Introduction to Quantitative Text Analysis

Kostas Gemenis and Bastiaan Bruinsma

Contents

Welcome! **5**

1 Getting Started with Quantitative Text Analysis **7**
1.1 QTA in steps . . . 8
1.2 How this book works . . . 9
1.3 Further Literature . . . 9

2 Getting Started with R **11**
2.1 What is R? . . . 11
2.2 R on Windows . . . 11
2.3 R on Linux . . . 12
2.4 R on MacOS . . . 12
2.5 R in the Cloud . . . 13

3 Installing Packages **15**
3.1 Installing from CRAN . . . 15
3.2 Installing from GitHub . . . 15
3.3 Packages for Quantitative Text Analysis in R . . . 16
3.4 Problems, Bugs and Errors . . . 18

4 Importing Data **19**
4.1 Text in R . . . 19
4.2 Import .txt Files . . . 21
4.3 Import .pdf Files . . . 22
4.4 Import .csv Files . . . 23
4.5 Import from an API . . . 24
4.6 Import using Web Scraping . . . 27

5 Reliability and Validity **29**
5.1 Inter-Coder Agreement . . . 30
5.2 Visualizing Quality . . . 33

6 Preliminaries **41**
6.1 The Corpus . . . 41

6.2 Keywords in Context 43
6.3 Visualisations and Descriptives 44
6.4 Text Statistics 46

7 Dictionary Analysis **51**
7.1 Classical Dictionary Analysis 51
7.2 Sentiment Analysis 54
7.3 Sentiment Analysis using VADER 63

8 Exercices **67**

9 Scaling Methods **69**
9.1 Wordscores 69
9.2 Wordfish 78
9.3 Correspondence Analysis 81

10 Supervised Methods **85**
10.1 Support Vector Machines 85
10.2 Naive Bayes 90

11 Unsupervised Methods **99**
11.1 Latent Dirichlet Allocation 99
11.2 Seeded Latent Dirichlet Allocation 106
11.3 Structural Topic Model 108

12 Texttricks **117**

Welcome!

Welcome to our introductory textbook on quantitative text analysis! This book originated as a collection of assignments and lecture slides that we prepared for the ECPR Winter and Summer Schools in Methods and Techniques. Later, as we taught the Introduction to Quantitative Analysis course at the ECPR Schools, the MethodsNET Summer School, and seminars at the Max Planck Institute for the Study of Societies, Goethe University Frankfurt, Chalmers University, and the Cyprus University of Technology, we added more and more material and text, resulting in this book. The version you see today has been updated with the help of a grant from the [Learning Development Network] (https://ldn.cut.ac.cy/) at the Cyprus University of Technology. For now, the book focuses on some of the best-known quantitative text analysis methods in the field, showing what they are and how to run them in R.

So why bother with quantitative content analysis? For one thing, we can say that developments over the last twenty years have made research using quantitative text analysis a particularly exciting proposition. First, the huge increase in computing power has made it possible to work with large amounts of text. Second, there is the development of R - a free, open-source, cross-platform statistical software. This development has enabled many researchers and programmers to develop packages for statistical methods for working with text. In addition, the spread of the internet has made many interesting sources of textual data available in digital form. Add to this the emergence of social media as a massive source of text generated daily by millions of users around the world.

However, quantitative text analysis can be a daunting experience for someone unfamiliar with quantitative methods or programming. Our aim with this book is to guide you through it, combining theoretical explanations with a step-by-step explanation of the code. There are also several exercises designed for those with little or no experience in text analysis, R, or quantitative methods. Ultimately, we hope that you will find this book not only informative but also engaging and educational.

Chapter 1

Getting Started with Quantitative Text Analysis

Quantitative text analysis, like many other techniques, is at its core a method. This means that while it provides you with the tools to answer a particular question, it does not provide you with a theoretical framework. Nor is there any reality to be discovered: the only thing we can do with QTA methods is to provide (hopefully) accurate summaries of our texts.

With this in mind, there are five questions that we can hope to answer with QTA (Grimmer et al., 2022):

1. What do our texts look like?
2. What are our texts about?
3. What do our texts measure?
4. What can our texts predict?
5. What can our texts prove?

In the case of the first question, we might simply be interested in how many words we have in different documents, which authors wrote together, or whether certain texts have a distinctive type of wording. Questions like these help us to get a better idea of the type of data we are dealing with, to work out what might be interesting to look at and to identify potential problems early on. We can then ask ourselves what the texts are actually about. Here we could run topic models to look at (a representation of) the underlying structure of our texts. We could look at the sentiment of the texts using different dictionaries, or calculate different readability statistics. In each case, we get a better understanding of what our texts represent and what they might be about. For example, we might discover that some texts cluster together unexpectedly, or have more themes in common than we expected. This might then lead us to focus on them exclusively, to collect more texts on the same topic, or to focus on different documents

altogether.

Now that we have our texts and know what we want from them, we can start to use them to measure a concept we are interested in. For example, we could use the codes from the Manifesto Project to measure the political left-right position of our texts. Or we could measure the occurrence of different issues in these documents to find out the agendas of different parties. Once this is done, we could then use these measurements to predict what kind of agenda a new party might have, or a step further, we could use them to estimate what effect a particular text will have on, say, voters or legislators.

Note that we can stop this process and any point and do interesting work. That is, if we go about rigorously collecting texts from different sources, structuring them, cleaning them, and describing what they are like, this can be interesting in itself. Similarly, measuring the positions of texts on a variety of scales can be enough to make for interesting research. Ultimately, how far you go depends on the questions you want to ask and the problems you want to solve.

1.1 QTA in steps

So what does a QTA look like in practice? Let's say you already have an idea of what you want to do and what questions you want to ask. Then you need to go through the following steps:

1. **Choose and select**. If you want to look at political manifestos, do you want to see all of them or just those of the major parties? And do you want only the most recent ones or all of them?

2. **Find and collect**. Find all the texts you need and save them somewhere. Make sure that everything you want is included, that you have the right version of the document and that the documents are in a readable format (pdf, txt or .doc format).

3. **Check**. If your documents are in .txt format, are there any conversion errors? For example, is a letter like "Ü" visible in the document, or do you see something like "™" instead? Note that most researchers work in English, and non-English and non-Latin alphabets can cause problems. The best option is to ensure that all your documents are in UTF-8 (more on this here).

4. **Create a corpus**. Load all the texts you want to analyse and associate them with any metadata you want to include. Then transform the texts into a data frequency matrix (DFM). This matrix has your individual texts in the rows and all possible words in the columns - this turns your corpus of textual data into a matrix of numbers.

5. **Preprocess**. Remove words you do not need, such as stop words, remove punctuation, and apply stemming or lemmatisation algorithms.

6. **Describe**. Check your data. Are there any words that occur a lot (and which you might want to remove?) Are there any strange patterns? Is the data in the right form?

7. **Run your model**. Select your model, run it and check that all the hyperparameter settings are correct. Check that all the steps are correct and repeat the process at least once to ensure future reproducibility.

Visualise and interpret. Look at your results using tables and graphs and try to see if you can answer your research question.

Of all these steps, you will often find that the last two are the most commonly covered. However, it is equally important to ensure that your data is correct and of sufficient quality to be of real use. Often problems later in the analysis are caused by problems in the data early on.

1.2 How this book works

From here on, this book will work as follows. In 2 we will look at R and how to get it working on our system. Our choice of R here is based on the fact that it is both open and free, as well as being the current choice of software for most social scientists (and therefore the one you are most likely to be working with). R also has a wide range of packages that we can use for text analysis. Then in @ref(#importing-data), we will focus on the actual texts we are going to use and how to get them into R. This will cover converting PDF files to TXT, reading CSV files, and downloading files from an on-line database. Then in 5, we will cover the outstanding issues of reliability and validity, and how to ensure that the codes you get from (a more classical) text analysis are reliable enough to use later. Then, in @ref(#preliminaries), we look at what is in our data and how we can best describe the texts we are dealing with. Finally, the last four chapters cover the four main types of techniques we can use to find out more about our text or measure things about it: 7, 9, 10 and 11. We conclude, of course, with a list of references that we have used and that you can use if you are interested in learning more.

1.3 Further Literature

There has been no shortage recently of good introductions to content analysis. Which book would be best for you depends therefore mostly on your focus. For a traditional (more qualitative) introduction, “Content Analysis - an Introduction to Its Methodology” by Klaus Krippendorff (currently in its 4th edition) is a good place to start. For a more quantitative approach, “Text as Data: A New Framework for Machine Learning and the Social Sciences” by Grimmer, Roberts and Stewart from 2022, is the latest in combining machine learning with text analysis. Finally, for another qualitative approach, “The Content Analysis

Guidebook" by Kimberly Neuendorf (currently in its 2nd edition) delves deeper into the underlying theory (which we cursorily discuss here).

Chapter 2

Getting Started with R

Over the last few years, two approaches have emerged to deal with issues related to the quantitative analysis of text: R and Python. While Python is a general-purpose language (useful for everything from websites to games to databases), R was always designed with statistics in mind. As a result, R is better suited to deep statistical analysis and a wide range of data visualisation. As a result, while Python is a good choice when dealing with issues related to machine learning and large-scale applications, R is better suited to statistical learning and data exploration. Since we are mainly interested in the latter here, we built this book around R (and all the packages it offers). If you are more interested in large-scale analysis using Python, be sure to take a look at its spaCy and NLTK libraries.

2.1 What is R?

R is an open-source program that allows you to perform a wide variety of statistical tasks. At its core, it is a modification of the S and Scheme programming languages, making it not only flexible but also fast. R is available for Windows, Linux and OS X, and is updated regularly. In its basic form, R uses a simple command line interface. To make it more friendly, you can use one of the integrated development environments (IDEs) such as RStudio, Jupyter and R Commander. These environments not only look better but also offer some additional practical features. In this book, we will use RStudio as it is (in our opinion) the best-looking and working IDE available.

2.2 R on Windows

To install R on Windows, go to https://cran.r-project.org/bin/windows/base/, download the file, double-click it and run it. During installation, it is best to leave default options (such as the installation folder) unchanged. This makes it

easier for other programs to know where to find R. Once installed, you will find two shortcuts for R on your desktop. These refer to the two versions of R that come with the installation - the 32-bit version and the 64-bit version. Which version you need will depend on your version of Windows. To find out which version of Windows you have, go to **This PC** (or **My Computer**), right-click and select **Properties**. Here you should find the version of Windows installed on your PC. If you have a 64-bit version of Windows, you can use either version. However, it is best to use the 64-bit version as it makes better use of your computer's memory and runs more smoothly. If you have the 32-bit version of Windows, you must use the 32-bit version of R.

To install RStudio, go to https://www.rstudio.com/products/rstudio/download/ and download the free version of RStudio at the bottom of the page. Make sure you select **Installers for Supported Platforms** and select the option for Windows. Once downloaded, install the program leaving all settings unchanged. If all goes well, RStudio will have found your installation of R and placed a shortcut on your desktop. Whether you have a 32-bit or 64-bit version of Windows or R does not matter to RStudio. What does matter are the slashes. R uses slashes (`/`) instead of the backslashes (\) that Windows uses. So whenever you refer to a folder or file in R, make sure you reverse the slashes. For example, you should refer to a file that has the Windows address **C:\Users\Desktop\data.csv** as **C:/Users/Desktop/data.csv**.

2.3 R on Linux

How you install R on Linux depends on which flavour of Linux you have. In most cases, R is already included in your Linux distribution. You can check this by opening a terminal and typing `R`. If it is installed, R will start in the terminal. If R is not part of your system, run the following in a terminal

1. sudo apt update
2. sudo apt install r-base r-base-dev -y
3. Now run "R" to see if the installation worked

To install RStudio, go to https://www.rstudio.com/products/rstudio/download/. At the bottom of the page, select the installer for your operating system. Then install the file either using an installation manager or via the terminal. After running the launcher, you will find RStudio in the Dash.

2.4 R on MacOS

For OS X, you must have OS X 10.6 (Snow Leopard) or later. Otherwise, you can still install R, but you will not be able to use a certain number of packages (like some we use here). To check this, click on the Apple icon at the top left of your screen. Then click on the **About This Mac** option. A window should appear telling you which version of OS X (or MacOS) you have.

To install R, go to https://cran.r-project.org/index.html and click on **Download R for (Mac) OS X**. Once there, download and install the appropriate .pkg file for your version of OS X. You will also need to download the **Clang 6.x Compiler** and the **GNU Fortran Compiler** from https://cran.r-project.org/bin/macosx/tools/. Install both, leaving the selected options as they are. Once installed, check that R works by running it.

To install RStudio, go to https://www.rstudio.com/products/rstudio/download/ and download the OSX version at the bottom of the page. Once downloaded and installed, you should be able to find RStudio among your other applications.

2.5 R in the Cloud

As well as installing R on your system, you can also use the cloud version. RStudio hosts this version at https://posit.cloud/. To use it, go to the Sign-Up button at the top right of the screen. Then select the **Cloud Free** option and select Sign-Up again. Finish the process by either filling in your details or connecting to your Google or GitHub account. Once done, log in and you will be taken to your workspace. To get started, you will need to create a new project. To do this, click on the **New Project** button, which will take you to an instance of RStudio. From here on, the program works in exactly the same way as the desktop version. Note that anything you do - or packages you install - in the project will remain in the project. So if you want to create a new project, you will have to reinstall it. Also, note that Posit Cloud is highly dependent on both the number of users on the server and your internet connection. Therefore, some actions (such as installing packages) may take longer to complete.

Chapter 3

Installing Packages

R on its own is a pretty bare-bones experience. What makes it work are the many packages that exist for it. These packages come in two types: those with an official release and those in development.

3.1 Installing from CRAN

For an official release, the package must be part of CRAN: the *Comprehensive R Archive Network*.CRAN is a website that collects and hosts all the material R needs, such as the various distributions, packages and more. The main advantage of being official is that it means the package has been through a review process. This ensures that the package is free of major bugs, has README and NEWS files, and has a unique version number. In addition, packages that have been officially released often have accompanying articles in *The R Journal* or *The Journal of Statistical Software* that provide detailed explanations of the code, examples, etc.

To install official packages, we can use the `install.packages' command or the **Packages** tab in RStudio. If we run`install.packages("package", dependencies=TRUE)`, we will also install any other packages a package depends on. Finally, to update, we can either go to the **Packages** tab in RStudio and click the **Update** button. We can also type`update.packages()` in the **Console** for the same result. Try to update your package often to avoid unnecessary bugs and problems.

3.2 Installing from GitHub

Some packages that have not yet had an official release are in development on **GitHub**. As a result, these packages change very often and are more unstable

than their official counterparts. We can install packages from Github using the `devtools` package. To install this, type:

```
install.packages("devtools", dependencies=TRUE)
```

Here, `dependencies=TRUE` ensures that if we need other packages to make `devtools` work, R will install these as well. Depending on your operating system, you might have to install some other software for `devtools` to work.

On Windows, `devtools` requires the **RTools** software. To install this, go to https://cran.r-project.org/bin/windows/Rtools/, download the latest **recommended** version (in green), and install it. Then re-open R again and install `devtools` as shown above.

On Linux, the way you install `devtools` depends on the flavour of Linux you have. In most cases, installing 'devtools' from the RStudio console will work fine. If not, the problem is most likely a missing package in your Linux distribution. To fix this, close R, open a terminal and type:

1. `sudo apt-get update`
2. `sudo apt-get upgrade`
3. `sudo apt install build-essential libcurl4-gnutls-dev libxml2-dev libssl-dev`
4. Close the terminal, open R, and install `devtools` as shown above.

On OSX (or macOS), `devtools` requires the *XCode* software. To install this, follow these steps:

1. Launch the terminal (which you can find in */Applications/Utilities/*), and type:
2. In the terminal, type: `xcode-select --install`
3. A software update window should pop up. Here, click **Install** and agree to the Terms of Service.
4. Go to R and install `devtools` as shown above.

3.3 Packages for Quantitative Text Analysis in R

There are several packages that we can use for quantitative text analysis in R, such as `tm`, `tidytext`, `RTextTools`, `corpus` and `koRpus` (Welbers et al., 2017). Many of these packages offer specialised features that can sometimes be very useful, but in this book, we will mainly rely on `quanteda` (Benoit et al., 2018), which is currently in its fourth version. The advantage of `quanteda` over other packages is that it integrates into a common framework many of the text analysis functions of R that were previously spread across many different packages(Welbers et al., 2017).In addition, many 'quanteda' functions can be easily combined with functions in other packages, while the package as a whole has simple and logical commands and a well-maintained website.

The current version of `quanteda` at the time of writing is 4.0. This version works best with R version 4.0.1 or higher. To check if your system has this, type `R.Version()` in your console. The result will be a list. Look for `$version.string` to see what version number your version of R is. If you do not have the latest version, see the steps above to install the latest version.

To install `quanteda`, type:

```
install.packages("quanteda", dependencies = TRUE)
```

Note that because we wrote `dependencies = TRUE`, this command also installed three other `quanteda` helper packages, which serve to extend the basic tools that are already inside `quanteda`. In the future, more of these helper packages can be expected to extend the main `quanteda` package even further. However, before these helper packages get an official release, we can already find them in development on GitHub.In this book, we will install two of them - `quanteda.classifiers`, which we will use for supervised learning methods, and `quanteda.dictionaries`, which we will use for dictionary analysis:

```
library(devtools)
install_github("quanteda/quanteda.classifiers", dependencies = TRUE)
install_github("kbenoit/quanteda.dictionaries", dependencies = TRUE)
install_github("quanteda/quanteda.corpora", dependencies = TRUE)
```

In addition to `quanteda` we then use the following packages:

```
install_github("mikegruz/kripp.boot", dependencies = TRUE)
install.packages("ca", dependencies = TRUE)
install.packages("combinat", dependencies = TRUE)
install.packages("DescTools", dependencies = TRUE)
install.packages("FactoMineR", dependencies = TRUE)
install.packages("factoextra", dependencies = TRUE)
install.packages("Factoshiny", dependencies = TRUE)
install.packages("Hmisc", dependencies = TRUE)
install.packages("httr", dependencies = TRUE)
install.packages("jsonlite", dependencies = TRUE)
install.packages("manifestoR", dependencies = TRUE)
install.packages("readr", dependencies = TRUE)
install.packages("readtext", dependencies = TRUE)
install.packages("reshape2", dependencies = TRUE)
install.packages("RTextTools", dependencies = TRUE)
install.packages("R.temis", dependencies = TRUE)
install.packages("rvest", dependencies = TRUE)
install.packages("seededlda", dependencies = TRUE)
install.packages("stm", dependencies = TRUE)
install.packages("tidyverse", dependencies = TRUE)
install.packages("topicmodels", dependencies = TRUE)
install.packages("magick", dependencies = TRUE)
```

```
install.packages("vader", dependencies = TRUE)
```

Some of these are specialised packages for text analysis, others for statistical estimation and visualisation. After installation, you will find these packages, as well as the `quanteda` and `devtools` packages, under the **Packages** tab in RStudio.

3.4 Problems, Bugs and Errors

It is not uncommon to get errors when typing a command in R. Errors often occur when you misspell the code or use the wrong code for the job at hand. In these cases, R will print a message (in red) telling you why it cannot do what you want it to do. Sometimes this message is quite clear, like telling you to install an extra package. Other times it is more complicated and requires some extra work. In these cases, there are four questions you should ask yourself:

1. Have I downloaded all the packages I need?
2. Are all my packages up to date?
3. Did I spell the commands correctly?
4. Is the data in the correct form or format?

If none of these provide a solution, you can always look online to see if others have encountered the same problem. Often, copying and pasting your error into a search engine will give you other examples, and usually a solution. A well-known place for solutions is Stack Overflow, where you can share your problem with others and see if someone can offer a solution. However, make sure you read through the problems that have already been posted to make sure you do not post the same problem twice. We have also compiled a list of problems encountered by people who have used this book, along with their solutions, in this spreadsheet.

Chapter 4

Importing Data

Before we can do any kind of analysis, we need to get our text into R. Here we look at five different ways of doing this: a) using .txt files, b) using .pdf files, c) using .csv files, d) using an API, and e) using web scraping techniques. Before we do that, though, let us first look at how R looks at text in the first place, and what some of the most basic things we can do with text there are.

4.1 Text in R

R sees any form of text as a vector consisting of different types of characters. In their simplest form, these vectors only have a single character in them. At their most complicated, they can contain many sentences or even whole stories. To see how many characters a vector has, we can use the `nchar` function:

```
vector1 <- "This is the first of our character vectors"
nchar(vector1)
```

```
## [1] 42
```

```
length(vector1)
```

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

This example also shows the logic of R. First, we assign the text we have to a certain object. We do so using the `<-` arrow. This arrow points from the text we have to the object R stores it in, which we here call `vector1`. We then ask R to give us the number of characters inside this object, which is 40 in this case. The `length` command returns something else, namely 1. This means that we have a single sentence, or word, in our object. If we want to, we can place more sentences inside our object using the `c()` option:

```
vector2 <- c("This is an example", "This is another", "And so we can go on.")
length(vector2)
```

```
## [1] 3
```

```
nchar(vector2)
```

```
## [1] 18 15 20
```

```
sum(nchar(vector2))
```

```
## [1] 53
```

Another thing we can do is extract certain words from a sentence. For this, we use the `substr()` function. With this function, R gives us all the characters that occur between two specific positions. So, when we want the characters between the 4th and 10th characters, we write:

```
vector3 <- "This is yet another sentence"
substr(vector3, 4, 10)
```

```
## [1] "s is ye"
```

We can also split a character vector into smaller parts. We often do this when we want to split a longer text into several sentences. To do so, we use the `strsplit` function:

```
vector3 <- "Here is a sentence - And a second"
parts1 <- strsplit(vector3, "-")
parts1
```

```
## [[1]]
## [1] "Here is a sentence " " And a second"
```

If we now look in the Environment window, we will see that R calls `parts1` a list. This is another type of object that R uses to store information. We will see it more often later on. For now, it is good to remember that lists in R can have many vectors (the layers of the list) and that in each of these vectors we can store many objects. Here, our list has only a single vector. To create a longer list, we have to add more vectors, and then join them together, again using the `c()` command:

```
vector4 <- "Here is another sentence - And one more"
parts2 <- strsplit(vector4, "-")
parts3 <- c(parts1, parts2)
```

We can now look at this new list in the Environment and check that it indeed has two elements. A further thing we can do is to join many vectors together. For this, we can use the `paste` function. Here, the `sep` argument defines how R will combine the elements:

```
fruits <- paste("oranges", "lemons", "pears", sep = "-")
fruits
```

```
## [1] "oranges-lemons-pears"
```

Note that we can also use this command that pastes objects that we made earlier together. For example:

```
sentences <- paste(vector3, vector4, sep = ".")
sentences
```

```
## [1] "Here is a sentence - And a second.Here is another sentence - And one more"
```

Finally, we can change the case (lowercase, uppercase) of the sentence. To do this, we can use `tolower` and `toupper`:

```
tolower(sentences)
```

```
## [1] "here is a sentence - and a second.here is another sentence - and one more"
```

```
toupper(sentences)
```

```
## [1] "HERE IS A SENTENCE - AND A SECOND.HERE IS ANOTHER SENTENCE - AND ONE MORE"
```

Again, we can also run the same command when we have more than a single element in our vector:

```
sentences2 <- c("This is a piece of example text", "This is another piece of example text")
toupper(sentences2)
```

```
## [1] "THIS IS A PIECE OF EXAMPLE TEXT"
## [2] "THIS IS ANOTHER PIECE OF EXAMPLE TEXT"
```

```
tolower(sentences2)
```

```
## [1] "this is a piece of example text"
## [2] "this is another piece of example text"
```

And that is it. As you can see, the options for text analysis in basic R are rather limited. This is why packages such as `quanteda` exist in the first place. Note though, that even `quanteda` uses the same logic of character vectors and combinations that we saw here.

