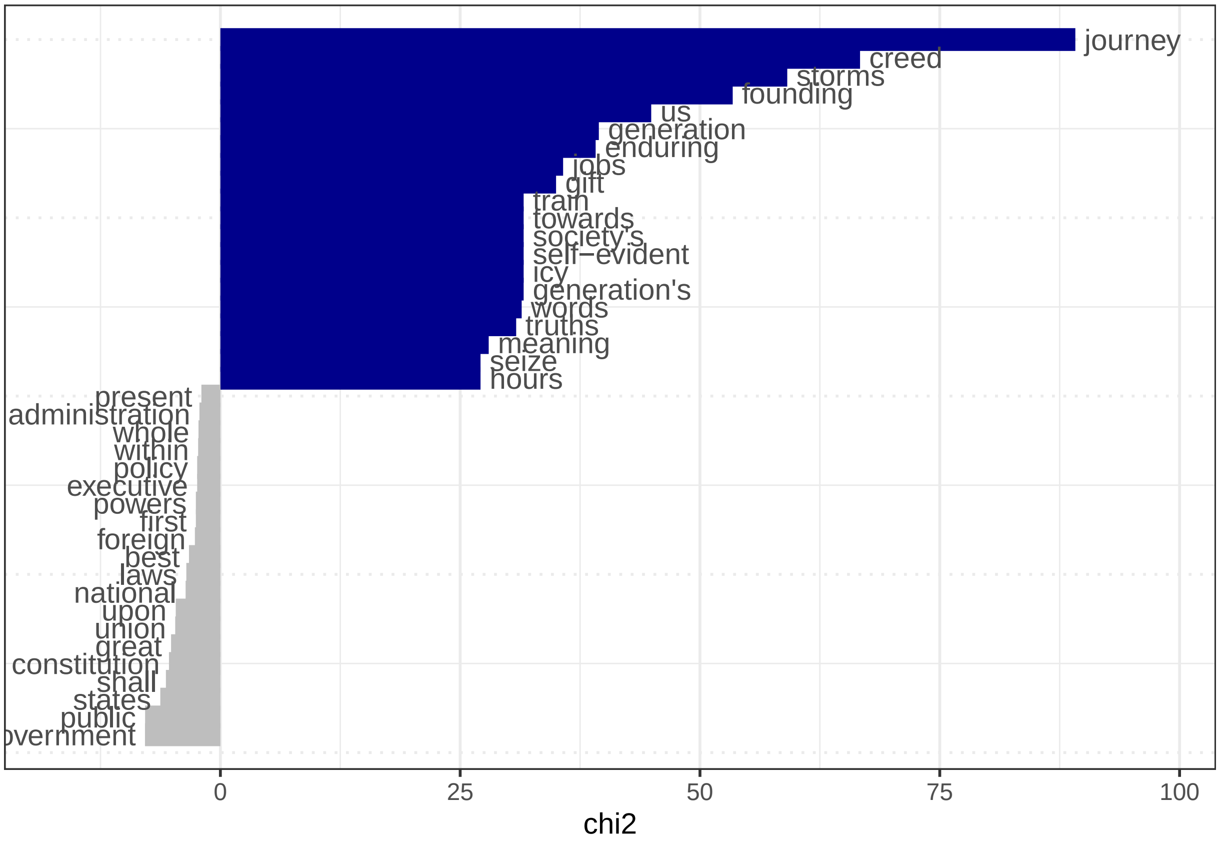
The code is a bit more complicated than before, but if you look closely you should recognize the general idea. We use the `textstat_keyness` function, which will compute for us the `keyness` text statistic. As input, we first provide our DFM called `m`.

Next, we need to specify which texts in the corpus we want to compare to the other texts. In this example, we want to compare Obama to all other presidents. Before in the `dfm_subset` function we could simply say `President == "Obama"`, but in this function this is not allowed (sometimes, life is not perfect). Instead, we need to use the less elegant `docvars(m, "President") == "Obama"`. The difference is that now we need to explicitly tell R to look for the `President` column in the document variables for m (`docvars(m)`).

Sidenote. As you see, it is not always staightforward *how* a function should be used. For this course you'll never have to figure this out by yourself, but you might be wondering how we know this. We provide an optional section at the bottom of this document explaining this.

You don't need to understand why the `textstat_keyness` function works this way, just that it does. So now let's visualize the results from the `textstat_keyness` function, using the `textplot_keyness` function. For sake of simplicity, we repeat the previous step, so that you see that the `textstat_keyness` and `textplot_keyness` functions work in tandem.

```
tk = textstat_keyness(m, docvars(m, 'President') == "Obama")
textplot_keyness(tk, show_legend = FALSE)
```



In the plot we see 20 terms that are over-represented in Obama's speeches, and 20 terms that are under-represented. The over-represented terms are those that Obama was much more likely to use compared to other presidents, and the under-represented terms are those that Obama was much less likely to use.