4.2 Import .txt Files

The .txt format is one of the most common formats to store texts in. Not only are these files format-free, but they are also small in size, support a wide array of characters, and are readable on all platforms. To read them into R, we first specify where on our computer these files are. Here they are in a folder called **Texts** in our Working Directory. To tell R where this is, we first set our working directory (to see where your current Working Directory is, type `getwd()` into

the Console). Then, we add `/Texts` to this and save it as `txt_directory`. We can then use this to refer to this folder when we import the files using `readtext`:

```
library(readtext)
setwd("Your Working Directory")
txt_directory <- paste0(getwd(), "/Texts")
data_texts <- readtext(paste0(txt_directory, "*"), encoding = "UTF-8")
```

Note that now you not only have the texts in your environment but the object `txt_directory` as well. Note that this is nothing more than a string that we can use later on to prevent us from having to type the whole address of the folder.

4.3 Import .pdf Files

Apart from .txt, .pdf files are another common format for texts. Yet, as .pdf files contain a lot of information (tables, figures and graphs), besides the texts, we will have to "get out" the texts first. To do so, we first use the `pdftools` package to convert the .pdf files into .txt files, which we then read (as above) with `readtext`. Note that this only works if the .pdf files are *readable*. This means that we can select (and copy-paste) the text in them. Thus, `readtext` does not work with .pdf files where we cannot select the text. Often this happens when a .pdf is a scan or contains many figures. In such cases, you might have to use optical character recognition (OCR) (such as offered by the `tesseract` package) to generate the .txt files.

As with the .txt files above, here we will place our .pdf files in a folder called **PDF** in our Working Directory. Also, we have an (empty) folder called **Texts** where R will write the new .txt files to. We then tell R where these folder are:

```
library(pdftools)
library(readtext)

pdf_directory <- paste0(getwd(), "/PDF")
txt_directory <- paste0(getwd(), "/Texts")
```

Then, we ask R for a list of all the files in the .pdf directory. This is both to ensure that we are not overlooking anything and to tell R which files are in the folder. Here, setting `recurse=FALSE` means that we only list the files in the main folder and not any files that are in other folders in this main folder.

```
files <- list.files(pdf_directory, pattern = ".pdf", recursive = FALSE, full.names = TR

files
```

While we could convert a single document at a time, more often we have to deal with more than one document. To read all documents at once, we have to write a little function. This function does the following. Firstly, we tell R to

make a new function that we label `extract`, and as input give it an element we call `filename`. This file name is at this point an empty element, but to which we will later refer to the files we want to extract. Then, we ask R to print the file name to ensure that we are working with the right files while the function is running. In the next step, we ask R to try to read this file name using the `pdf_text` function and save the result as a file called `text`. Afterwards, we ask R to do so for each of the files that end on .pdf that are in the element `files`. Then, we have R write this text file to a new file. This file is the extracted .pdf in .txt form:

```
extract <- function(filename) {
  print(filename)
  try({
    text <- pdf_text(filename)
  })
  title <- gsub("(.*)/([^/]*).pdf", "\\2", filename)
  write(text, file.path(txt_directory, paste0(title, ".txt")))
}
```

We then use this function to extract all the .pdf files in the `pdf_directory` folder. To do so, we use a `for` loop. The logic of this loop is that for each individual `file` in the element `files`, we run the `extract` function we created. This will create an element called `file` for the file R is currently working on, and will create the .txt files in the `txt_directory`:

```
for(file in files) {
 extract(file)
}
```

We can now read the .txt files into R. To do so, we use `paste0(txt_directory, "*")` to tell `readtext` to look into our `txt_directory`, and read any file in there. Besides this, we need to specify the encoding. Most often, this is **UTF-8**, though sometimes you might find **latin1** or **Windows-1252** encodings. While `readtext` will convert all these to **UTF-8**, you have to specify the original encoding. To find out which one you need, you have to look into the properties of the .txt file.

Assuming our texts are in UTF-8 encoding, we run:

```
data_texts <- readtext(paste0(txt_directory, "*"), encoding = "UTF-8")
```

The result of this is a data frame of texts, which we can transform into a corpus for use in `quanteda` or keep as it is for other types of analyses.

4.4 Import .csv Files

We can also choose not to import the texts into R in a direct fashion, but import a .csv file with word counts instead. One way to generate these counts is by

using JFreq (Lowe, 2011). This stand-alone programme generates a .csv file where rows represent the documents and columns the individual words. The cells then contain the word counts for each word within each document. In addition, JFreq also allows for some basic pre-processing (though we would suggest you do this in R). Note that while JFreq is not under active maintenance, you can still find it at https://conjugateprior.org/software/jfreq/.

To use JFreq, open the programme and drag and drop all the documents you want to process into the window of the programme. Once you do this, the document file names will appear in the document window. Then, you can choose from several pre-processing options. Amongst these are options to make all words lower-case or remove numbers, currency symbols, or stop words. The latter are words that often appear in texts which do not carry important meaning. These are words such as 'and', 'or' and 'but'. As stop words are language-specific and often context-specific as well, we need to tell JFreq what words are stop words. We can do so by putting all the stop words in a separate .txt file and load it in JFreq. You can also find many lists of stop words for different languages on-line (see, for example this collection). Finally, we can apply a stemmer which reduces words such as 'Europe' and 'European' to a single 'Europ*' stem. JFreq allows us to use pre-defined stemmers by choosing the relevant language from a drop-down menu. Figure 4.1 shows JFreq while importing the .txt files of some electoral manifestos.

Note that here the encoding is UTF-8 while the locale is English (UK). Once we have specified all the options we want, we give a name for the output folder and press **Process**. Now we go to that folder we named and copy-paste the **data.csv** file into your Working Directory. In R, we then run the following:

```
data_manifestos <- read.csv("data.csv", row.names = 1, header = TRUE)
```

By specifying `row.names=1`, we store the information of the first column in the data frame itself. This column contains the names of the documents, and belongs to the object of the data frame and does not appear as a separate column. The same is true for `header=TRUE` which ensures that the first row gives names to the columns (in this case containing the words).

4.5 Import from an API

Another way to import texts is by using an Application Programming Interface (API). While comparable to web scraping, APIs are much more user friendly and communicate better with R. This makes it easier to download a large amount of data at once and import the results into R. There are APIs for many popular

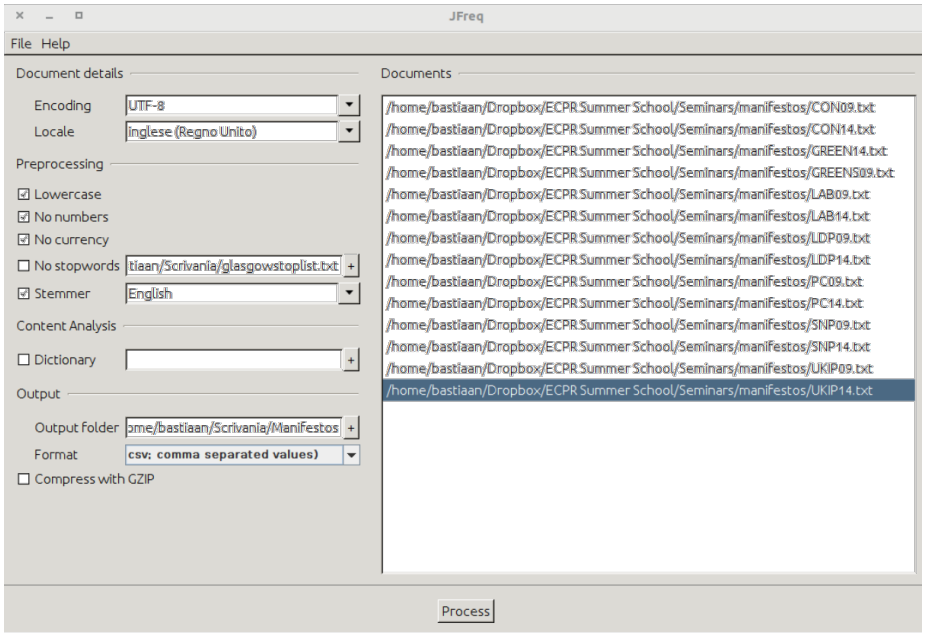


Figure 4.1: Overview of JFreq with several documents loaded

websites, such as Wikipedia, Twitter, YouTube, Weather Underground, The New York Times, the European Union and so on. Note, however, that you often, if not always, need to register before you can use an API. Moreover, social media platforms such as Facebook and Twitter have recently introduced restrictions in the use of their APIs that have limited researchers' ability to conduct critical scholarly research (Bruns, 2019). For instance, Facebook has taken steps in restricting access to their public APIs for research purposes. As such, free research on Facebook users' posts is no longer an option (Freelon, 2018; Perriam et al., 2020). Even more recently, Twitter (rebranded as 'X' in July 2023) has eliminated the free access to its API for third-party developers. At the time of writing, the 'Basic' subscription that costs $100 per month allows you to create a project to pull up to 10,000 Tweets.

While web scraping, in general, is easy with the `rvest` package, for the APIs you often need a specific package. For example, for Twitter there is the `rtweet` package, for Facebook `Rfacebook`, and `ggmap` for Google maps. Moreover, there are many APIs with associated R packages made by researchers for researchers, such as `manifestoR`, a package that provides researchers access to the corpus of the Manifesto Project (Merz et al. (2016)).

Let's look at an example using an API for the *New York Times*. If you look at the *New York Times*'s API page (https://developer.nytimes.com/), you will find that we use the API to extract information ranging from opinion articles to book reviews, movie reviews, and so on. In our example, we will use the API to extract a corpus of movie reviews that were originally published in the *New York Times*.

Before we start here, we first have to gain permission to use the API. For this, you need to register an account at the website and log in. Then, make a new **app** under: **https://developer.nytimes.com/my-apps** and ensure you select the movie reviews. Then, you can click on the new app to see your key under **API Keys**. It is this string of codes and letters you will have to place at the `[YOUR_API_KEY_HERE]` bit shown below.

Now, let us first load the necessary packages:

```
library(tidyverse)
library(httr)
library(jsonlite)
```

We can then build our request. As you can see on the site, the request requires us to give a search term (here we choose "love"). Also, we can set a time frame from which we want to extract the reviews:

```
reviews <- fromJSON("https://api.nytimes.com/svc/movies/v2/reviews/search.json?query=l
```

The result is a JSON object that you can see in the environment. While JSON (JavaScript Object Notation) is a generic way in which information is easy to share, and is thus often used, it is not in an ideal form. So, we change the JSON

information to a data frame using the following:

```
reviews_df <- fromJSON("https://api.nytimes.com/svc/movies/v2/reviews/search.json?query=love&oper
    flatten = TRUE) %>%
    data.frame()
```

You can now find all the information in the new `reviews_df` object, which also contains other useful information about each movie. As we can see, having a package makes things easier, though more limited.

4.6 Import using Web Scraping

If there is no specific API, we can also choose to scrape the website. The logic of web scraping is that we use the structure of the underlying HTML document to find and download the text we want. Note though that not all websites encourage (or even allow) scraping, which means that we need to have a look at their disclaimer beforehand. You can do this by either checking the website's **Terms and Conditions** page, or the **robots.txt** file that you can usually find appended at the home page (e.g. https://www.facebook.com/robots.txt).

One easy way to scrape a website is to search and see whether someone else has already built a tool that automates the webscraping process for the particular website that we are interested in. For instance, since Twitter has ended the free access to its API, we can look at places like Apify to find a suitable scraper for popular websites such as Twitter/X, Wikipedia, Instagram (to scrape data on public profiles), Google search, Google maps, Tiktok, Amazon (to scrape data on its products), and so on. Apify is not free, but registering an account for a free trial and using a month's subscription may be enough for a small project.

If you cannot find a ready-made scraper for your project, or if you cannot pay for such services, you can use `rvest` package to configure a webscraper of your own. One of the most popular sites to scrape is Wikipedia as it is very welcoming to web scraping and has pages with a clear structure (indeed, underlying sites such as WikiData are built with web scraping in mind). Here, let us take the following page on the Cold War as an example: https://en.wikipedia.org/wiki/Cold_War. If you have a quick look at the website you see that there is a lot of information on there, including figures, tables and the actual body of text. Here, we are only interested in the latter.

To begin with, we store the address of the webpage in an object and ask R to read it for us:

```
url <- "https://en.wikipedia.org/wiki/Cold_War"
url <- rvest::read_html(url)
```

We now have the HTML page (though R will not yet show it), but we before we can do anything further, we have to figure out how to get the content we want. To do so, we have to inspect the HTML document to find the right element. The

easiest way to do this is in the browser. To do so, open the page and use **Cmd + Shift + C** to open up Developer Tools. In the tools that then open, we can hover over the main body of text on the page and see which element belongs to it. Here, we find that the main body of text is stored in an element called **mw-content-text** and that within that the individual paragraphs are stored in elements labelled **

**. This latter designation is a standard reference in HTML to individual paragraphs and are what we are after here.

We now ask R to take the HTML extracted from the URL, look only for the nodes labelled **

**, extract the text from it, place the results into a data frame, and then into a character vector:

```
library(rvest)
```

```
##
## Attaching package: 'rvest'
```

```
## The following object is masked from 'package:readr':
##
##     guess_encoding
```

```
data_coldwar <- url %>%
    html_nodes("p") %>%
    html_text2() %>%
    as.data.frame()
data_coldwar <- data_coldwar$.
data_coldwar <- data_coldwar[-c(1, 2)]
```

Note that for the text we can choose to use either `html_text` or `html_text2`. The difference between the two is that while the former gives us the raw underlying text, the latter gives us the text as it looks like in the browser, which is what we opt for here. Now, looking at the resulting data-frame, we have 207 observations representing the 207 paragraphs of the text (with the first two empty, so we removed them here). We can then use this as an input for any further analysis (and we will get back to this later). If you would like to learn more about web scraping in the context of quantitative text analysis have a look at the textbook by Munzert et al. (2014).

Chapter 5

Reliability and Validity

As noted earlier, as there is no reality we are measuring, there is no way for us to ensure that what we are doing is correct. In other words - everything we do is probably wrong in one way or the other - but it might be useful. So how do we ensure that? To begin with, we should ensure that everything we are do should is both *reliable* and *valid*. Reliability here refers to consistency, that is, the degree to which we get similar results whenever we apply a measuring instrument to measure a given concept. This is similar to the concept of *Replicability*. Validity, on the other hand, refers to unbiasedness, that is, the degree to which our measure does really measure the concept that we intend to measure. In other words, validity looks at whether the measuring instrument that we are using is objective.

Carmines & Zeller (1979) distinguish among three types of validity. *Content Validity*, which refers to whether our measure represents all facets of the construct of interest; *Criterion Validity*, which looks at whether our measure correlates with other measures of the same concept, and *Construct Validity*, which looks at whether our measure behaves as expected within a given theoretical context. We should also say here, that these three types of validity are not interchangeable. In the ideal case, one has to prove that their results pass all three validity tests. In the words of Grimmer & Stewart (2013): "Validate, validate, validate"!

Krippendorff (2018) distinguishes among three types of reliability. The first is *Stability*, which he considers as the weakest form of coding reliability. This we can measure by having our text coded more than once by the same coder. The higher the differences between the different codings, the lower our reliability. The second is *Reproducibility*, which reflects the agreement among independent coders on the same text. Finally, the third is *Accuracy*, which he considers as the strongest form of coding reliability. For this, we look at the agreement between coders, as with reproducibility, but now use a given standard. Yet, as benchmarks are rare, reproducibility is often the highest form we can go for.

This agreement between coders we need for this is also known as inter-coder agreement, which we will look at next.

5.1 Inter-Coder Agreement

Hayes & Krippendorff (2007, p. 79) argue that a good measure of the agreement should at least address five criteria. The first is that it should apply to many coders, and not only two. Also, when we use the method for more coders, there should be no difference in how many coders we include. The second is that the method should only take into account the actual number of categories the coders used and not all that were available. This as while the designers designed the coding scheme on what they thought the data would look like, the coders use the scheme based on what the data is. Third, it should be numerical, meaning that we can use it to make a scale between 0 (absence of agreement) and 1 (perfect agreement). Fourth, it should be appropriate for the level of measurement. So, if our data is ordinal or nominal, we should not use a measure that assumes metric data. This ensures that the metric uses all the data and that it does not add or not use other information. Fifth, we should be able to compute (or know), the sampling behaviour of the measure.

With these criteria in mind, we see that popular methods, such as % agreement or Pearson's r, can be misleading. Especially for the latter - as it is a quite popular method - this often leads to problems, as Krippendorff (2018) shows:

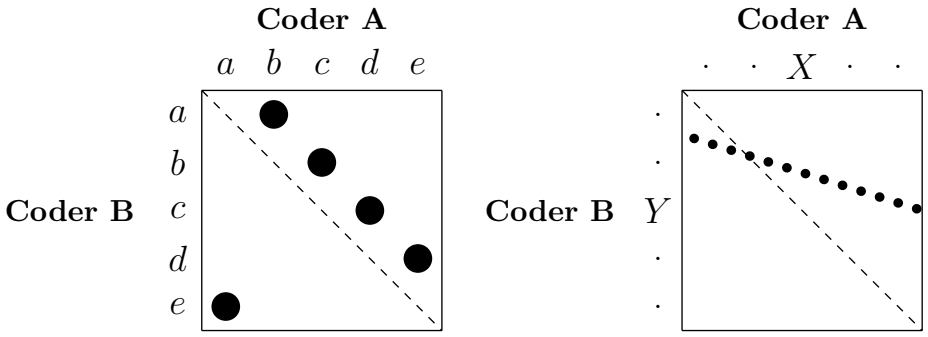


Figure 5.1: Perfect Disagreement between Two Coders [@Krippendorff2004a]

Here, Figure 5.1 shows, on the left, two coders: A and B. The dots in the figure show the choices both coders made, while the dotted line shows the line of perfect agreement. If a dot is on this line, it means that both Coder A and Coder B

made the same choice. In this case, they disagreed in all cases. When Coder A chose *a*, Coder B chose *e*, when Coder A chose *b*, Coder B chose *a*, and so on. Yet, when we would calculate Pearson's r for this, we would find a result as shown on the right-hand side of the figure. Seen this way, the agreement between both coders does not seem a problem at all. The reason for this is that Pearson's r works with the distances between the categories *without* taking into account their location. So, for a positive relationship, the only thing Pearson's r requires is that for every increase or decrease for one coder, there is a similar increase or decrease for the other. This happens here with four of the five categories. The result is thus a high Pearson's r, though the actual agreement should be 0.

Pearson's r thus cannot fulfil all our criteria. A measure that can is Krippendorff's α (Krippendorff, 2018). This measure can not only give us the agreement we need, but can also do so for nominal, ordinal, interval, and ratio level data, as well as data with many coders and missing values. Besides, we can compute 95% confidence intervals around α using bootstrapping, which we can use to show the degree of uncertainty around our reliability estimates.

Despite this, Krippendorff's α is not free of problems. One main problem occurs when coders agree on only a few categories and use these categories a considerable number of times. This leads to an inflation of α, making it is higher than it should be (Krippendorff, 2018), as in the following example:

Coder B

Coder A	**0**	**1**	**2**	
0	80	0	1	81
1	1	0	1	2
2	0	0	3	3
	81	0	5	86

$\alpha = 0.686$

1st Distinction

	0	**1&2**	
0	80	1	81
1&2	1	4	5
	81	5	86

$\alpha = 0.789$

2nd Distinction

	1	**2**	
1	0	1	1
2	0	3	3
	0	4	4

$\alpha = 0.000$

Figure 5.2: Possible Inflation of Krippendorff's‘ α [@Krippendorff2004a]

Here, the top left of the figure shows coders A and B, who have to code into three categories: 0, 1, or 2. In this example, categories 1 and 2 carry a certain meaning, while category 0 means that the coders did not know what to assign the case to. Of the 86 cases, both coders code 80 cases in the 0 category. This means that there are only 6 cases on which they can agree or disagree about a code that carries some meaning. Yet, if we calculate α, the result - 0.686 - takes

into account all the categories. One solution for this is to add up categories 1 and 2, as the figure in the middle shows. Here, the coders agree in 84 of the 86 cases (on the diagonal line) and disagree in only 2 of them. Calculating α now shows that it would increase to 0.789. Finally, we can remove the 0 category and again view 1 and 2 as separate categories (as the most right-hand figure shows). Yet, the result of this is quite disastrous. While the coders agree in 3 of the 4 cases, the resulting α equals 0.000, as coder B did not use category 1 at all.

Apart from these issues, Krippendorff's α is a stable and useful measure. A value of $\alpha = 1$ indicates perfect reliability, while a value of $\alpha = 0$ indicates the absence of reliability. This means that if $\alpha = 0$, there is no relationship between the values. It is possible for $\alpha < 0$, which means that the disagreements between the values are larger than they would be by chance and are systematic. As for thresholds, Krippendorff (2018) proposes to use either 0.80 or 0.67 for results to be reliable. Such low reliability often has many causes. One thing might be that the coding scheme is not appropriate for the documents. This means that coders had categories that they had no use for, and lacked categories they needed. Another reason might be that the coders lacked training. Thus, they did not understand how to use the coding scheme or how the coding process works. This often leads to frustration on part of the coders, as in these cases the process often becomes time-consuming and too demanding to carry out.

To calculate Krippendorff's α, we can use the following software:

- KALPHA custom dialogue (SPSS)
- **kalpha** user-written package (Stata)
- KALPHA macro (SAS)
- `kripp.alpha` command in `kripp.boot` package (R) - amongst others

Let us try this in R using an example. Here, we will look at the results of a coding reliability test where 12 coders assigned the sentences of the 1997 European Commission work programme in the 20 categories of a policy areas coding scheme. We can find the results for this on GitHub. To get the data, we tell R where to find it, then to read that file as a .csv file and write it to a new object:

```
library(readr)

urlfile = "https://raw.githubusercontent.com/SCJBruinsma/qta-files/master/reliability_r
reliability_results <- read_csv(url(urlfile), show_col_types = FALSE)
```

Notice that in the data frame we created, the coders are in the columns and the sentences in the rows. As the `kripp.boot` package requires it to be the other way around and in matrix form, we first transpose the data and then place it in a matrix. Finally, we run the command and specify we want the nominal version:

```
library("kripp.boot")
```

```
reliability_results_t <- t(reliability_results)
reliability <- as.matrix(reliability_results_t)
kalpha <- kripp.boot(reliability, iter = 1000, method = "nominal")
kalpha$mean.alpha
```

Note also that `kripp.boot` is a GitHub package. You can still calculate the value (but without the confidence interval) with another package:

```
library("DescTools")

reliability_results_t <- t(reliability_results)
reliability <- as.matrix(reliability_results_t)
kalpha <- KrippAlpha(reliability, method = "nominal")
kalpha$value
```

As we can see, the results point out that the agreement among the coders is 0.634 with an upper limit of 0.650 and a lower limit of 0.618 which is short of Krippendorff's cut-off point of 0.667.

5.2 Visualizing Quality

Lamprianou (2020) notes that existing reliability indices may mask coding problems and that the reliability of coding is not stable across coding units (as illustrated in the example given for Krippendorff's α above). To investigate the quality of coding he proposes using social network analysis (SNA) and exponential random graph models (ERGM). Here, we illustrate a different approach, based on the idea of sensitivity analysis.

The idea of this is to compare the codings of each coder against all others (and also against a benchmark or a gold standard). To do so, we need to bootstrap the coding reliability results to create an uncertainty measure around each coder's results, following the approach proposed by Benoit et al. (2009). The idea here is to use a non-parametric bootstrap for the codings of each coder (using 1000 draws with replacement) at the category level and then calculate the confidence intervals. Their width then depends on both the number of sentences coded by each coder (n) in each category and the number of coding categories that are not empty. Thus, larger documents and fewer empty categories result in narrower confidence intervals, while a small number of categories leads to wider intervals (Lowe & Benoit, 2011).

To start, the first thing we do is load the packages we need into R:

```
library(Hmisc)
```

```
## Warning: package 'Hmisc' was built under R version 4.3.3
```

```
library(combinat)
library(readr)
```

In the following example we perform the sensitivity analysis on the coded sentences of the 1997 European Commission work programme, as seen earlier. Yet here, row represents a coder, and each column represents a coding category (`c0` to `c19`). In each cell, we see the number of sentences that each coder coded in each category, with the column `n` giving the sum of each row:

```
urlfile <- "https://raw.githubusercontent.com/SCJBruinsma/qta-book/main/data_uncertaint
data_uncertainty <- read_csv(url(urlfile))
```

We then tell R how many coders we have. As this number is equal to the number of rows we have, we can get this number using the `nrow` command. We also specify the number of bootstraps we want to carry out (1000) and transform our data frame into an array. We do the latter as R needs the data in this format later on:

```
nman <- nrow(data_uncertainty)
nrepl <- 1000
manifBSn <-
  manifBSnRand <- array(
    as.matrix(data_uncertainty[, 2:21]),
    c(nman, 20, nrepl + 1),
    dimnames = list(1:nman, names(data_uncertainty[, 2:21]),
                    0:nrepl)
  )
```

We then bootstrap the sentence counts for each coder and compute percentages for each category using a multinomial draw. First, we define `p`, which is the proportion of each category over all the coders. Then, we input this value together with the total number of codes `n` into the `rmultinomial` command, which gives the random draws. As we want to do this a 1000 times, we place this command into a `for` loop:

```
n <- data_uncertainty$n
p <- manifBSn[, , 1] / n

for (i in 1:nrepl) {
  manifBSn[, , i] <- rmultinomial(n, p)
}
```

With this data, we can then ask R to compute the quantities of interest. These are standard errors for each category, as well as the percentage coded for each category:

```
c0SE  <- apply(manifBSn[, "c0", ]  / n * 100, 1, sd)
c01SE <- apply(manifBSn[, "c01", ] / n * 100, 1, sd)
c02SE <- apply(manifBSn[, "c02", ] / n * 100, 1, sd)
c03SE <- apply(manifBSn[, "c03", ] / n * 100, 1, sd)
```

```
c04SE <- apply(manifBSn[, "c04", ] / n * 100, 1, sd)
c05SE <- apply(manifBSn[, "c05", ] / n * 100, 1, sd)
c06SE <- apply(manifBSn[, "c06", ] / n * 100, 1, sd)
c07SE <- apply(manifBSn[, "c07", ] / n * 100, 1, sd)
c08SE <- apply(manifBSn[, "c08", ] / n * 100, 1, sd)
c09SE <- apply(manifBSn[, "c09", ] / n * 100, 1, sd)
c10SE <- apply(manifBSn[, "c10", ] / n * 100, 1, sd)
c11SE <- apply(manifBSn[, "c11", ] / n * 100, 1, sd)
c12SE <- apply(manifBSn[, "c12", ] / n * 100, 1, sd)
c13SE <- apply(manifBSn[, "c13", ] / n * 100, 1, sd)
c14SE <- apply(manifBSn[, "c14", ] / n * 100, 1, sd)
c15SE <- apply(manifBSn[, "c15", ] / n * 100, 1, sd)
c16SE <- apply(manifBSn[, "c16", ] / n * 100, 1, sd)
c17SE <- apply(manifBSn[, "c17", ] / n * 100, 1, sd)
c18SE <- apply(manifBSn[, "c18", ] / n * 100, 1, sd)
c19SE <- apply(manifBSn[, "c19", ] / n * 100, 1, sd)

per0  <- apply(manifBSn[, "c0", ]  / n * 100, 1, mean)
per01 <- apply(manifBSn[, "c01", ] / n * 100, 1, mean)
per02 <- apply(manifBSn[, "c02", ] / n * 100, 1, mean)
per03 <- apply(manifBSn[, "c03", ] / n * 100, 1, mean)
per04 <- apply(manifBSn[, "c04", ] / n * 100, 1, mean)
per05 <- apply(manifBSn[, "c05", ] / n * 100, 1, mean)
per06 <- apply(manifBSn[, "c06", ] / n * 100, 1, mean)
per07 <- apply(manifBSn[, "c07", ] / n * 100, 1, mean)
per08 <- apply(manifBSn[, "c08", ] / n * 100, 1, mean)
per09 <- apply(manifBSn[, "c09", ] / n * 100, 1, mean)
per10 <- apply(manifBSn[, "c10", ] / n * 100, 1, mean)
per11 <- apply(manifBSn[, "c11", ] / n * 100, 1, mean)
per12 <- apply(manifBSn[, "c12", ] / n * 100, 1, mean)
per13 <- apply(manifBSn[, "c13", ] / n * 100, 1, mean)
per14 <- apply(manifBSn[, "c14", ] / n * 100, 1, mean)
per15 <- apply(manifBSn[, "c15", ] / n * 100, 1, mean)
per16 <- apply(manifBSn[, "c16", ] / n * 100, 1, mean)
per17 <- apply(manifBSn[, "c17", ] / n * 100, 1, mean)
per18 <- apply(manifBSn[, "c18", ] / n * 100, 1, mean)
per19 <- apply(manifBSn[, "c19", ] / n * 100, 1, mean)
```

We then bind all these quantities together in a single data frame:

```
dataBS <- data.frame(cbind(data_uncertainty[, 1:22], c0SE, c01SE, c02SE, c03SE, c04SE,
    c05SE, c06SE, c07SE, c08SE, c09SE, c10SE, c11SE, c12SE, c13SE, c14SE, c15SE,
    c16SE, c17SE, c18SE, c19SE, per0, per01, per02, per03, per04, per05, per06, per07,
    per08, per09, per10, per11, per12, per13, per14, per15, per16, per17, per18,
    per19))
```

While we can now inspect the results by looking at the data, it becomes more clear when we visualise this. While R has some inbuilt tools for visualisation (in the `graphics` package), these tools are rather crude. Thus, here we will use the `ggplot2` package, which extends our options, and which has an intuitive structure:

```
library(ggplot2)
```

First, we make sure that the variable `coderid` is a factor and make sure that it is in the right order:

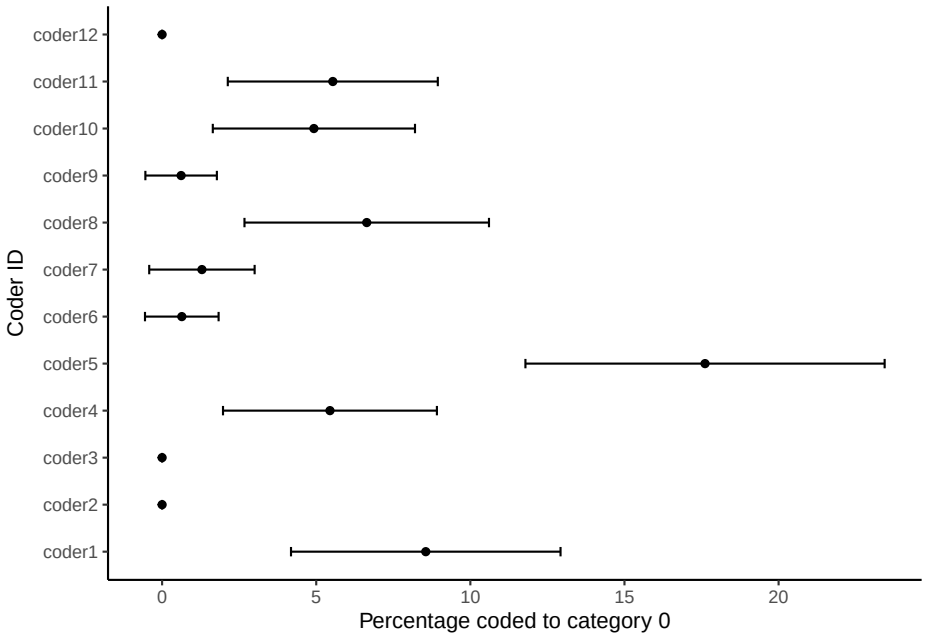
```
dataBS$coderid <- as.factor(dataBS$coderid)
dataBS$coderid <- factor(dataBS$coderid, levels(dataBS$coderid)[c(1, 5:12, 2:4)])
```

Then, we calculate the 95% confidence intervals for each category. We do so using the percent of each category and the respective standard error, and add these values to our data-set:

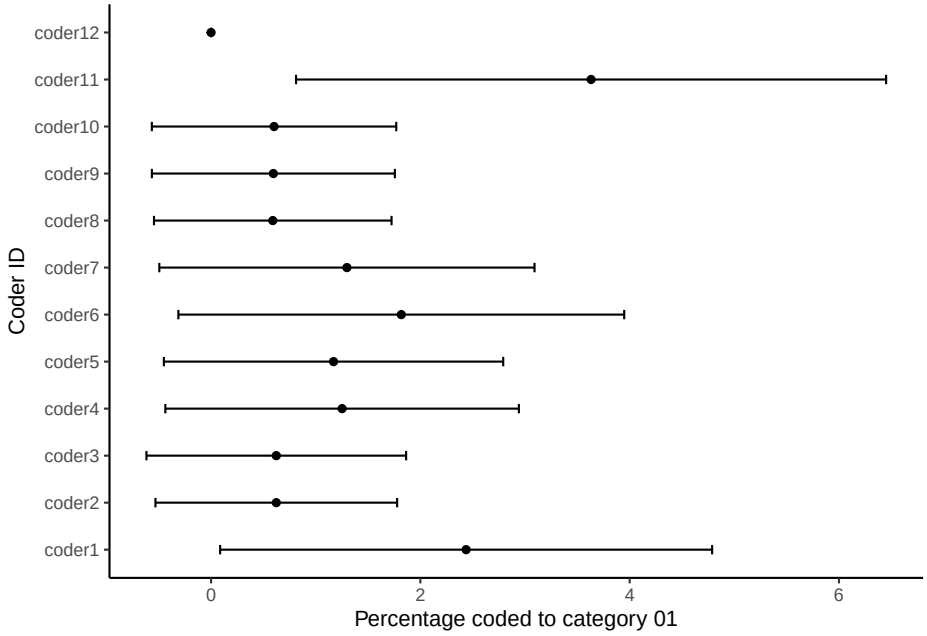
```
c0_lo <- per0 - (1.96 * c0SE)
c0_hi <- per0 + (1.96 * c0SE)
c01_lo <- per01 - (1.96 * c01SE)
c01_hi <- per01 + (1.96 * c01SE)
c02_lo <- per02 - (1.96 * c02SE)
c02_hi <- per02 + (1.96 * c02SE)

dataBS <- cbind(dataBS, c0_lo, c0_hi, c01_lo, c01_hi, c02_lo, c02_hi)
```

Finally, we generate the graphs for each individual category:

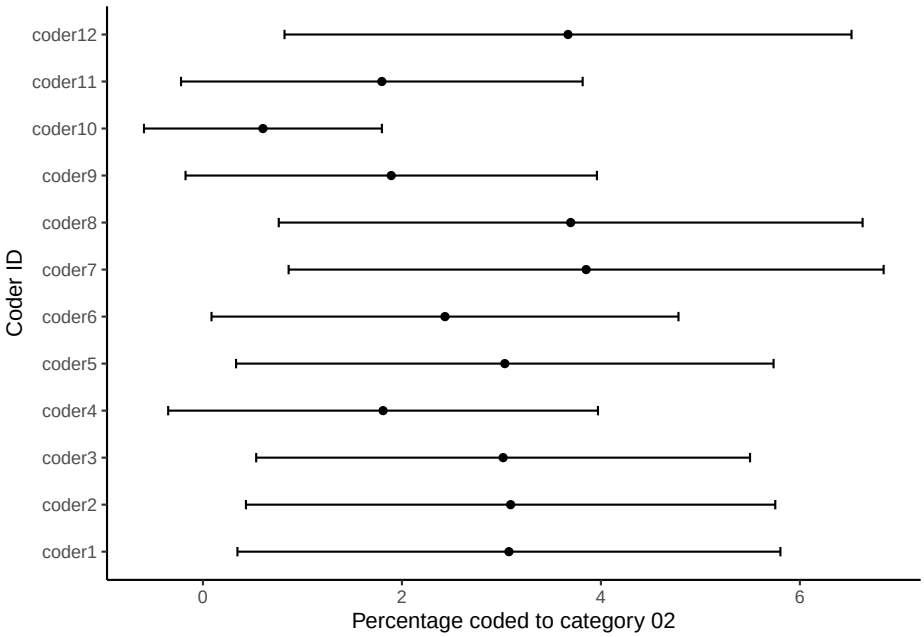
```
ggplot(dataBS,aes(per0, coderid))+
 geom_point() +
 geom_errorbarh(aes(xmax = c0_hi, xmin = c0_lo),height = .2)+
 xlab("Percentage coded to category 0")+
 ylab("Coder ID")+
 theme_classic()
```




```
ggplot(dataBS,aes(per01, coderid))+
 geom_point() +
 geom_errorbarh(aes(xmax = c01_hi, xmin = c01_lo),height = .2)+
 xlab("Percentage coded to category 01")+
 ylab("Coder ID")+
 theme_classic()
```




```
ggplot(dataBS,aes(per02, coderid))+
 geom_point() +
 geom_errorbarh(aes(xmax = c02_hi, xmin = c02_lo),height = .2)+
 xlab("Percentage coded to category 02")+
 ylab("Coder ID")+
 theme_classic()
```



Each figure shows the percentage that each of the coders coded in the respective category of the coding scheme. We thus use the confidence intervals around the estimates to look at the degree of uncertainty around each estimate. We can read the plots by looking if the dashed line is within the confidence intervals for each coder. The more the coders deviate from the benchmark or standard, the less likely it is that they understood the coding scheme in the same way. It also means that it is more likely that a coder would have coded the work programme much different from the benchmark coder. Thus, such a sensitivity analysis is like having a single reliability coefficient for each coding category.

Chapter 6

Preliminaries

Now that we have loaded our texts into R, it is time to understand *what* our texts are about, who their authors are, and what we can expect to find in them. Here we will look at three different techniques for doing this: keywords-in-context, visualisations and text statistics. Before that, however, we will have a brief look at the idea of the *corpus*, as it is central to the idea of how `quanteda` works.

6.1 The Corpus

Within `quanteda`, the main way to store documents is in the form of a `corpus` object. This object contains all the information that comes with the texts and does not change during our analysis. Instead, we make copies of the main corpus, convert them to the type we need and run our analyses on them. The advantage of this is that we can always go back to our original data.

There are various ways to make a corpus. One we already saw in the 4 chapter were we used `readtext` to generate a data frame, with both a variable that gave us a **doc_id** and another that gave us the text. As `readtext` and `quanteda` work close together, we can directly change this to a corpus. Alternatively, we can take a character vector, where each element of the vector is taken as an individual document. If the vector is named, those names will be used as document names - if not, new ones are generated.

Here, let us go back to the Cold War page we scraped earlier and use the resulting text from that as our input:

```
url <- "https://en.wikipedia.org/wiki/Cold_War"
url <- rvest::read_html(url)
data_coldwar <- url %>%
    html_nodes("p") %>%
    html_text2() %>%
```

```
    as.data.frame()
data_coldwar <- data_coldwar$.
data_coldwar <- data_coldwar[-c(1, 2)]
```

Note that the resulting `data_coldwar` vector is unnamed, so `quanteda` will generate those names for us. Now, we simply put this into the `corpus` command:

```
library(quanteda)
```

```
## Warning: package 'quanteda' was built under R version 4.3.3
```

```
## Package version: 4.0.2
## Unicode version: 15.1
## ICU version: 74.1
```

```
## Parallel computing: 8 of 8 threads used.
```

```
## See https://quanteda.io for tutorials and examples.
```

```
data_corpus <- corpus(data_coldwar)
```

Apart from importing texts ourselves, `quanteda` contains several corpora as well. Here, we use one of these, which contains the inaugural speeches of all the US Presidents. For this, we first have to load the main package and then load the data into R:

Now we have our corpus, we can start with the analysis. As noted, we try not to carry out any analysis on the corpus itself. Instead, we keep it as it is and work on its copies. Often, this means transforming the data into another shape. One of the more popular shapes is the *data frequency matrix* (dfm). This is a matrix that contains the documents in the rows and the word counts for each word in the columns.