The over-represented terms are the blue bars, which have a positive value on the x-axis. Most strongly over-represented are the terms "journey", "creed", "storms" and "founding". The 20 terms with the grey bars are the under-represented terms, meaning that Obama used them relatively less often than the average president.

You can also see the exact numbers by looking at `tk`. (Here we just show the top rows)

```
tk
```

feature	chi2	p	n_target	n_reference
journey	89.13356	0	9	12
creed	66.68814	0	6	6
storms	59.09945	0	3	0
founding	53.41252	0	5	5

feature	chi2	p	n_target	n_reference
us	44.91788	0	44	461
generation	39.45575	0	9	31

So we see that the term (aka feature) “journey” is on top, with a chi2 value of 87.37. We won’t interpret what this means, but you might remember the Chi^2 test from statistics. In the `quanteda` keyness analysis, the terms with the highest positive chi2 scores are the most over-represented, and the terms with the lowest negative chi2 scores are the most under-represented. The `n_target` column furthermore tells us that Obama used the term 9 times, and all other presidents combined (`n_reference`) used the term 12 times. Relative to the total number of word spoken, Obama used this word much more often, which can tell us something about the message Obama was trying to convey.

9 Assignment 3: Keyness

For this assignment you’ll be using the keyness code from the previous section. You’ll again have to copy (some) of the code, and change it for your own analysis.

> **Question 3.a**. Create a keyness plot for the president that you analyzed in `2.a`. Report the code and keyness plot.
>
> **Question 3.b**. Interpret the results. What terms were over-represented and under-represented? Does this help you get a quick idea of what the president talked about? How is this different from your observation in `2.c`?
>
> **Question 3.c**. You work for OBI4wan to advise them on usefull tools in the dashboard for their users. They heard about this `keyness` thing, and are in doubt whether it is something that would be usefull. You are asked to look into this, and write a brief (+/- 200 word) evaluation. Specifically, you are asked to reflect on how this type of analysis is different from just looking at word frequencies in a wordcloud.

10 Optional: better understanding R

10.0.1 What If I want to learn more?

If you want to learn more about R besides what you need for this course, there are various places to start. One of them is a page of tutorials maintained by teachers of this course (Wouter van Atteveldt and Kasper Welbers). The `Getting started` tutorial on this pages covers the same topics as this tutorial, so you could skip that.

A fun place to start is the `Data mangling in the tidyverse` section. The `tidyverse` is a modern, popular and powerful approach to managing and visualizing data. If you would rather focus on something that is also relevant to the current course (and that you could use for the project in period 6), the `Text analysis` section will cover many of the same topics, but with a stronger focus on actually learning how to do this yourself in R.

10.0.2 How to figure out how a function works

At several examples in this tutorial you might have been thinking: Sure, I can more or less see what’s happening, but how would I EVER figure this out on my own? That’s a fair sentiment, and the good news is, **nobody** figures this out “on their own”. Programming is 90% of the time looking up documentation. Becoming a good programmer is for a large part becoming good in knowing how and where to search for explanations and example code. You only need to understand enough about the basics of a language to understand these explanations and examples.

Conveniently, R packages typically come with a lot of documentation about functions included. You can access this documentation by typing a question mark in front of a function, and running the code. Let's do this for the `textplot_wordcloud` function

```
?textplot_wordcloud
```

The first time you see this, it's probably a bit overwhelming, because it contains a lot of information about a function. The nice thing is that the structure of these help pages is always the same, so after a few times you'll know exactly what to look for.

If you want to know what the main purpose of a function is, it's good to start with the `Description`. But often you already know what a function does, and just want to know what `arguments` there are. This is listed in the `Arguments` section. So in our example we see that there is an argument called `max_words`, that we can use to specify the `maximum number of words to be plotted`. In the `Usage` section (above `Arguments`), we can also see the default settings for these arguments. Here we see that the default is `max_words = 500`.

If you want to get a quick idea of how to use a function, a good strategy is to just scroll all the way down to the `Examples` section. This section contains example code, that you can copy to your script to play around with the function. Remember that weird case where we had to use `docvars(m, 'President') == 'Obama'` in the `textstat_keyness` function? We simply adopted the solution from the example of the pre-post war comparison

```
?textstat_keyness
```

If the R documentation doesn't help you, you can always Google. If you Google for `quanteda textstat_keyness`, you'll find various pages with help. One will just be the same help page, but more nicely formatted. But there are also additional instructions and tutorials.