Before we can do so, we have to split up our texts into unique words. To do this, we first have to construct a `tokens` object. In the command that we use to do this, we can specify how we want to split our texts (here we use the standard option) and how we want to clean our data. For example, we can specify that we want to convert all the texts into lowercase and remove any numbers and special characters.

```
data_tokens <- tokens(
 data_corpus,
 what = "word",
 remove_punct = TRUE,
 remove_symbols = TRUE,
 remove_numbers = TRUE,
 remove_url = TRUE,
 remove_separators = TRUE,
 split_hyphens = FALSE,
 include_docvars = TRUE,
```

```
 padding = FALSE,
 verbose = TRUE
)
```

We can also remove certain stopwords so that words like "and" or "the" do not influence our analysis too much. We can either specify these words ourselves or we can use a list that is already present in R. To see this list, type `stopwords("english")` in the console. Stopwords for other languages are also available (such as German, French and Spanish). There are even more stopwords in the `stopwords` package, which works well with `quanteda`. For now, we will use the English ones. As all the stopwords here are lower-case, we will have to lower case our words as well. Also notice that we also do this for any acronyms in our text (so, "NATO" will become "nato"):

```
data_tokens <- tokens_tolower(data_tokens,
                              keep_acronyms = FALSE)
data_tokens <- tokens_select(data_tokens,
                             stopwords("english"),
                             selection = "remove")
```

Then, we can construct our dfm:

```
data_dfm <- dfm(data_tokens)
```

6.2 Keywords in Context

One simple - but effective - way to learn more about our texts is by looking at keywords-in-context (kwic). Here, we look at with which other words a certain word appears in our texts. This is also known as looking at the *concordance* of our text. To do so is easy with our tokens data frame. Let's take all those words that start with 'secur' and look at which three words occur before and after this word. We can then run:

```
kwic_output <- kwic(data_tokens, pattern = "secur*", valuetype = "glob", window = 3)
```

In the outputted object, we find a column labelled `pre` and another labelled `post`. These refer to the words that came either before or after the word 'secur*'. We can easily take these out and combine them:

```
text_pre <- kwic_output$pre
text_post <- kwic_output$post
text_word <- kwic_output$keyword
text <- as.data.frame(paste(text_pre, text_word, text_post))
```

We then combine this information with the name of the document it came from so that we know which text the word is from:

```
extracted <- cbind(kwic_output$docname, text)
names(extracted) <- c("docname", "text")
head(extracted)
```

```
##   docname                                                          text
## 1  text10             making allowances peace security ushering period détente
## 2  text27 establishment maintenance post-war security scholars contend western
## 3  text27        western allies desired security system democratic governments
## 4  text27  churchill's mainly centered securing control mediterranean ensuring
## 5  text34    peace enforcement capacity security council effectively paralyzed
## 6  text44        leaders establishing secret security force prevent subversion
```

6.3 Visualisations and Descriptives

Another thing we can do is generate various visualisations to understand our data. One interesting thing can be to see which words occur most often. We can do this using the `topfeatures` function. For this, we first have to save the 50 most frequently occurring words in our texts (note that there is also the `textstat_frequency` function in the `quanteda.textstats` helper package that can do this):

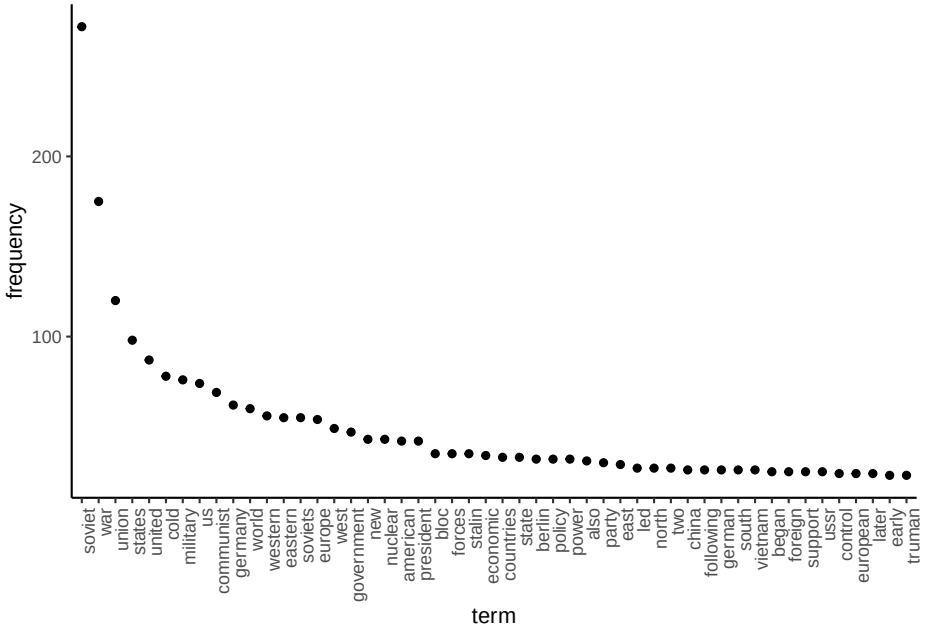
```
features <- topfeatures(data_dfm, 50)
```

We then have to transform this object into a data frame, and sort it by decreasing frequency:

```
features_plot <- data.frame(list(term = names(features),frequency = unname(features)))
features_plot$term <- with(features_plot, reorder(term, -frequency))
```

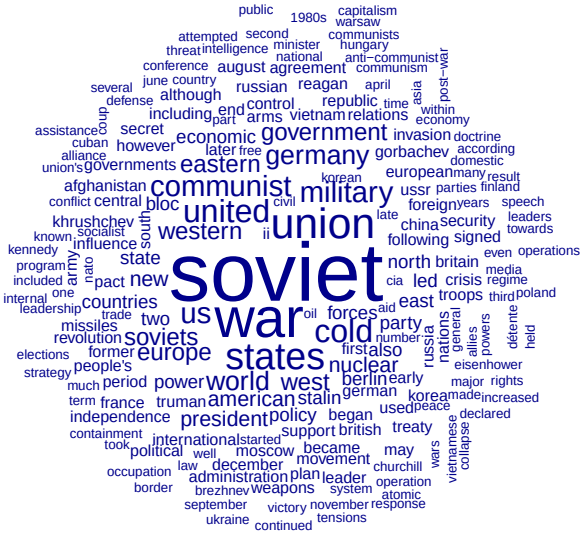
Then we can plot the results:

```
library(ggplot2)
ggplot(features_plot) +
 geom_point(aes(x=term, y=frequency)) +
 theme_classic()+
 theme(axis.text.x=element_text(angle=90, hjust=1))
```



We can also generate word clouds. As these show all the words we have, we will trim our dfm first to remove all those words that occurred less than 30 times. We can do this with the `dfm_trim` function. Then, we can use this trimmed dfm to generate the word cloud:

```
library(quanteda.textplots)

wordcloud_dfm_trim <- dfm_trim(data_dfm, min_termfreq = 10)
textplot_wordcloud(wordcloud_dfm_trim)
```



6.4 Text Statistics

Finally, `quanteda` also allows us to calculate quite some textual statistics. These are all collected in the `quanteda.textstats` helper package. Here, we will look at several of them, starting with a simple overview of our corpus in the terms of a summary. This tells us the number of characters, sentences, tokens, etc. for each of the texts:

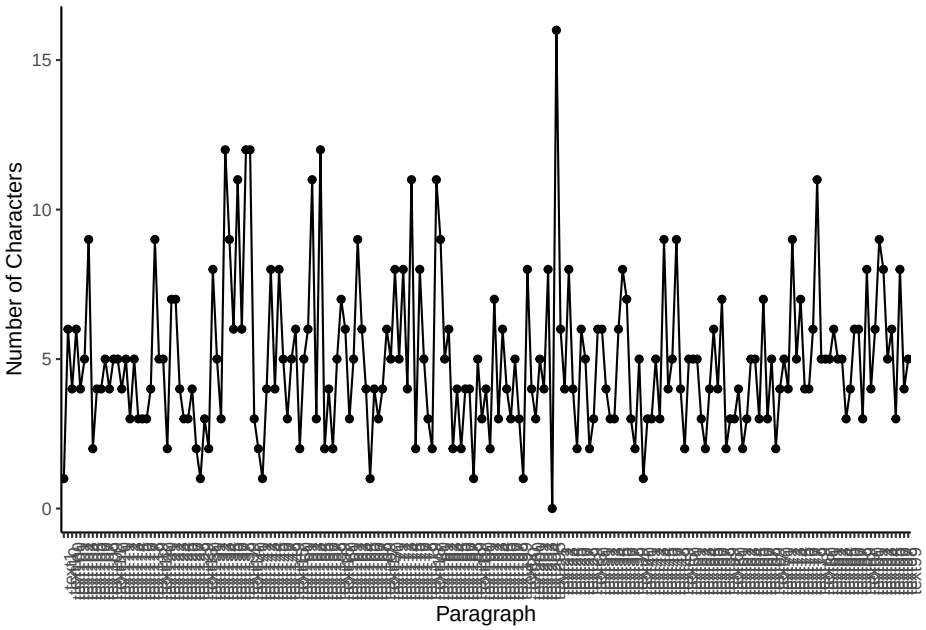
```
library(quanteda.textstats)
```

```
## Warning: package 'quanteda.textstats' was built under R version 4.3.3
```

```
corpus_summary <- textstat_summary(data_corpus)
```

If we want, we can then use this data to make some simple graphs telling us various things about the texts in our corpus. As an example, let's look at the number of sentences in the various paragraphs:

```
ggplot(data=corpus_summary, aes(x=document, y=sents, group=1)) +
 geom_line()+
 geom_point()+
 ylab("Number of Characters")+
 xlab("Paragraph")+
 theme_classic()+
 theme(axis.text.x = element_text(angle = 90))
```



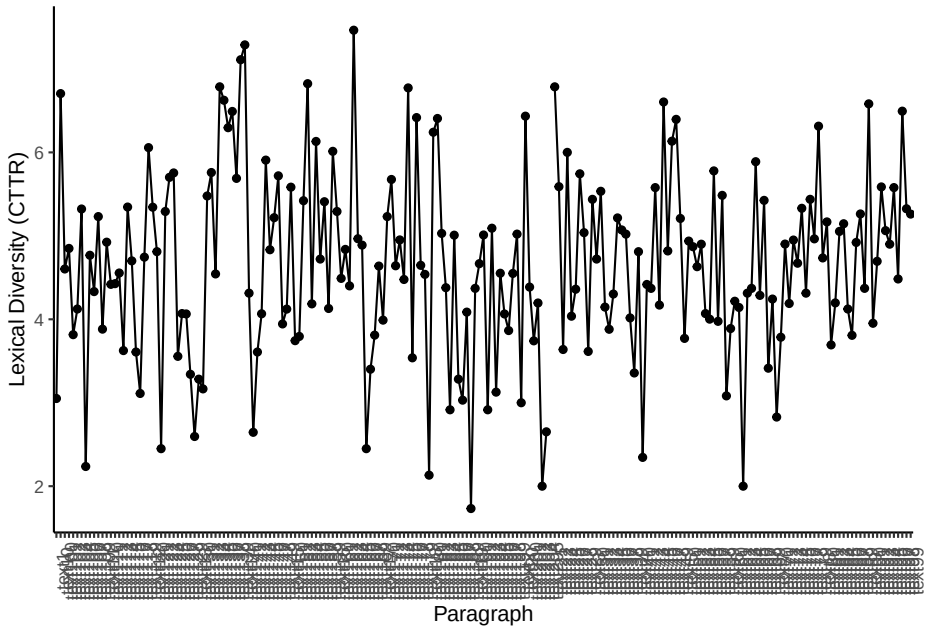
Other things we can look at are the readability and lexical diversity of the texts. The former one of these refers to how readable a text is (i.e. how easy or difficult it is to read), while the latter tells us how many different types of words there are in the texts and thus how *diverse* the text is in word choice and use. Given that there are many ways to calculate both metrics, please have a look at the help file to see which one works best for you. Here, we will use the most popular:

```
corpus_readability <- textstat_readability(data_corpus, measure = "Flesch.Kincaid")
corpus_lexdiv <- textstat_lexdiv(data_tokens, measure = "CTTR")
```

As before, we can plot this data in a graph to see how lexical diversity developed over the course of the article:

```
ggplot(data=corpus_lexdiv, aes(x=document, y=CTTR, group=1)) +
 geom_line()+
 geom_point()+
 ylab("Lexical Diversity (CTTR)")+
 xlab("Paragraph")+
 theme_classic()+
 theme(axis.text.x = element_text(angle = 90))
```

```
## Warning: Removed 1 row containing missing values or values outside the scale
## range (`geom_point()`).
```



Another thing we can do is look at the similarities and distances between documents. With this, we can answer questions such as: how *different* are these documents from each other? And if different (or similar), how different (or similar)? The idea is that the larger the similarity is, the smaller the distance is as well. A good way to understand the idea of similarity is to consider how many operations you need to perform to change one text into the other. The more "replace" options you have to carry out, the more different the text. As for the distances, it is best to consider the texts as having positions on a Cartesian plane (with positions based on their word counts). The distance between these two points (either Euclidean, Manhattan or other) is then the distance between the texts.

Let's start with a look at these similarities (note again that there are many different methods to calculate this):

```
corpus_similarties <- textstat_simil(data_dfm, method = "correlation", margin = "docume
corpus_similarties <- as.data.frame(corpus_similarties)
```

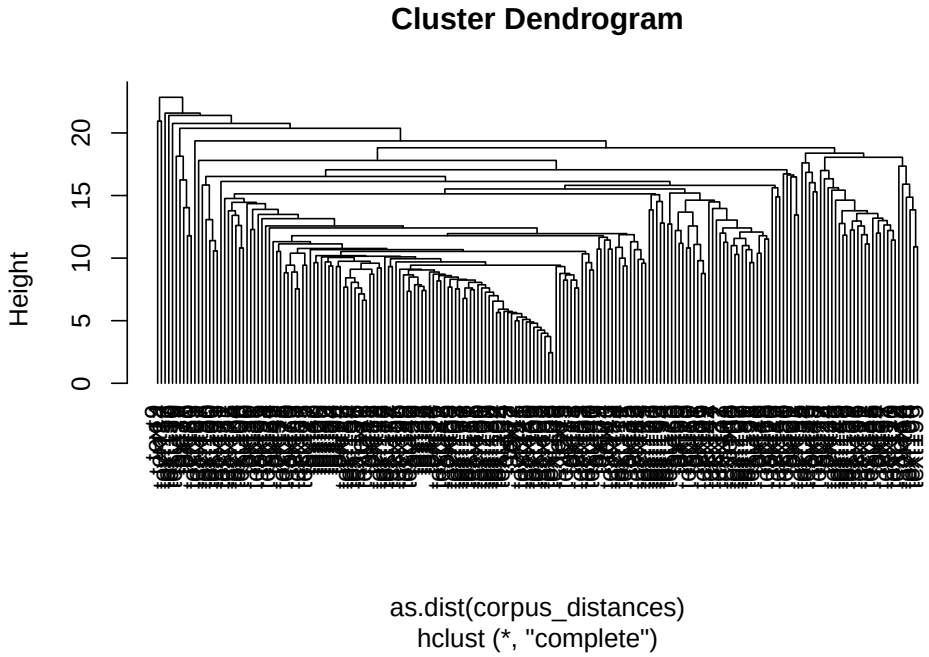
Note that while we look here at the documents, we could also look at individual words (set `margin="features`). For now, let us look at the distances between the documents, choosing the Euclidean distance between the documents as our metric:

```
corpus_distances <- textstat_dist(data_dfm, margin = "documents", method = "euclidean")
corpus_distances_df <- as.data.frame(corpus_distances)
```

If we want to, we can even convert this data into a dendrogram. We do this by

taking the information on the distances out of the `corpus_distances` object, make them into a triangular matrix, and plot them:

```
plot(hclust(as.dist(corpus_distances)), hang = -1)
```



Finally, let us look at the entropy of our texts. The entropy of a document measures the 'amount' of information each letter of the text produces. To get an idea of what this means, consider the 'e' is an often occurring letter in an English text, while 'z' is not. Thus, a word with a 'z' in it, it more unique and thus likely to carry unique and interesting information. The 'higher' the entropy of a text, the less 'information' is in it:

```
corpus_entropy_docs <- textstat_entropy(data_dfm, "documents")
corpus_entropy_docs <- as.data.frame(corpus_entropy_docs)
```

While not as common as the other distance metrics, entropy is sometimes used to measure the similarity between texts. Thus, it can be useful if we want to know the importance of certain words. This is because if a certain word is not important, we could consider it to be a stop word:

```
corpus_entropy_feats <- textstat_entropy(data_dfm, "features")
corpus_entropy_feats <- as.data.frame(corpus_entropy_feats)
corpus_entropy_feats <- corpus_entropy_feats[order(-corpus_entropy_feats$entropy),]
head(corpus_entropy_feats, 10)
```

```
##       feature  entropy
## 8      soviet 6.620700
```

```
## 2         war 6.443909
## 9       union 6.081728
## 7      states 5.923134
## 6      united 5.823699
## 57   military 5.612866
## 1        cold 5.573738
## 107 communist 5.374768
## 222        us 5.293932
## 12    western 5.289149
```

Looking at the data, we find that 'soviet', 'war' and 'union' have pretty high entropies. This indicates that the words added little to the information of the documents, and would-be candidates for removal from our corpus.

Chapter 7

Dictionary Analysis

One of the simplest forms of quantitative text analysis is dictionary analysis. The idea here is to look at the rate at which keywords appear in a text to classify documents into categories. Also, we can measure the extent to which documents belong to particular categories. As we do so without making any assumptions, dictionary methods present a non-statistical to text analysis. A well-known example is measuring the tone in newspaper articles, speeches, children's writings, and so on by using sentiment analysis dictionaries. Another example is the measuring of policy content in different documents as illustrated by the Policy Agendas Project dictionary (Albaugh et al., 2013).

Here, we will carry out three such analyses, the first a standard analysis and the other two focusing on sentiment. For the former, we will use political party manifestos, while for the latter we will use movie reviews and Twitter data.

7.1 Classical Dictionary Analysis

As for our dictionaries, we can either make the dictionary ourselves or use an off-the-shelf version. For the latter, we can either import the files we already have into R or use some of the versions that come with the `quanteda.dictionaries` package. For this, we first load the package:

```
library(quanteda.dictionaries)
```

We then apply one of these dictionaries to the document feature matrix we in the previous chapter. As a dictionary, we will use the one made by Laver & Garry (2000), meant for estimating policy positions from political texts. We first load this dictionary into R and then run it on the dfm using the `dfm_lookup` command:

```
data_dictionary_LaverGarry
dictionary_results <- dfm_lookup(data_dfm, data_dictionary_LaverGarry)
dictionary_results
```

Apart from off-the-shelf dictionaries, it is also possible to create our own which could suit our research question better. One approach is to use prior theory to come up with different categories and their associated words. Another approach is to use reference texts to come up with categories and words. We can also combine different dictionaries as illustrated by Young & Soroka (2012), or different dictionaries and keywords from categories in a manual coding scheme (Lind et al., 2019). Finally, we can use expert or crowd coding assessments to determine the words that best match different categories in a dictionary (Haselmayer & Jenny, 2017).

If we want to create our own dictionary in `quanteda` we use the same commands as above, but we first have to create the dictionary. To do so, we specify the words in a named list. This list contains keys (the words we want to look for) and the categories to which they belong. We then transform this list into a dictionary. Here, we choose some words which we believe will allow us to identify the different parties with ease:

```
dic_list <- list(economy = c("tax*", "invest*", "trade"),
                 war = c("army", "troops", "fight"),
                 diplomacy = c("nato","comintern","un"),
                 government = c("washington","moscow","beijing")
                 )

dic_created <- dictionary(dic_list, tolower = FALSE)
dic_created
```

```
## Dictionary object with 4 key entries.
## - [economy]:
##   - tax*, invest*, trade
## - [war]:
##   - army, troops, fight
## - [diplomacy]:
##   - nato, comintern, un
## - [government]:
##   - washington, moscow, beijing
```

If you compare the `dic_list` file with the `data_dictionary_LaverGarry` file, you will find that it has the same structure. To see the result, we can use the same command:

```
dictionary_created <- dfm_lookup(data_dfm, dic_created)
dictionary_created
```

```
## Document-feature matrix of: 205 documents, 4 features (89.39% sparse) and 0 docvars
```

```
##        features
## docs    economy war diplomacy government
##   text1       0   0         0          0
##   text2       0   0         0          0
##   text3       0   0         0          0
##   text4       0   0         0          0
##   text5       0   0         0          0
##   text6       0   0         0          0
## [ reached max_ndoc ... 199 more documents ]
```

Also note that if you would like to convert this dfm into a regular dataframe, you can use the `convert` command included in `quanteda`:

```
dictionary_df <- convert(dictionary_created, to = "data.frame")
```

Moreover, while we could look at this dataframe by either calling it in the console or looking at it in the Environment, we can also make it into an HTML widget, using the `DT` and `data.table` packages:

```
DT::datatable(dictionary_df)
```

```
## PhantomJS not found. You can install it with webshot::install_phantomjs(). If it is installed,
```

Show 10 entries Search:

	doc_id	economy	war	diplomacy	government
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	0	0	0
6	text6	0	0	0	0
7	text7	0	0	0	1
8	text8	0	0	0	0
9	text9	0	0	1	0
10	text10	0	0	0	0

Showing 1 to 10 of 205 entries

Previous 1 2 3 4 5 … 21 Next

7.2 Sentiment Analysis

The logic of dictionaries is that we can use them to see which kind of topics are present in our documents. Yet, we can also use them to provide us with measurements that are most often related to scaling. One way to do so is with *sentiment analysis*. Here, we look at whether a certain piece of text is happy, angry, positive, negative, and so on. One case in which this can help us is with movie reviews. These reviews give us a description of a movie and then tell us their opinion. Another is when we look at Twitter data, to capture the 'mood of the moment'. Here, we will look at both, starting with the movie reviews.

7.2.1 Movie Reviews

First, we load some reviews into R. The corpus we use here contains 50,000 movie reviews, each with a 1-10 rating (amongst others). As 50,000 reviews make the analysis quite slow, we will first select 30 reviews at random from this corpus. We do so via `corpus_sample`, after which we transform it via a tokens object into a dfm:

```
library(quanteda.classifiers)
reviews <- corpus_sample(data_corpus_LMRD, 30)
reviews_tokens <- tokens(reviews)
reviews_dfm <- dfm(reviews_tokens)
```

The next step is to load in a sentiment analysis dictionary. Here, we will use the Lexicoder Sentiment Dictionary, included in `quanteda` and run it on the dfm:

```
data_dictionary_LSD2015
results_dfm <- dfm_lookup(reviews_dfm, data_dictionary_LSD2015)
results_dfm
```

The next step is to convert the results to a data frame and view them:

```
sentiment <- convert(results_dfm, to="data.frame")
head(sentiment)
```

```
##                 doc_id negative positive neg_positive neg_negative
## 1  test/pos/6466_10.txt        2        5            0            0
## 2  test/neg/11718_1.txt        9        9            0            0
## 3     test/neg/47_2.txt        4        4            0            0
## 4  train/neg/8602_2.txt       18        4            0            0
## 5 train/neg/11866_1.txt       14       11            0            0
## 6  train/neg/6126_1.txt        3        2            0            0
```

Since movie reviews usually come with some sort of rating (often in the form of stars), we can see if this relates to the sentiment of the review. To do so, we have to take the rating out of the dfm and place it in a new data frame with the positive and negative sentiments:

```
star_data <- reviews_dfm@docvars$rating
stargraph <- as.data.frame(cbind(star_data, sentiment$negative, sentiment$positive))
names(stargraph) <- c("stars","negative","positive")
```

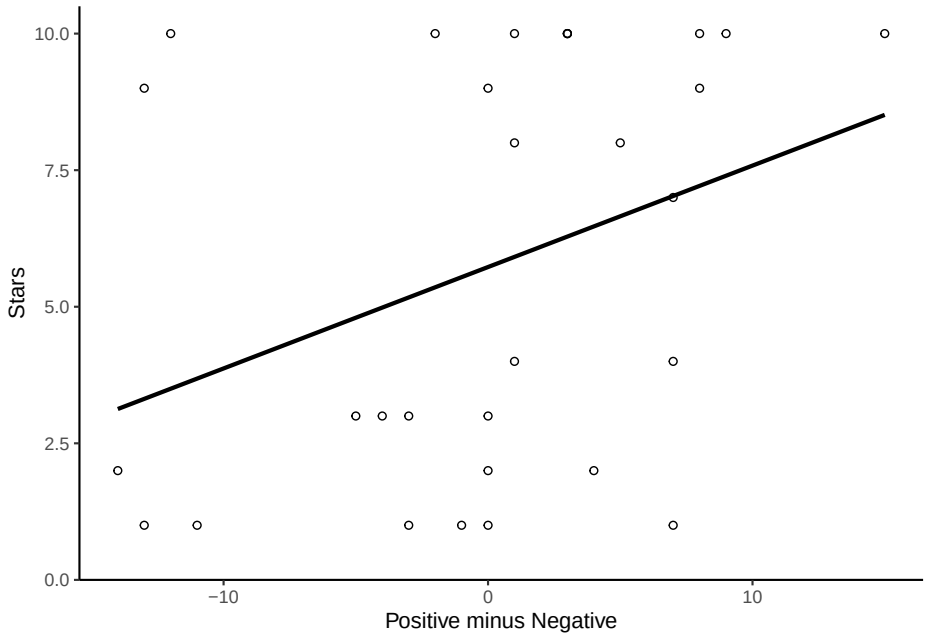
To compare the sentiment with the stars, we first have to combine the senitments into a scale. Of the many ways to do so, the simplest is to take the difference between the positive and negative words (positive – negative). Another option is to take the ratio of positive words against both positive and negative (positive/positive+negative). Here, we do both:

```
sentiment_difference <- stargraph$positive-stargraph$negative
sentiment_ratio <- (stargraph$positive/ (stargraph$positive + stargraph$negative))
stargraph <- cbind(stargraph, sentiment_difference,sentiment_ratio)
```

Then, we can plot the ratings and the scaled sentiment measures together with a linear regression line:

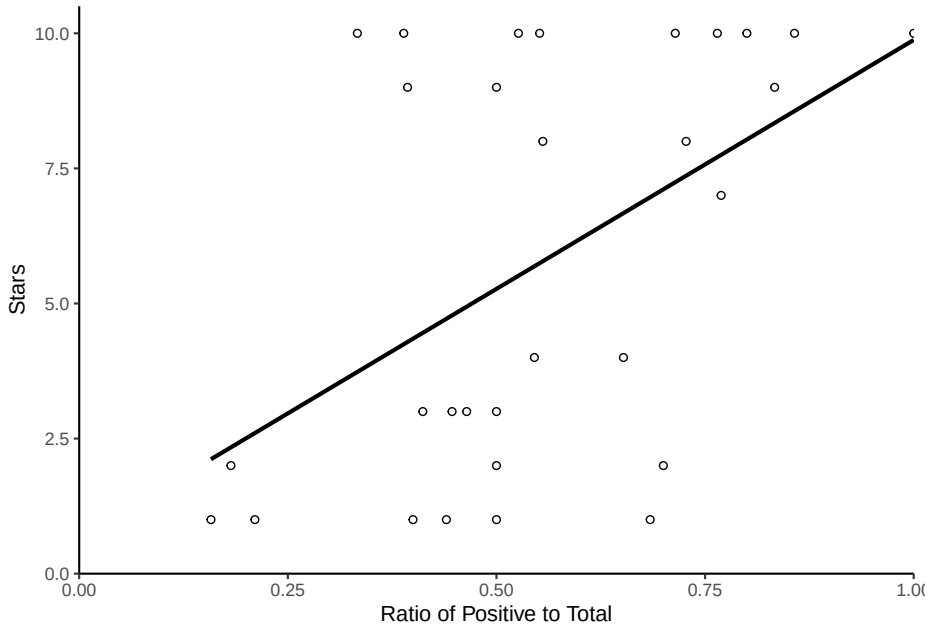
```
library(ggplot2)

ggplot(stargraph,aes(x = sentiment_difference, y = stars)) +
 geom_point(shape = 1) +
 geom_smooth(method = lm, se = FALSE, color="black") +
 scale_y_continuous(limits = c(0, 10.5), expand = c(0,0))+
 xlab("Positive minus Negative") +
 ylab("Stars") +
 theme_classic()
```

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




```
ggplot(stargraph,aes(x = sentiment_ratio, y = stars)) +
 geom_point(shape = 1) +
 geom_smooth(method = lm, se = FALSE, color="black") +
 scale_y_continuous(limits = c(0, 10.5), expand = c(0,0))+
 scale_x_continuous(limits = c(0, 1), expand = c(0,0))+
 xlab("Ratio of Positive to Total") +
 ylab("Stars") +
 theme_classic()
```

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



Finally, let us look at how we can make any more inferences, by estimating confidence intervals around the point estimates. For this, we again add a column, this time one with the total of positive and negative words as scored by the dictionary. We do so by copying the data frame to a new data frame and adding a new column filled with NA values:

```
reviews_bootstrap <- sentiment
reviews_bootstrap$n <- NA
```

We then again specify the number of reviews, the replications that we want and change the data frame into an array:

```
library(combinat)

nman <- nrow(reviews_bootstrap)
nrepl <- 1000
manifBSn <- manifBSnRand <- array(as.matrix(reviews_bootstrap[,2:3]),
 c(nman, 2, nrepl + 1),
 dimnames = list(1:nman,names(reviews_bootstrap[,2:3]),
 0:nrepl))
```

Then, we bootstrap the word counts for each movie review and compute percentages for each category using a multinomial draw:

```
n <- apply(manifBSn[1:nrow(manifBSn), , 1], 1, sum)
p <- manifBSn[, , 1]/n
```

```
for(i in 1:nrepl) {
 manifBSn[, , i] <- rmultinomial(n, p)
 }
```

We can then ask R to compute the quantities of interest. These are standard errors for each category, as well as the percentage coded for each category (Mikhaylov et al., 2012):

```
NegativeSE <- apply(manifBSn[, "negative", ]/n * 100, 1, sd)
PositiveSE <- apply(manifBSn[, "positive", ]/n * 100, 1, sd)
perNegative <- apply(manifBSn[, "negative", ]/n * 100, 1, mean)
perPositive <- apply(manifBSn[, "positive", ]/n * 100, 1, mean)
```

We then save these quantities of interest in a new data frame:

```
dataBS <- data.frame(cbind(reviews_bootstrap[, 1:3], NegativeSE,PositiveSE, perNegative
```

Then, we first calculate the confidence intervals and add these:

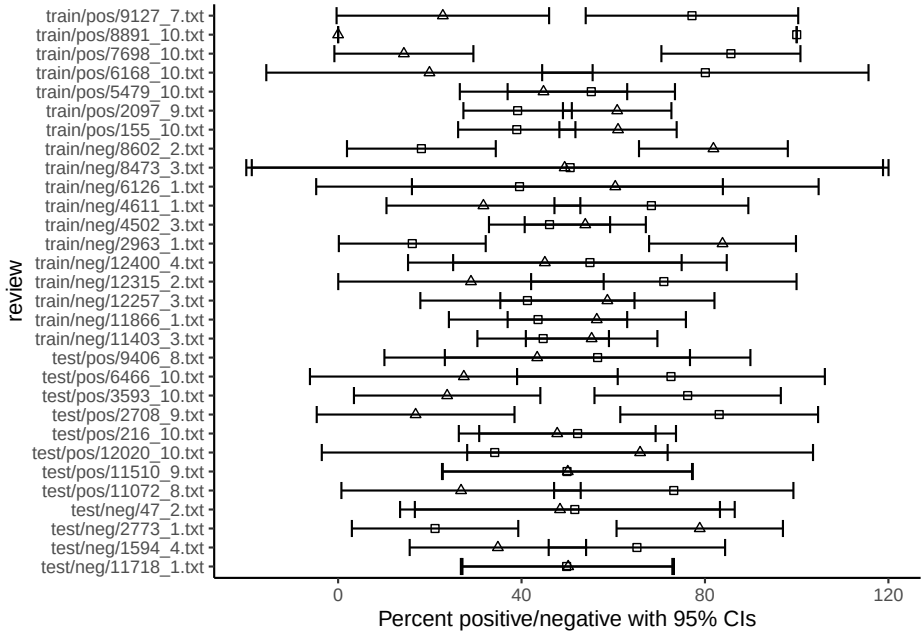
```
pos_hi <- dataBS$perPositive + (1.96 * dataBS$PositiveSE)
pos_lo <- dataBS$perPositive - (1.96 * dataBS$PositiveSE)
neg_lo <- dataBS$perNegative - (1.96 * dataBS$NegativeSE)
neg_hi <- dataBS$perNegative + (1.96 * dataBS$NegativeSE)
dataBS <- cbind(dataBS, pos_hi, pos_lo, neg_lo, neg_hi)
```

Finally, we can then make the graph. Here, we plot each of the positive and negative points and then overlay them with their error bars:

```
library(ggplot2)

ggplot() +
 geom_point(data = dataBS,aes(x = perPositive, y = doc_id), shape = 0) +
 geom_point(data = dataBS,aes(x = perNegative, y = doc_id), shape = 2) +
 geom_errorbarh(data = dataBS,aes(x = perPositive, xmax = pos_hi,xmin = pos_lo, y = doc
 geom_errorbarh(data = dataBS,aes(x = perNegative, xmax = neg_hi,xmin = neg_lo, y = doc
 xlab("Percent positive/negative with 95% CIs") +
 ylab("review")+
 theme_classic()
```

```
## Warning in geom_errorbarh(data = dataBS, aes(x = perPositive, xmax = pos_hi, :
## Ignoring unknown aesthetics: x
```

```
## Warning in geom_errorbarh(data = dataBS, aes(x = perNegative, xmax = neg_hi, :
## Ignoring unknown aesthetics: x
```



As we can see in this particular example, the fact that some documents are shorter than others introduces a lot of uncertainty in the estimates. As evident from the overlapping confidence intervals, for most reviews, the percentage of negative words is not very different from the percentage of positive words. In other words: the sentiment for these reviews is rather mixed.

7.2.2 Twitter

Now, let us turn to an example using Twitter data. Here, we will look at the major problems that have occurred to several of the major US airlines. For this, researchers scraped data from Twitter between 16 and 24 February of 2015. Then, using the Crowdflower platform, they asked contributors to classify each tweet (their sentiment) as either negative, positive, or neutral, and, if negative, what their reason was for classifying it as such. Besides this, the data also contains information on how 'confident' coders were about their classification and reason, some information on the Airline, and some info on the Tweet. Finally, we get some information on the "gold" tweets, which Crowdflower uses to figure out how well their coders are doing.

While we can download the data from the website (https://www.kaggle.com/crowdflower/twitter-airline-sentiment), for ease-of-use, we also placed it on GitHub:

```
urlfile = "https://raw.githubusercontent.com/SCJBruinsma/qta-files/master/Tweets.csv"
tweets <- read.csv(url(urlfile))
```

Given that this is Twitter data, we have to do quite some cleaning to filter out everything we do not want. While we earlier saw that we can perform cleaning

on a corpus, we can also clean our text while still in a data frame. We can do this with R's in-house `gsub` command, which can replace any part of a string. To understand how this works, say that we want to remove all the mentions of websites from our tweets. We then do as such:

```
tweets$text <- gsub("http.*","", tweets$text)
```

Thus, we substitute those strings that start with `http.*` (the asterisk denotes a wildcard, which means that anything can follow) and replace it with (that is, nothing). We do this for any string that is in `tweets$text`. Using this technique, we also remove slashes, punctuation, various symbols, `RT` (retweets), and references (`href`):

```
tweets$text <- gsub("https.*","", tweets$text)
tweets$text <- gsub("\\$", "", tweets$text)
tweets$text <- gsub("@\\w+", "", tweets$text)
tweets$text <- gsub("[[:punct:]]", "", tweets$text)
tweets$text <- gsub("[ |\t]{2,}", "", tweets$text)
tweets$text <- gsub("^ ", "", tweets$text)
tweets$text <- gsub(" $", "", tweets$text)
tweets$text <- gsub("RT", "", tweets$text)
tweets$text <- gsub("href", "", tweets$text)
```

We then transform our dataframe into a corpus (specifying that our text is in the `tweets$text` field), transform this into a tokens object, lower all the words, remove the stop words, and finally make it into a dfm:

```
corpus_tweets <- corpus(tweets, text_field = "text")
data_tweets_tokens <- tokens(corpus_tweets)
data_tweets_tokens <- tokens_tolower(data_tweets_tokens, keep_acronyms = TRUE)
data_tweets_tokens <- tokens_select(data_tweets_tokens, stopwords("english"), selectio
data_tweets_dfm <- dfm(data_tweets_tokens)
```

Now we can apply our dictionary. We can do this in two ways: applying it to the dfm, or applying it to the tokens object. Both should give roughly similar results. Yet, given that `dfm_lookup()` cannot detect multi-word expressions (as the dfm gets rid of all word order), we can use the `tokens_lookup()` and then convert this into a dfm, to compensate for this. One reason to do this here is that the LSD2015 dictionary contains some multi-word expressions that `dfm_lookup()` might miss. As a comparison, let us have a look at both:

```
results_tokens <- tokens_lookup(data_tweets_tokens, data_dictionary_LSD2015)
results_tokens <- dfm(results_tokens)
results_tokens <- convert(results_tokens, to="data.frame")

results_dfm <- dfm_lookup(data_tweets_dfm, data_dictionary_LSD2015)
results_dfm <- convert(results_dfm, to="data.frame")
```

Now let us see how well our dictionary has done. To see this, we compare the

sentiment of the tweet according to the dictionary with the sentiment assigned by the coder. We take this information out of our original data, and recode it (so it has got numerical values):

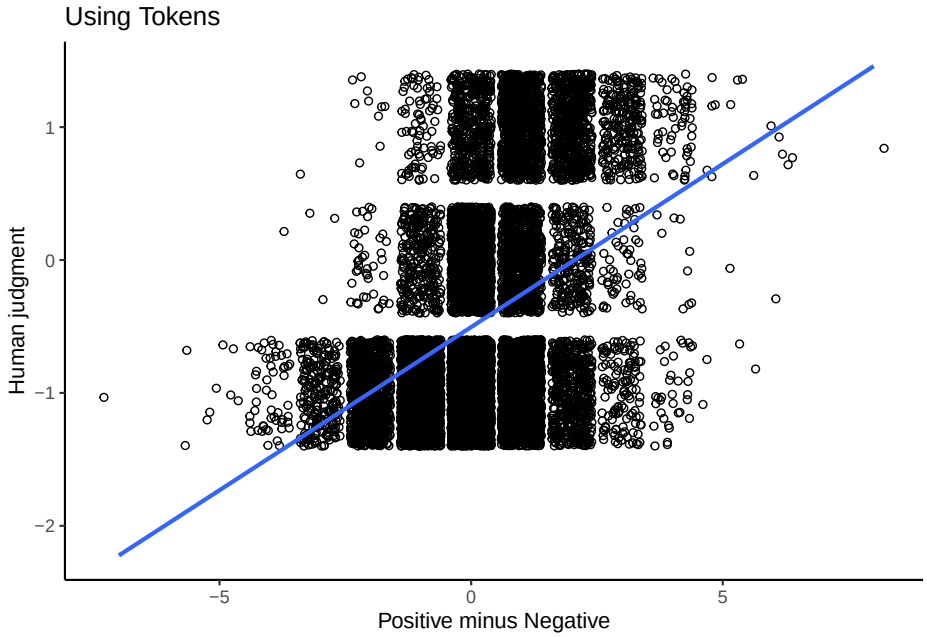
```
library(car)

labels <- tweets$airline_sentiment
labels <- car::recode(labels, "'positive'=1;'negative'=-1;'neutral'=0")
table(labels)
```

A quick look at the data (with `table()`) reveals that the majority of the tweets are negative, a fair share neutral, and finally some positive ones. Now, let us bind this data to the output of our dictionary analysis, and calculate an overall score for each tweet. We do this by subtracting the positive score from the negative score (that is, the higher the score, the more positive the tweet):

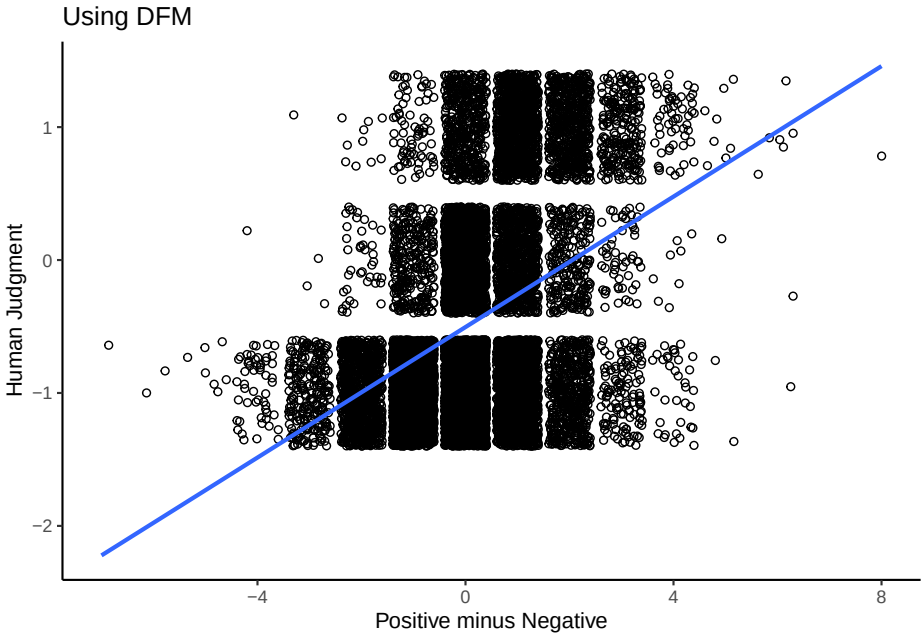
```
comparison_tokens <- as.data.frame(cbind(results_tokens$positive, results_tokens$negative, labels
difference_tokens <- results_tokens$positive - results_tokens$negative
comparison_tokens <- cbind(comparison_tokens, difference_tokens)

comparison_dfm <- as.data.frame(cbind(results_dfm$positive, results_dfm$negative, labels))
difference_dfm <- results_dfm$positive - results_dfm$negative
comparison_dfm <- cbind(comparison_dfm, difference_dfm)
```

Finally, we can place this all in a graph, in which we plot both the human judgement scores and the scores we calculated by subtracting the positive and negative codes. Also, we plot a simple linear equation to better understand the relation:

```
library(ggplot2)

ggplot(comparison_tokens ,aes(x = difference_tokens, y = labels)) +
 geom_jitter(shape = 1) +
 geom_smooth(method = lm, se = FALSE) +
 xlab("Positive minus Negative") +
 ylab("Human judgment") +
 ggtitle("Using Tokens")+
 theme_classic()
```




```
ggplot(comparison_dfm, aes(x = difference_dfm, y = labels)) +
 geom_jitter(shape = 1) +
 geom_smooth(method = lm, se = FALSE) +
 xlab("Positive minus Negative") +
 ylab("Human Judgment") +
 ggtitle("Using DFM")+
 theme_classic()
```



As we can see, there is a positive relation (0.6947 for the tokens and 0.6914 for the dfm), which is quite good considering our approach does not involve any human coders at all.

7.3 Sentiment Analysis using VADER

Another type of sentiment-analysis we can use is known as VADER Hutto & Gilbert (2014) (Valence Aware Dictionary and sEntiment Reasoner) which is a sentiment dictionary specifically made for sentiment in social media. Also, where most dictionaries tend to depend on a single coder classifying the terms, VADER uses multiple coders in order to arrive at a dictionary. So how well does it work? Let us test this again this the airline data we had before. First, we re-load this data back into R. Then, we separate the text and select 1000 tweets to work with:

```
urlfile = "https://raw.githubusercontent.com/SCJBruinsma/qta-files/master/Tweets.csv"
tweets <- read.csv(url(urlfile))
tweets <- tweets[sample(nrow(tweets), 1000), ]

text <- tweets$text
```

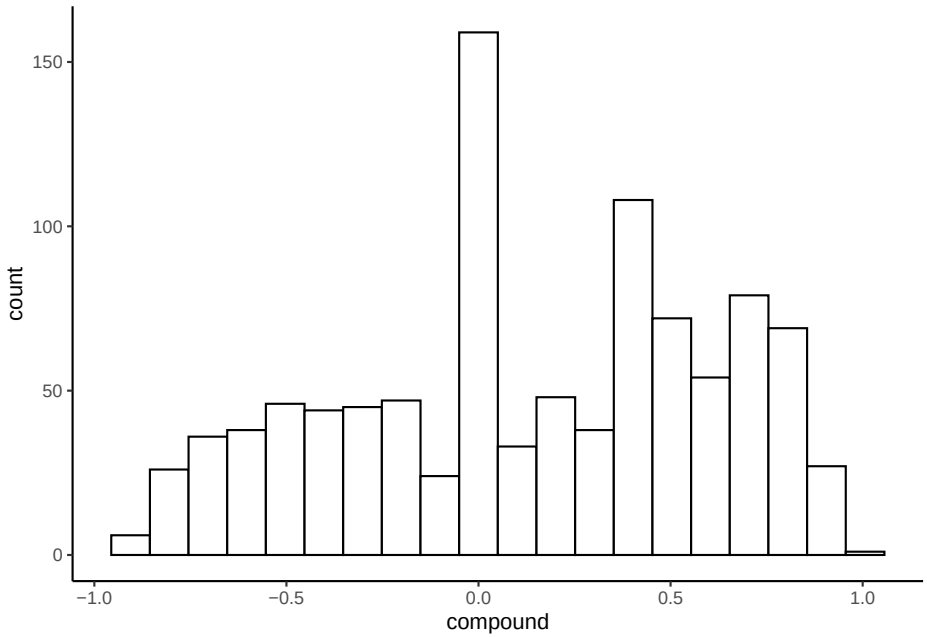
We then apply VADER to our tweets. Note that the `vader` package has just two available commands: either to measure values for a single string, or to measure values for a dataframe. Here we will use the latter:

```
library(vader)
results_vader <- vader_df(text)
```

VADER then provides us with a dataframe consisting of seven different variables. The first contains the text, the second the "word_scores" which is a string containing an ordered list with the scores for each of the words in the text, the third the "compound" which is the sum of all the valence scores in the document, and "pos", "neg" and "neu" refer to the positive, negative and neutral content specifically. In addition, there is an additional count for the occurrence of the word "but", as this often complicates the calculation of any type of sentiment.

To get a better idea of the output, we can look at the distribution of the scores:

```
library(ggplot2)

ggplot(data = results_vader,
       aes(x=compound)
       ) +
geom_histogram(bins = 20,
               color="black",
               fill="white") +
theme_classic()
```



As we can see, there are quite a lot of neutral scores. If we look at the scores, tweets such as "@JetBlue Counting on your flight 989 to get to DC!" does not have any apparent sentiment, and seem to be quite often occurring, which most

likely explains our results here.

Chapter 8

Exercices

1. Using the sentiment analysis dictionary, see if you can estimate the sentiment of the Wikipedia article on the Cold War? Are there differences between the paragraphs, and if so, what causes them?
2. How did the Cold War influence culture? Make a dictionary with words measuring this and run it on the data to discuss this.

Chapter 9

Scaling Methods

With a dictionary, we aimed to classify our texts into different categories based on the words they contain. While practical, there is no real way to compare these categories: one category is no better or worse than the other. If we do want to compare texts, we have to place them on some sort of scale. Here, we will look at three ways in which we can do so: *Wordscores* (Laver et al., 2003), *Wordfish* (Slapin & Proksch, 2008), and *Correspondence Analysis*. The first two methods used to be part of the main `quanteda` package, but have now moved to the `quanteda.textmodels` package, while we find CA in the `FactoMineR` package.

9.1 Wordscores

The idea of Wordscores is to use reference texts (from which we know the position) to position our virgin texts (from which we do not know the position). Here, we aim to position the 2005 party manifestos of the five largest parties in the United Kingdom on a general left-right scale. For this, we will use the 2001 party manifestos of the same parties as reference texts. To know their positions, we will use the left-right scale from the 2002 Chapel Hill Expert Survey (Bakker et al., 2012) to do so. So, we load our data, make a subset, transform it into a dfm, and clean it:

```
library(quanteda)
library(quanteda.corpora)

data(data_corpus_ukmanifestos)
corpus_manifestos <- corpus_subset(data_corpus_ukmanifestos, Year == 2001 | Year == 2005)
corpus_manifestos <- corpus_subset(corpus_manifestos, Party=="Lab" | Party=="LD" | Party == "Con"

data_manifestos_tokens <- tokens(
```

```
 corpus_manifestos,
 what = "word",
 remove_punct = TRUE,
 remove_symbols = TRUE,
 remove_numbers = TRUE,
 remove_url = TRUE,
 remove_separators = TRUE,
 split_hyphens = FALSE,
 include_docvars = TRUE,
 padding = FALSE,
 verbose = TRUE
)

data_manifestos_tokens <- tokens_tolower(data_manifestos_tokens, keep_acronyms = FALSE)
data_manifestos_tokens <- tokens_select(data_manifestos_tokens, stopwords("english"), s

data_manifestos_dfm <- dfm(data_manifestos_tokens)
```

Then, we check the order of the documents inside our dfm:

```
data_manifestos_dfm@Dimnames$docs
```

```
##  [1] "UK_natl_2001_en_Con" "UK_natl_2001_en_Lab" "UK_natl_2001_en_LD"
##  [4] "UK_natl_2001_en_PCy" "UK_natl_2001_en_SNP" "UK_natl_2005_en_Con"
##  [7] "UK_natl_2005_en_Lab" "UK_natl_2005_en_LD"  "UK_natl_2005_en_PCy"
## [10] "UK_natl_2005_en_SNP"
```

We can then set the scores for the reference texts. For the virgin texts, we set `NA` instead. Then, we run the wordscores model - providing the dfm and the reference scores - and save it into an object:

```
library(quanteda.textmodels)
```

```
## Warning: package 'quanteda.textmodels' was built under R version 4.3.3
```

```
scores <- c(7.72,5.18,3.82,3.2,3,NA,NA,NA,NA,NA)
ws <- textmodel_wordscores(data_manifestos_dfm, scores)
summary(ws)
```

```
##
## Call:
## textmodel_wordscores.dfm(x = data_manifestos_dfm, y = scores)
##
## Reference Document Statistics:
##                      score total min max   mean median
## UK_natl_2001_en_Con  7.72  7179   0  92 0.8646      0
## UK_natl_2001_en_Lab  5.18 16386   0 166 1.9735      0
## UK_natl_2001_en_LD   3.82 12338   0 101 1.4860      0
```

```
## UK_natl_2001_en_PCy  3.20  3508   0  72 0.4225      0
## UK_natl_2001_en_SNP  3.00  5692   0 108 0.6855      0
## UK_natl_2005_en_Con    NA  4349   0  46 0.5238      0
## UK_natl_2005_en_Lab    NA 13366   0 147 1.6098      0
## UK_natl_2005_en_LD     NA  9263   0 109 1.1156      0
## UK_natl_2005_en_PCy    NA  4203   0 148 0.5062      0
## UK_natl_2005_en_SNP    NA  1508   0  49 0.1816      0
##
## Wordscores:
## (showing first 30 elements)
##         time       common        sense conservative    manifesto introduction
##        5.838        6.540        7.376        7.161        4.478        3.982
##        lives      raising       family       living       safely      earning
##        6.047        4.427        5.519        4.719        5.743        6.046
##      staying      healthy      growing        older      knowing        world
##        6.946        4.294        4.745        6.280        7.720        4.366
##       leader     stronger      society         town      country    civilised
##        4.524        4.910        4.342        7.515        4.401        4.278
##        proud    democracy   conclusion      present    ambitious    programme
##        6.069        5.267        6.946        3.594        4.466        4.233
```

When we run the `summary` command, we can see the word scores for each word. This is the position of that word on our scale of interest. We then only need to figure out how often these words occur in each of the texts, add up their scores, and divide this by the total number of words of the texts. This gives us the *raw score* of the text. Yet, this raw score has some problems. Most important of which is that as some words occur in almost all texts, all the scores will be very clustered in the middle of our scale. To prevent this, we can spread out the scores again, so they look more like the scores of our reference texts. This rescaling has two versions. The first was the original as proposed by Laver et al. (2003), and focuses on the variance of the scores. The idea here is that the distribution of the scores of the virgin texts has the correct mean, but an incorrect variance that needs rescaling. The second, proposed by Martin & Vanberg (2008), focuses on the extremes of the scores. What it does is to take the scores of the virgin texts and stretch them out to match the extremes of the scores of the reference texts. Here, we run both so we can compare them. For the MV transformation, we will calculate the standard errors for the scores as well:

```
pred_lbg <- predict(ws, rescaling = "lbg")
```

```
## Warning: 2187 features in newdata not used in prediction.
```

```
pred_mv <- predict(ws, rescaling = "mv", se.fit = TRUE, interval = "confidence")
```

```
## Warning: 2187 features in newdata not used in prediction.
```

```
## Warning in predict.textmodel_wordscores(ws, rescaling = "mv", se.fit = TRUE, :
## More than two reference scores found with MV rescaling; using only min, max
```

```
## values.
```

```
pred_lbg
```

```
## UK_natl_2001_en_Con UK_natl_2001_en_Lab  UK_natl_2001_en_LD UK_natl_2001_en_PCy
##            8.796502            5.438328            3.971706            1.921375
## UK_natl_2001_en_SNP UK_natl_2005_en_Con UK_natl_2005_en_Lab  UK_natl_2005_en_LD
##            2.169511            5.657514            5.126266            5.045241
## UK_natl_2005_en_PCy UK_natl_2005_en_SNP
##            3.752780            4.289562
```

```
pred_mv
```

```
## $fit
##                           fit      lwr      upr
## UK_natl_2001_en_Con 7.720000 7.633905 7.806095
## UK_natl_2001_en_Lab 5.328179 5.292403 5.363955
## UK_natl_2001_en_LD  4.283594 4.242225 4.324963
## UK_natl_2001_en_PCy 2.823268 2.748279 2.898257
## UK_natl_2001_en_SNP 3.000000 2.932911 3.067089
## UK_natl_2005_en_Con 5.484291 5.386166 5.582417
## UK_natl_2005_en_Lab 5.105916 5.057239 5.154593
## UK_natl_2005_en_LD  5.048207 4.985890 5.110523
## UK_natl_2005_en_PCy 4.127666 4.038006 4.217325
## UK_natl_2005_en_SNP 4.509983 4.321648 4.698318
##
## $se.fit
## UK_natl_2001_en_Con UK_natl_2001_en_Lab  UK_natl_2001_en_LD UK_natl_2001_en_PCy
##          0.04392699          0.01825342          0.02110697          0.03826022
## UK_natl_2001_en_SNP UK_natl_2005_en_Con UK_natl_2005_en_Lab  UK_natl_2005_en_LD
##          0.03422976          0.05006496          0.02483581          0.03179478
## UK_natl_2005_en_PCy UK_natl_2005_en_SNP
##          0.04574545          0.09609115
```

Note that this does not only predict the 2005 texts, but also the 2001 texts. As such, we can use these scores to see how well this procedure can recover the original scores. One reason why this might be a problem is because of a warning you most likely received. This says that 'n features in newdata not used in prediction'. This is as the method does not use all the words from the reference texts to score the virgin texts. Instead, it only uses the words that occur in them both. Thus, when we compare the reference scores with the scores the method gives to the reference documents, can see how well the method does.

To compare the scores, we will use the Concordance Correlation Coefficient as developed by Lin (1989). This coefficient estimates how far two sets of data deviate from a line of 45 degrees (which indicates perfect agreement). To calculate this, we take the scores (here we take the LBG version) from the object we created and combine them with the original scores. From this, we only select

the first five texts (those from 2001) and calculate the CCC:

```
library(DescTools)
```

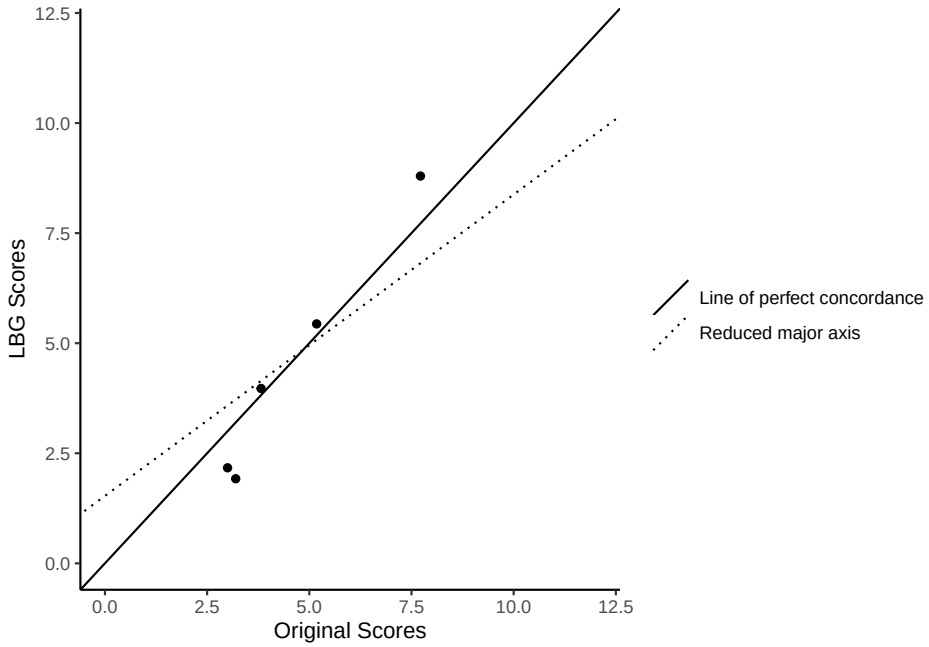
```
comparison <- as.data.frame(cbind(pred_lbg, scores))
comparison <- comparison[1:5, ]

CCC(comparison$scores, comparison$pred_lbg, ci = "z-transform", conf.level = 0.95,
    na.rm = TRUE)
```

```
## $rho.c
##         est    lwr.ci    upr.ci
## 1 0.9239214 0.8243074 0.9680456
##
## $s.shift
## [1] 1.444046
##
## $l.shift
## [1] -0.05943275
##
## $C.b
## [1] 0.9345458
##
## $blalt
##       mean      delta
## 1 8.258251 -1.0765018
## 2 5.309164 -0.2583283
## 3 3.895853 -0.1517065
## 4 2.560688  1.2786249
## 5 2.584755  0.8304892
```

The result here is not bad, though the confidence intervals are rather large. We can have a further look at why this is the case by plotting the data. In this plot, we will show the position of the texts, as well as a 45-degree line. Also, we plot the reduced major axis, which shows the symmetrical relationship between the two variables. This line is a linear regression, which we compute first using the `lm` command:

```
library(ggplot2)

lm_line <- lm(comparison$scores ~ comparison$pred_lbg)

ggplot(comparison, aes(x=scores, y=pred_lbg)) +
 geom_point()+
 xlab("Original Scores")+
 ylab("LBG Scores")+
 ylim(0, 12)+
 xlim(0, 12)+
```

```
geom_abline(aes(intercept = 0,
                slope =1,
                linetype = "dashed"))+
geom_abline(aes(intercept = lm_line$coefficients[1],
                slope = lm_line$coefficients[2],
                linetype = "solid" ))+
scale_shape_manual(name = "",
                   values=c(1,3),
                   breaks=c(0,1),
                   labels=c("Line of perfect concordance" , "Reduced major axis"))+
scale_linetype_manual(name = "",
                      values=c(1,3),
                      labels=c("Line of perfect concordance" , "Reduced major axis"))+
theme_classic()
```

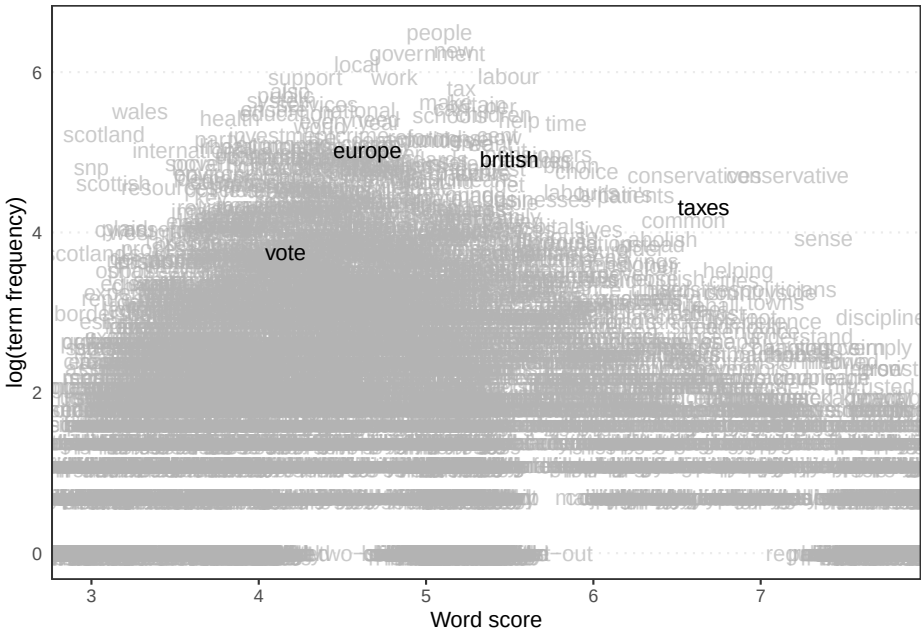


This graph allows us to spot the problem. That is that while we gave the manifesto for Plaid Cymru (PCy) a reference score of 3.20, Wordscores gave it 1.91. Removing this manifesto from our data-set would thus improve our estimates.

Apart from positioning the texts, we can also have a look at the words themselves. We can do this with the `textplot_scale1d` command, for which we also specify some words to highlight:

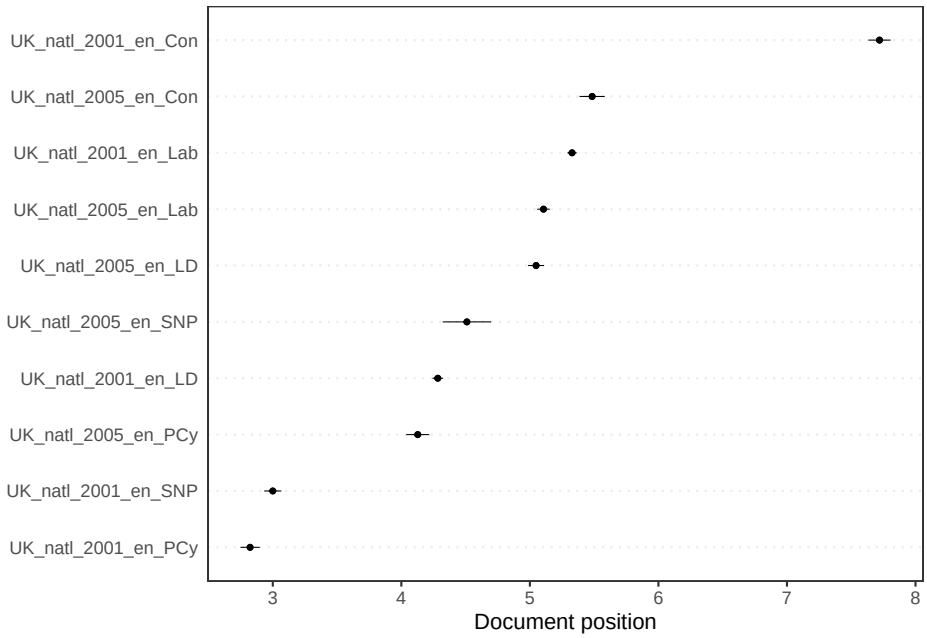
```
library(quanteda.textplots)
```

```
textplot_scale1d(ws,
                 margin = "features",
                 highlighted =c("british","vote", "europe", "taxes")
                 )
```



Finally, we can have a look at the confidence intervals around the scores we created. For this, we use the same command as above, though instead of specifying `features` (referring to the words), we specify `texts`. Note that we can only do this for the MV scores, as only here we also calculated the standard errors:

```
textplot_scale1d(pred_mv, margin = "documents")
```



Note that we can also make this graph ourselves. This requires some data-wrangling using the `dplyr` package. This package allows us to use pipes, denoted by the `%>%` command. This pipe transports an output of a command to another one before saving it. This saves us from constructing too many intermediate data sets. Thus, here we first bind together the row names of the fit (which denotes the documents), the fit itself, and the standard error of the fit (which also includes the lower and upper bound). We then transform this into a tibble (similar to a data frame), rename the first and fifth columns, and finally ensure that all the values (which are still characters) are numeric (and year a factor):

```
library(dplyr)

data_textplot <- cbind(rownames(as.data.frame(pred_mv$se.fit)), pred_mv$fit, pred_mv$se
 as_tibble() %>%
 rename(id = 1, se = 5) %>%
 mutate(fit = as.numeric(fit),
        lwr = as.numeric(lwr),
        upr = as.numeric(upr),
        se = as.numeric(se),
        year = as.factor(stringr::str_sub(id, start = 9, end = 12)))
```

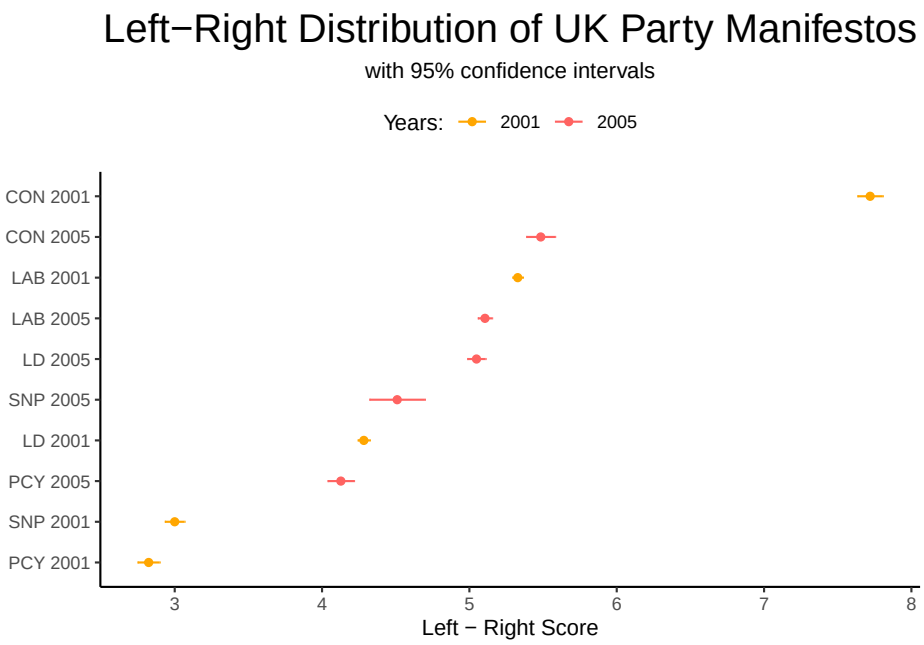
If we now look at our `data_textplot` object, we see that we have all the data we need: the fit (the average value), the lower and upper bounds, the year and the id that tells us with which party and year we are dealing. The only thing that remains is to give the parties better names. To see the current ones, type `data_textplot$id` in the console. We can then give them different names

(ensure that the order remains the same). We then sort them in decreasing order based on their fit:

```
data_textplot$id <- as.character(c("CON 2001", "LAB 2001", "LD 2001", "PCY 2001", "SNP 2001", "CO
data_textplot$id <- with(data_textplot, reorder(id, fit))
```

Then, we can plot this data using `ggplot`:

```
ggplot() +
 geom_point(data = data_textplot,
            aes(x = fit, y = id, colour = year)) +
 geom_errorbarh(data = data_textplot,
                aes(xmax = upr, xmin = lwr, y = id, colour = year),
                height = 0) +
 theme_classic() +
 scale_colour_manual(values = c("#ffa600", "#ff6361"),
                     name = "Years:",
                     breaks = c("2001", "2005"),
                     labels = c("2001", "2005")) +
 labs(title = "Left-Right Distribution of UK Party Manifestos",
      subtitle = "with 95% confidence intervals",
      x = "Left - Right Score",
      y = NULL) +
 theme_classic()+
 theme(plot.title = element_text(size = 20, hjust = 0.5),
       plot.subtitle = element_text(hjust = 0.5),
       legend.position = "top")
```



9.2 Wordfish

Different from Wordscores, for Wordfish we do not need any reference text. Instead of this, the method using a model (based on a Poisson distribution) to calculate the scores for the texts. The only thing we have to tell Wordfish is which texts define the extremes of our scale. While this might seems very practical, it also leaves us with a problem: which scale do we want? For example, let us have another look at the corpus of inaugural speeches of American presidents we saw earlier. What scale should we expect? Let us, for now, say that we care about a general left-right position. As benchmarks, we then set the 1965 Johnson speech as the most "left" and the 1985 Reagan speech as the most "right". Also, we set a seed as the model draws random numbers and we want our work to be replicable:

```
set.seed(42)

library(quanteda)

data(data_corpus_inaugural)
corpus_inaugural <- corpus_subset(data_corpus_inaugural, Year > 1900)

data_inaugural_tokens <- tokens(
 corpus_inaugural,
 what = "word",
 remove_punct = TRUE,
```

```
 remove_symbols = TRUE,
 remove_numbers = TRUE,
 remove_url = TRUE,
 remove_separators = TRUE,
 split_hyphens = FALSE,
 include_docvars = TRUE,
 padding = FALSE,
 verbose = TRUE
)

data_inaugural_tokens <- tokens_tolower(data_inaugural_tokens, keep_acronyms = FALSE)
data_inaugural_tokens <- tokens_select(data_inaugural_tokens, stopwords("english"), selection = '

data_inaugural_dfm <- dfm(data_inaugural_tokens)

data_inaugural_dfm@Dimnames$docs
wordfish <- textmodel_wordfish(data_inaugural_dfm, dir = c(17,22))
summary(wordfish)
```

Here, `theta` gives us the position of the text. As with Wordscores, we can also calculate the confidence intervals (note that `theta` is now called `fit`):

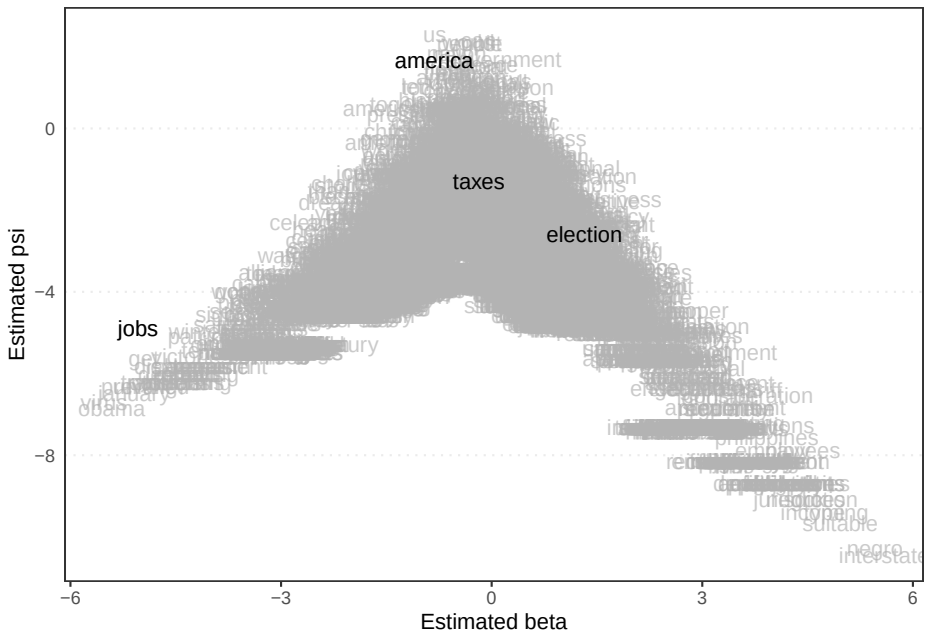
```
pred_wordfish <- predict(wordfish, interval = "confidence")
head(pred_wordfish)
```

```
## $fit
##                        fit         lwr         upr
## 1901-McKinley    1.5264409  1.45790457  1.59497725
## 1905-Roosevelt   0.5506952  0.40261155  0.69877876
## 1909-Taft        2.1180136  2.08705631  2.14897090
## 1913-Wilson      0.9494614  0.84832377  1.05059905
## 1917-Wilson      0.8648796  0.75150283  0.97825636
## 1921-Harding     1.3071861  1.24655829  1.36781392
## 1925-Coolidge    1.4398978  1.38585947  1.49393615
## 1929-Hoover      1.5400830  1.48640261  1.59376344
## 1933-Roosevelt   1.1206324  1.03168956  1.20957530
## 1937-Roosevelt   0.6493799  0.54742236  0.75133754
## 1941-Roosevelt   0.1153382 -0.01421607  0.24489242
## 1945-Roosevelt  -0.1670750 -0.36383112  0.02968117
## 1949-Truman      0.8432860  0.75602390  0.93054816
## 1953-Eisenhower  0.2792466  0.18725486  0.37123825
## 1957-Eisenhower  0.1574969  0.04465718  0.27033658
## 1961-Kennedy    -0.5395875 -0.64994887 -0.42922607
## 1965-Johnson    -0.7821091 -0.88376502 -0.68045324
## 1969-Nixon      -0.9602454 -1.03699050 -0.88350021
## 1973-Nixon      -0.4420077 -0.54461368 -0.33940174
```

```
## 1977-Carter     -0.3423320 -0.46809663 -0.21656746
## 1981-Reagan     -0.6953845 -0.77712614 -0.61364280
## 1985-Reagan     -0.6413867 -0.71992268 -0.56285068
## 1989-Bush       -0.8785297 -0.95604292 -0.80101647
## 1993-Clinton    -1.1462836 -1.22477539 -1.06779189
## 1997-Clinton    -0.8669737 -0.94358633 -0.79036108
## 2001-Bush       -0.7438888 -0.84090471 -0.64687286
## 2005-Bush       -0.4113008 -0.50479109 -0.31781044
## 2009-Obama      -1.0807899 -1.14809049 -1.01348932
## 2013-Obama      -1.0547830 -1.12762813 -0.98193782
## 2017-Trump      -1.3804172 -1.45178196 -1.30905236
## 2021-Biden      -1.3289432 -1.38772817 -1.27015814
```

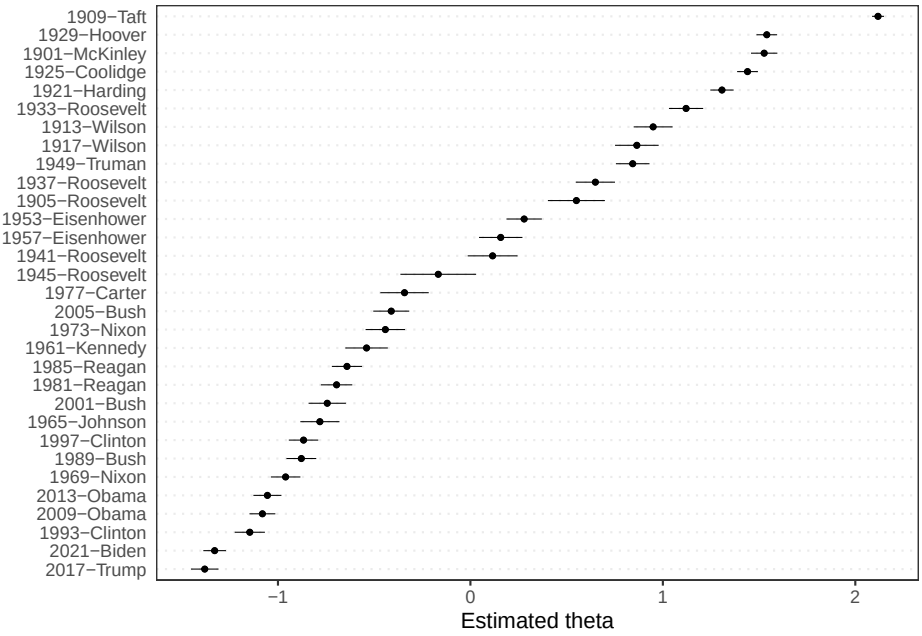
As with Wordscores, we can also plot graphs for Wordfish, using the same commands. The first graph we will again be looking at is the distribution of the words, which here forms an 'Eifel Tower'-like graph:

```
textplot_scale1d(wordfish,
                 margin = "features",
                 highlighted = c("america","jobs","taxes","election")
                 )
```



And then we can do the same for the documents as well. Note that we can also make a similar graph to the one we made ourselves above (just replace `pred_mv` with `pred_wordfish`):

```
textplot_scale1d(wordfish,
                 margin = "documents"
                 )
```



Looking at the results here gives us an interesting picture. Remember that we chose our benchmark texts to look at the left-right position of our texts? Here, we see that both these texts (the 1965 Johnson and 1985 Reagan) are quite close to each other. Sticking with our interpretation that Reagan is more right-wing than Johnson, this would mean that the 1909 Taft address was the most right-wing and the 2017 Trump text the most left-wing. Whether this is true is of course up to our interpretation.

9.3 Correspondence Analysis

Correspondence Analysis uses a similar logic as Principal Component Analysis. Yet, while PCA requires metric data, CA only requires nominal data (such as text). The idea behind both is to reduce the complexity of the data by looking for new dimensions. These dimensions should then explain as much of the original variance that is present in the data as possible. Within R many packages can run CA (such as the `ca` and `FactoMineR` packages and even `quanteda.textmodels`). One interesting package is the `R.temis` package. This package aims to bring the techniques of qualitative text analysis into R. Thus, the package focuses on the import of corpus from programs such as Alceste and sites such as LexisNexis - programs that are often used in qualitative text analysis. The package itself is built on the popular `tm` package and has a similar logic.

To carry out the Correspondence Analysis, `R.temis` uses the `FactoMineR` and `factoextra` packages (Lê et al., 2008). Here, we will look at an example using data from an article on the stylistic variations in the Twitter data of Donald Trump between 2009 and 2018 (Clarke & Grieve, 2019). Here, the authors aimed to figure out whether the way Trump's tweets were written fluctuated over time. To do so, they downloaded 21,739 tweets and grouped them into 63 categories over 4 dimensions based on their content. Given that all the data used in the article is available for inspection, we can attempt to replicate part of the analysis here.

First, we load the packages we need for the Correspondence Analysis:

```
library(FactoMineR)
library(factoextra)
library(readr)
```

Then, we import the data. You can do so either by downloading the replication data yourselves, or use the file we already put up on GitHub:

```
urlfile = "https://raw.githubusercontent.com/SCJBruinsma/qta-files/master/TRUMP_DATA.cs
tweets <- read_csv(url(urlfile), show_col_types = FALSE)
```

This data set contains quite some information we do not need. To begin with, we remove all those variables that do not contain information about the 63 categories and the length of the tweet in words. Also, for clarity's sake, we sample 200 of the tweets:

```
tweets <- tweets[sample(nrow(tweets), 200), ]
tweets_mat <- tweets[,2:65]
```

We can then run the MCA with the `FactoMineR` package. For this, we have to give the data set and the number of dimensions we think are in the data. We can set the latter either by establishing the dimensions as in a regular PCA (for example through a scree plot) or based on theory. Here we combine both and use the 5 dimensions established in the article. Besides this, we set a supplementary quantitative variable as `quanti.sup=1`. As this is a quantitative variable, it is not taken into consideration by the MCA, but does allow us to assess later on how it correlates with each of the five dimensions:

```
mca_tweets <- MCA(tweets_mat, ncp=5, quanti.sup=1, graph = FALSE)
```

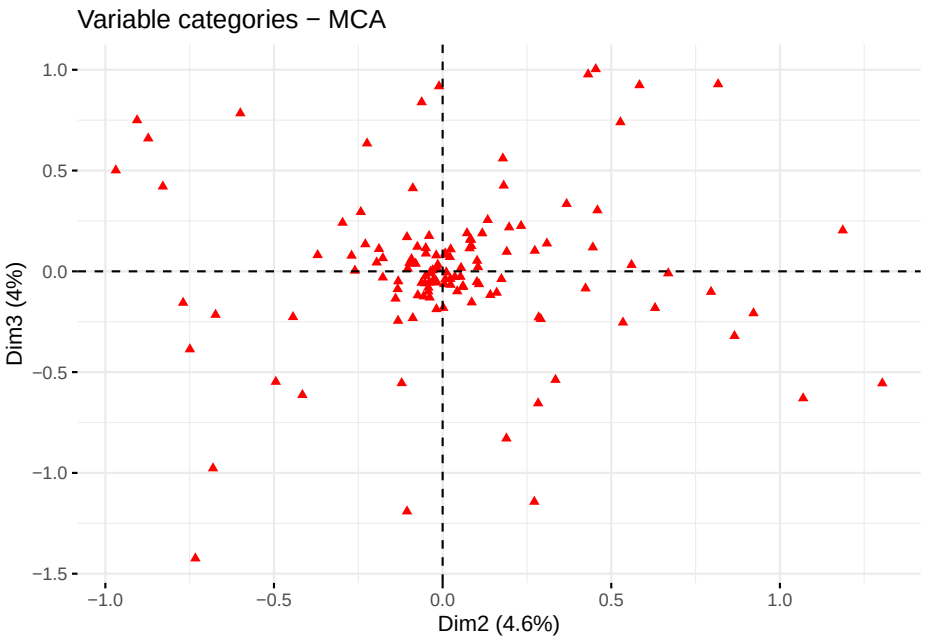
First, let's start by looking at the association of the word length with the five dimensions:

```
mca_tweets$quanti.sup
```

```
## $coord
##               Dim 1      Dim 2      Dim 3      Dim 4       Dim 5
## WORDCOUNT 0.8668104 -0.2107786 0.02090331 0.03345767 -0.05383935
```

As we can see, the word length has a strong correlation with Dimension 1. This means that this dimension captures the length of the words and is not a separate dimension. Thus, when we want to look at the correspondence between the categories and the dimensions, we can ignore this dimension. Thus, for the MCA, we will look at dimensions 2 and 3:

```
fviz_mca_var(mca_tweets,
             repel = TRUE,
             geom = c("point"),
             axes = c(2, 3),
             ggtheme = theme_minimal()
             )
```



Here, we only plot the points as adding the labels as well will make the picture quite cluttered. In the article, the authors identify Dimension 2 as 'Conversational Style' and Dimension 3 as 'Campaigning Style'. The plot thus shows us that some categories belong to one of these dimensions and not to the other. To see for which cases this is most often the case (the ones that have the most extreme positions), we can have a look at their coordinates:

```
var <- get_mca_var(mca_tweets)
coordinates <- as.data.frame(var$coord)
coordinates <- coordinates[order(coordinates$`Dim 2`),]
head(coordinates)
```

```
##                    Dim 1      Dim 2      Dim 3        Dim 4       Dim 5
## POSESPRPN_P    0.03383985 -0.9689511  0.5011988  0.036488377  0.36083073
```

```
## PROGRESSIVE_P  0.79201463 -0.9059134  0.7499092 -0.674316668  0.46134193
## SUPERLATIVE_P  0.37755724 -0.8728057  0.6594311 -0.147514294 -0.41714831
## GERUND_P       0.20213101 -0.8295159  0.4209137  0.089642937  0.82377111
## MULTIWVB_P     0.42162442 -0.7694130 -0.1558600 -0.003865414  0.37457619
## COLON_P       -0.24915096 -0.7493897 -0.3866968  1.023830211 -0.07073923
```

Here, remember to look only at the results from the second column onward. Here, we see that one extreme category for the second dimension (Conversational Style) was the use of a colon (:) or possessive proper nouns (such as Hillary's). This seems to fit well with the idea of conversational style. We can also see that the latter one also corresponds quite well with Dimension 3 (Campaigning Style), while the first one does not.

As you can see, the possibilities with MCA call for a rather investigative approach. For this reason, the designers of `FactoMineR` developed a Shiny app that allows you to play around with the data and look at all the various options. Load it by running:

```
library(Factoshiny)
res.shiny <- MCAshiny(tweets_mat)
```

Ensure you quit by clicking the "Quit the App" button to return to R. For more information on the 'Facto'-family packages, please have a look at the original article by Lê et al. (2008) or the website that belongs to it: http://factominer.free.fr/.

Chapter 10

Supervised Methods

While with scaling we try to place our texts on a scale, with supervised methods we go back to what we did with dictionary analysis: classification. Within `quanteda` there are many different models for supervised methods, of which we will cover two. These are *Support Vector Machines* (SVM) and *Naive Bayes* (NB). The first classifies texts by looking at their position on a hyperplane, the second by their (Bayesian) probabilities.

10.1 Support Vector Machines

To show how SVM works, we will look at an example of SVM in `quanteda` and one in `RTextTools`, and an example of NB in `quanteda`.

10.1.1 SVM with RTextTools

For the SVM, we will start with an example using our Twitter data and the `RTextTools` package. First, we load the Twitter data:

```
library("RTextTools")
library("car")

urlfile <- "https://raw.githubusercontent.com/SCJBruinsma/qta-files/master/Tweets.csv"
tweets <- read.csv(url(urlfile))
tweets$text <- gsub("http.*","", tweets$text)
tweets$text <- gsub("https.*","", tweets$text)
tweets$text <- gsub("\\$", "", tweets$text)
tweets$text <- gsub("@\\w+", "", tweets$text)
tweets$text <- gsub("[[:punct:]]", "", tweets$text)
tweets$text <- gsub("[ |\t]{2,}", "", tweets$text)
tweets$text <- gsub("^ ", "", tweets$text)
```

```
tweets$text <- gsub(" $", "", tweets$text)
tweets$text <- gsub("RT", "", tweets$text)
tweets$text <- gsub("href", "", tweets$text)

labels <- tweets$airline_sentiment
labels <- car::recode(labels, "'positive'=1;'negative'=-1;'neutral'=0")
```

The goal of the supervised learning task is to use part of this dataset to train a certain algorithm and then use the trained algorithm to assign categories to the remaining sentences. Since we know the coded categories for the remaining sentences, we will be able to evaluate how well this training was in guessing/estimating what the codes for these sentences were. We start by creating a document term matrix;

```
doc_matrix <- create_matrix(tweets$text,
                            language = "english",
                            removeNumbers = TRUE,
                            stemWords = TRUE,
                            removeSparseTerms = 0.998)
doc_matrix
```

```
## <<DocumentTermMatrix (documents: 14640, terms: 693)>>
## Non-/sparse entries: 84521/10060999
## Sparsity           : 99%
## Maximal term length: 18
## Weighting          : term frequency (tf)
```

Note that `RTextTools` gives you plenty of options in preprocessing. Apart from the options used above, we can also strip whitespace, remove punctuation, and remove stopwords. Stemming and stopword removal is language-specific, so when we select the language in the option above (`language=''english''`), `RTextTools` will carry this out according to our language of choice. As of now, the package supports Danish, Dutch, English, Finnish, French, German, Italian, Norwegian, Portuguese, Russian, Spanish, and Swedish.

We then create a container parsing the document matrix into a training set, and a test set. We will use the training set will to train the algorithm and the test set to test how well this algorithm was trained. The following command instructs R to use the first 4000 sentences for the training set the remaining 449 sentences for the test set. Moreover, we specify to append to the document matrix the variable that contains the assigned coders:

```
container <- create_container(doc_matrix,
                              labels,
                              trainSize = 1:10000,
                              testSize = 10001:14640,
                              virgin = FALSE)
```

We can then train a model using one of the available algorithms. For instance, we can use the Support Vector Machines algorithm (SVM) as follows:

```
SVM <- train_model(container, "SVM")
```

Other algorithms are available if you change the SVM option. Options exist for Lasso and Elastic-Net Regularized Generalized Linear Models (`GLMNET`), maximum entropy (`MAXENT`), scaled linear discriminant analysis (`SLDA`), bagging (`BAGGING`), boosting (`BOOSTING`), random forests (`RF`), neural networks (`NNET`), or classification trees (`TREE`).

We then use the model we trained to classify the texts in the test set. The following command instructs R to classify the documents in the test set of the container using the SVM model that we trained earlier.

```
SVM_CLASSIFY <- classify_model(container, SVM)
```

We can also view the classification that the SVM model performed as follows. The first column corresponds to the label that coders assigned to each of the tweets in the training set. The second column then gives the probability that the SVM algorithm assigned to that particular category. As you can see, while the probability for some sentences is quite high, for others it is quite low. This even while the classification always chooses the category with the highest probability.

```
head(SVM_CLASSIFY)
```

The next step is to check the classification performance of our model. To do this, we first request a function that returns a container with different summaries. For instance, we can request summaries based on the labels attached to the sentences, the documents (or in this case, the sentences) by label, or based on the algorithm.

```
analytics <- create_analytics(container, SVM_CLASSIFY)
summary(analytics)
```

```
## ENSEMBLE SUMMARY
##
##        n-ENSEMBLE COVERAGE n-ENSEMBLE RECALL
## n >= 1                   1               0.8
##
##
## ALGORITHM PERFORMANCE
##
## SVM_PRECISION    SVM_RECALL    SVM_FSCORE
##     0.6833333     0.6766667     0.6800000
```

Here, precision gives the proportion of bills that SVM classified as belonging to a category that does belong to that category (true positives) to all the bills that are classified in that category (irrespective of where they belong). Recall, then, is the proportion of bills that SVM classifies as belonging to a category

and belong to this category (true positives) to all the bills that belong to this category (true positives plus false negatives). The F score is a weighted average between precision and recall ranging from 0 to 1.

Finally, we can compare the scores between the labels given by the coders and those based on our SVM:

```
compare <- as.data.frame(cbind(labels[10001:14640], SVM_CLASSIFY$SVM_LABEL))
table(compare)
```

```
##     V2
## V1     -1    0    1
##   -1 3018  292  109
##   0   288  347   56
##   1   130   58  342
```

10.1.2 SVM with Quanteda

Instead of using a separate package, we can also use `quanteda` to carry out an SVM. For this, we load some movie reviews, select 1000 of them at random, and place them into our corpus:

```
set.seed(42)

library(quanteda)
library(quanteda.classifiers)
corpus_reviews <- corpus_sample(data_corpus_LMRD, 1000)
```

Our aim here will be to see how well the SVM algorithm can predict the rating of the reviews. To do this, we first have to create a new variable `prediction`. This variable contains the same scores as the original rating. Then, we remove 30% of the scores and replace them with NA. We do so by creating a `missing` variable that contains 30% 0s and 70% 1s. We then place the 0s with NAs. These NA scores are then the ones we want the algorithm to predict. Finally, we add the new variable to the corpus:

```
prediction <- corpus_reviews$rating

missing <- rbinom(1000, 1, 0.7)
prediction[missing == 0] <- NA

docvars(corpus_reviews, "prediction") <- prediction
```

We then transform the corpus into a data frame, and also remove stopwords, numbers and punctuation:

```
data_reviews_tokens <- tokens(
 corpus_reviews,
 what = "word",
```

```
 remove_punct = TRUE,
 remove_symbols = TRUE,
 remove_numbers = TRUE,
 remove_url = TRUE,
 remove_separators = TRUE,
 split_hyphens = FALSE,
 include_docvars = TRUE,
 padding = FALSE,
 verbose = TRUE
)
data_reviews_tokens <- tokens_tolower(data_reviews_tokens, keep_acronyms = FALSE)
data_reviews_tokens <- tokens_select(data_reviews_tokens, stopwords("english"), selection = "remo
dfm_reviews <- dfm(data_reviews_tokens)
```

Now we can run the SVM algorithm. To do so, we tell the model on which dfm we want to run our model, and which variable contains the scores to train the algorithm. Here, this is our `prediction` variable with the missing data:

```
library(quanteda.textmodels)
svm_reviews <- textmodel_svm(dfm_reviews, y = docvars(dfm_reviews, "prediction"))
svm_reviews
```

```
##
## Call:
## textmodel_svm.dfm(x = dfm_reviews, y = docvars(dfm_reviews, "prediction"))
##
## 672 training documents; 121,240 fitted features.
## Method: L2-regularized L2-loss support vector classification dual (L2R_L2LOSS_SVC_DUAL)
```

Here we see that the algorithm used 672 texts to train the model (the one with a score) and fitted 133,728 features. The latter refers to the total number of words in the training texts and not only the unique ones. Now we can use this model to predict the ratings we removed earlier:

```
svm_predict <- predict(svm_reviews)
```

While we can of course look at the resulting numbers, we can also place them in a two-way table with the actual rating, to see how well the algorithm did:

```
rating <- corpus_reviews$rating
table_data <- as.data.frame(cbind(svm_predict, rating))
table(table_data$svm_predict,table_data$rating)
```

```
##
##         1   2   3   4   7   8   9  10
##   1   172  15   9  16   5   3   3   3
##   2     7  69   6   5   2   2   1   3
##   3     7   0  82   3   1   3   0   1
```

```
##    4    5    2    5   86    7    6    0    4
##    7    3    1    1    1   55    3    3    2
##    8    0    2    2    2    8   90    7    7
##    9    4    0    0    3    6    9   76   12
##    10   5    2    4    2    7    6    6  138
```

Here, the table shows the prediction of the algorithm from top to bottom and the original rating from left to right. What we want is that all cases are on the diagonal: in that case, the prediction is the same as the original rating. Here, this happens in the majority of cases. Also, only in a few cases is the algorithm far off.

10.2 Naive Bayes

For the Naive Bayes example, we will use data from the Manifesto Project (Volkens et al., 2019), also known as the Comparative Manifesto Project (CMP), Manifesto Research Group (MRG), and MARPOR (Manifesto Research on Political Representation)). To use this data, ensure you have signed up and downloaded the API key, loaded the package and set the key:

```
library(manifestoR)
mp_setapikey("manifesto_apikey.txt")
```

While we can download the whole dataset, as it is rather large, it makes more sense to only download a part of it. Here, we take the manifestos for the United Kingdom in 2015. To tell R we want only these documents, we make a small data frame listing the party and the year we want, and then place this into the `mp_corpus` command. Note that instead of the names of the parties, the Manifesto Project assigns unique codes to each party. To see which code belongs to which party, see: https://manifesto-project.wzb.eu/down/data/2019a/codebooks/parties_MPDataset_MPDS2019a.pdf. Also, note that the date includes both the year and month of the election:

```
manifestos <- data.frame(party=c(51320, 51620, 51110, 51421, 51901, 51902, 51951), dat
manifesto_corpus <- mp_corpus(manifestos)
```

```
## Connecting to Manifesto Project DB API... corpus version: 2024-1
## Connecting to Manifesto Project DB API... corpus version: 2024-1
```

For now, we are only interested in the (quasi)-sentences the of the manifestos, the codes the coders gave them, and names of the parties. To make everything more clear, we will take these elements from the corpus, combine them into a new data-frame, and remove all the NA values. We do this because otherwise the data would also include the headers and titles of the document, which do not have any codes assigned to them:

```
text_51320 <- content(manifesto_corpus[["51320_201505"]])
text_51620 <- content(manifesto_corpus[["51620_201505"]])
```

```
text_51110 <- content(manifesto_corpus[["51110_201505"]])
text_51421 <- content(manifesto_corpus[["51421_201505"]])
text_51901 <- content(manifesto_corpus[["51901_201505"]])
text_51902 <- content(manifesto_corpus[["51902_201505"]])
text_51951 <- content(manifesto_corpus[["51951_201505"]])

texts <- c(text_51320,text_51620,text_51110,text_51421,text_51901,text_51902,text_51951)

party_51320 <- rep(51320,length.out=length(text_51320))
party_51620 <- rep(51620,length.out=length(text_51620))
party_51110 <- rep(51110,length.out=length(text_51110))
party_51421 <- rep(51421,length.out=length(text_51421))
party_51901 <- rep(51901,length.out=length(text_51901))
party_51902 <- rep(51902,length.out=length(text_51902))
party_51951 <- rep(51951,length.out=length(text_51951))

party <- c(party_51320,party_51620,party_51110,party_51421,party_51901,party_51902,party_51951)

cmp_code <- codes(manifesto_corpus)

manifesto_data <- data.frame(texts,cmp_code,party)
```

Before we go on, we have to transform the columns in our data-frame. This is because R considers two of them (*texts* and *cmp_code*) to be a factor, and also still uses the codes for the *party* variable. To solve the latter, we first transform *party* into a factor type, then assign the party names to each of the codes (Conservatives, Labour, Liberal Democrats, SNP, Plaid Cymru, The Greens, and UKIP), and then change the column to character type. We then change the **texts** column to character and the **cmp_code** column to numeric. We also create a back-up of our current dfm for later, and finally remove any missing data:

```
manifesto_data$party <- factor(manifesto_data$party,levels = c(51110, 51320, 51421, 51620, 51901,
manifesto_data$party <- as.character(manifesto_data$party)
manifesto_data$texts <- as.character(manifesto_data$texts)
manifesto_data$cmp_code <- as.numeric(as.character(manifesto_data$cmp_code))

manifesto_data_raw <- manifesto_data
manifesto_data <- na.omit(manifesto_data)
```

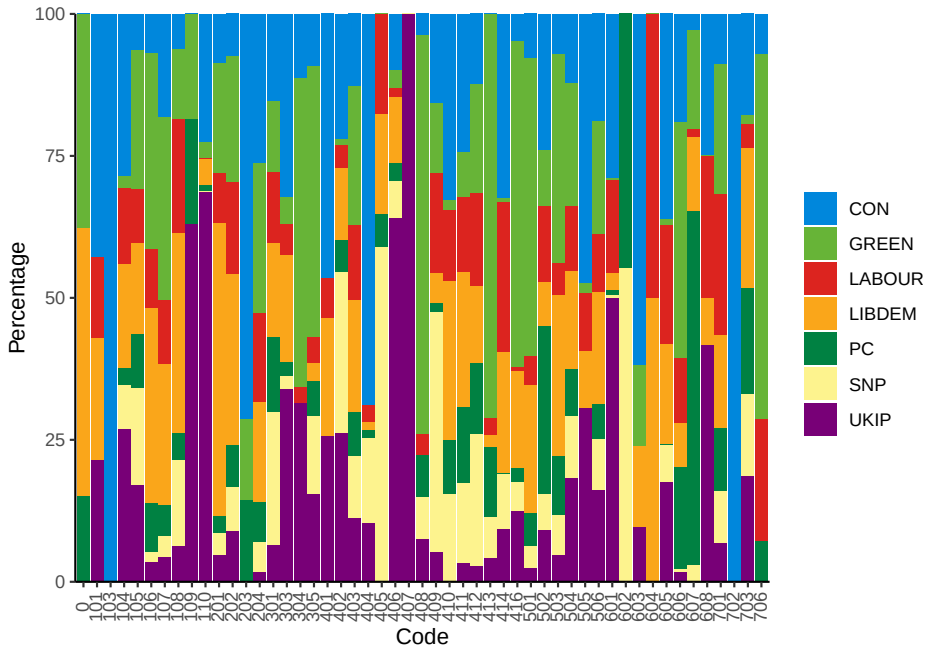
To get an idea of how much a party "owns" a code, we can calculate the row percentages. These inform us how much of the appearance of a certain code is due to a single party. To calculate these, we use the `prop.table` command. Here, the `,1` at the end tells R to look at the rows (no value would give the cell proportions, and 2 would give the column proportions). We then multiply the proportions by 100 to get the percentages. Then, we place the output in a

data-frame, and provide some names to the columns using the `names` command:

```
prop_row <- as.data.frame((prop.table(table(manifesto_data$cmp_code,manifesto_data$part
names(prop_row) <- c("Code", "Party", "Percentage")
```

While we can look at the results by looking at the `prop_row` object, it is clearer to do this in a graph. To build this graph, in the command we first specify the data, the x variable (the codes), the y variable (the percentages), and the filling of the bar (which should be the party colours). These party colours we provide in the next line (in hexadecimal notation). Then we tell `ggplot` to draw the bar chart and `stack` the bars on top of each other (the alternative is to `dodge`, in which R places the bars next to each other). Then, we specify our theme, turn the text for the codes 90 degrees, and move the codes a little bit so they are under their respective bars:

```
library(ggplot2)

ggplot(data=prop_row, aes(x=Code, y=Percentage, fill=Party)) +
 scale_fill_manual("", values = c("#0087DC","#67B437","#DC241F","#FAA61A","#008142","#
 geom_bar(stat = "identity", position = "stack") +
 scale_y_continuous(expand = c(0,0)) +
 theme_classic()+
 theme(axis.text.x = element_text(angle = 90))+
 theme(axis.text.x = element_text(vjust = 0.40))
```



Now, we can see that some parties dominate some categories, while for others

the spread is more even. For example, UKIP dominates the categories 406 and 407 - dealing with positive and negative mentions of protectionism, while the Conservatives do the same with category 103 (*Anti-Imperialism*). Note though, that these are percentages. This means that the reason the Conservatives dominate category 103 is as they have two (quasi)-sentences with that category. The others do not have the category at all (702 on *Negative Mentioning of Labour Groups* has the same issue). Other categories, such as 403 (*Market Regulation*) and 502 (*Positive Mentions of Culture*) are way better spread out over all the parties.

Another thing we can look at is what part of a party's manifesto belongs to any of the codes. This can help us answer the question: "what are the parties talking about?" To see this, we have to calculate the column percentages:

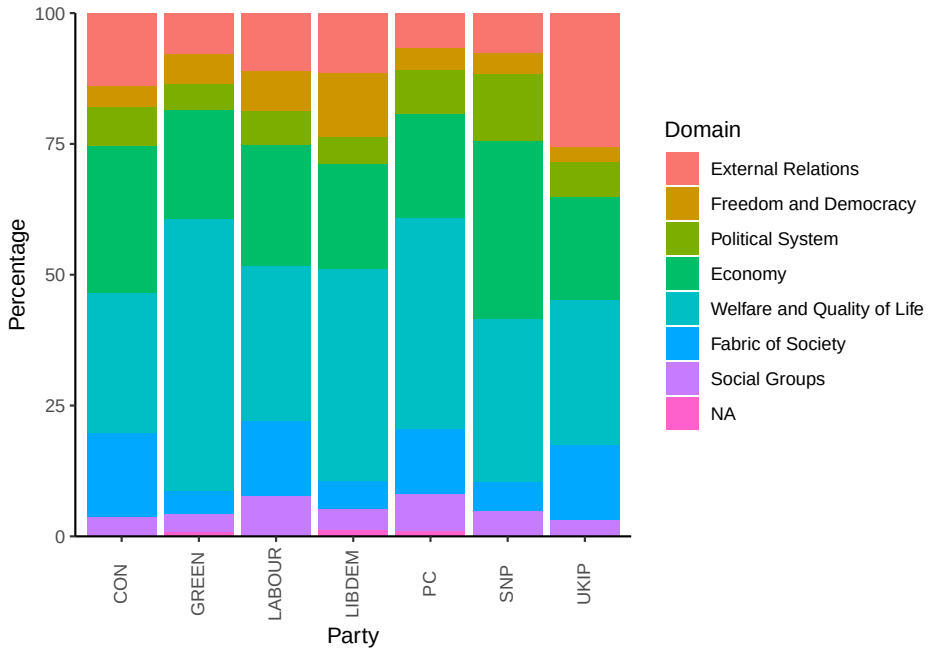
```
prop_col <- as.data.frame((prop.table(table(manifesto_data$cmp_code,manifesto_data$party), 2) * 1
names(prop_col) <- c("Code", "Party", "Percentage")
```

If we now type `prop_col`, we can see what percentage of a party manifesto was about a certain code. Yet, given that there are 57 possible codes, it is more practical to cluster these in some way. Here, we do this using the Domains to which they belonged in the codebook. In total there are 7 domains (https://manifesto-project.wzb.eu/down/papers/handbook_2014_version_5.pdf), and a category that contains the 0 code. To cluster the codes, we make a new variable called `Domain`. To do so, we first transform the codes into a numeric format, create an empty variable called `Domain`, and then replace the NA values in this empty category with the name of the domain based on the values in the Code variable. This we do using various operators R uses: `>=` means greater than and equal to, while `<=` means smaller than and equal to. Then, we make this new variable into a factor, and sort this factor in the way the codes occur:

```
prop_col$Code <- as.numeric(as.character(prop_col$Code))
prop_col$Domain <- NA

prop_col$Domain[prop_col$Code >= 101 & prop_col$Code <= 110] <- "External Relations"
prop_col$Domain[prop_col$Code >= 201 & prop_col$Code <= 204] <- "Freedom and Democracy"
prop_col$Domain[prop_col$Code >= 301 & prop_col$Code <= 305] <- "Political System"
prop_col$Domain[prop_col$Code >= 401 & prop_col$Code <= 416] <- "Economy"
prop_col$Domain[prop_col$Code >= 501 & prop_col$Code <= 507] <- "Welfare and Quality of Life"
prop_col$Domain[prop_col$Code >= 601 & prop_col$Code <= 608] <- "Fabric of Society"
prop_col$Domain[prop_col$Code >= 701 & prop_col$Code <= 706] <- "Social Groups"
prop_col$Domain[prop_col$Code == 0] <- "NA"

prop_col$Domain <- as.factor(prop_col$Domain)
prop_col$Domain <- factor(prop_col$Domain,levels(prop_col$Domain)[c(2,4,6,1,8,3,7,5)])
```

We then construct a plot as we did above:

```
ggplot(data=prop_col, aes(x=Party, y=Percentage, fill=Domain)) +
 geom_bar(stat = "identity", position = "stack") +
 scale_y_continuous(expand = c(0,0)) +
 theme_classic()+
 theme(axis.text.x = element_text(angle = 90))+
 theme(axis.text.x = element_text(vjust = 0.40))
```



Here, we see that the Domain of *Welfare and Quality of Life* was the most dominant in all the manifestos, with the *Economy* coming second. Also, especially UKIP paid a lot of attention to *External Relations*, while the Green party paid little attention to the *Fabric of Society*. In all, this gives us a good idea of what type of data we are dealing with.

Now let's get back to the classification. For this, we need to transform the corpus from the `manifestoR` package into a corpus for the `quanteda` package. To do so, we first have to transform the former into a data frame, and then turn it into a corpus. We then look at the first 10 entries:

```
corpus_data <- mp_corpus(manifestos) %>%
 as.data.frame(with.meta=TRUE)

manifesto_corpus <- corpus(corpus_data)

summary(manifesto_corpus, 10)
```

Here, we see that the corpus treats each sentence as a separate document (which

is confusing). We can still identify to which party they belong due to the `party` variable, which shows the party code. The `cmp_code` variable shows the code assigned to the sentence (here it is all NA as the first sentences have the 0 category). To run the NB, instead of providing our training documents using a vector with NA values, we have to split our data set into a training and a test set. For this, we first generate a string of 8000 random numbers between 0 and 10780 (the total number of sentences). We do so to prevent our training or test set to exist only of sentences from a single party document:

```
set.seed(42)
id_train <- sample(1:10780, 8000, replace = FALSE)
head(id_train, 10)
```

```
##  [1]  2369  5273  9290  1252  8826 10289   356  7700  3954 10095
```

Then we generate a unique number for each of the 10780 sentences in our corpus. This so we can later match them to the sentences we would like to place in our training set or our test set:

```
docvars(manifesto_corpus, "id_numeric") <- 1:ndoc(manifesto_corpus)
```

We should now see this new variable `id_numeric` appear in our corpus. We can now construct our training and test set using these id's. For the training set, the logic is to create a subset of the main corpus and to take only those sentences whose `id_numeric` is also in `id_train`. For the test set, we do the same, only now taking only those sentences whose `id_numeric` is not in `id_train` (note that the `!` mark signifies this). Then, we use the `%>%` pipe to transform the resulting object via a tokens object into a dfm:

```
manifesto_train <- corpus_subset(manifesto_corpus, id_numeric %in% id_train) %>%
 tokens() %>%
 dfm()

manifesto_test <- corpus_subset(manifesto_corpus, !id_numeric %in% id_train) %>%
 tokens() %>%
 dfm()
```

We then run the model using the `textmodel_nb` command, and ask it to use as classifiers the codes in the `cmp_code` variable:

```
manifesto_nb <- textmodel_nb(manifesto_train, docvars(manifesto_train, "cmp_code"))
summary(manifesto_nb)
```

Notice that this command gives us a prediction of how likely it is that an individual word belongs to a certain code (the estimated feature scores). While this can be interesting, what we want to know here is how good the algorithm was. This is when we move from the training of the model using the training set to the prediction of the test set.

A problem is that Naive Bayes can only use features that were both in the

training and the test set. To ensure this happens, we use the `dfm_match` option, which matches all the features in our dfm to a specified vector of features:

```
manifesto_matched <- dfm_match(manifesto_test, features = featnames(manifesto_train))
```

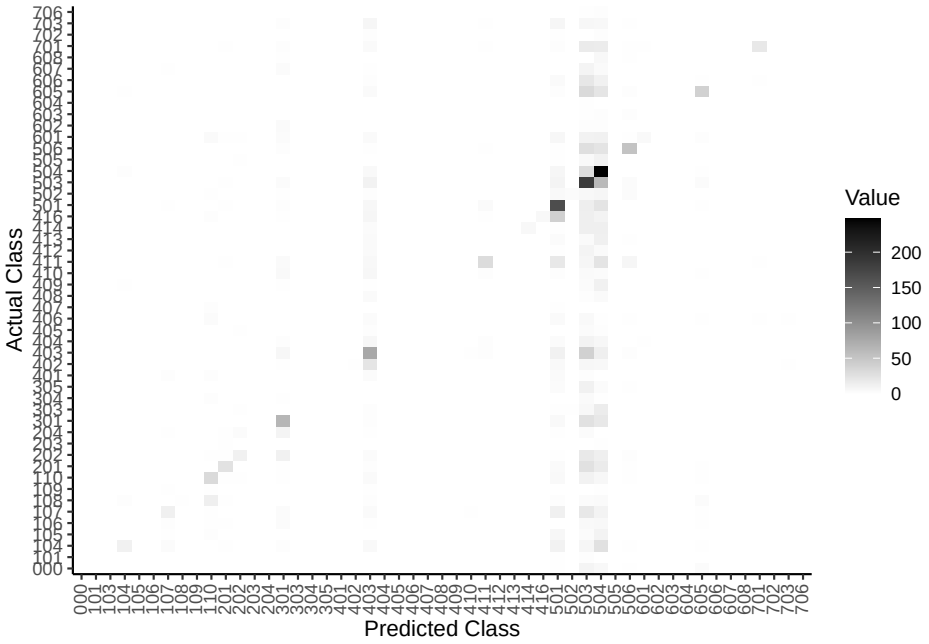
If we look at this new corpus we see that little has changed (there are still 2780 features). This means that all features that were in the test set were also there in the training set. This is good news as this means the algorithm has all the information needed for a good prediction. Yet, the lower the number of sentences, the less likely this is to occur, so matching is always a good idea.

Now we can predict the missing codes in the test set (now the manifesto_matched dfm) using the model we trained earlier. The resulting classes are what the model predicts (we already set this when we trained the model). If we would then open the `predicted_class` object we can see to which code R assigned each sentence. Yet, as before, this is a little too much information. Moreover, we do not want to know what the model assigned the sentence to, but how this corresponds to the original code. To see this, we take the actual classes from the `manifesto_matched` dfm and place them with the predicted classes into a cross table:

```
predicted_class <- predict(manifesto_nb, newdata = manifesto_matched)
actual_class <- docvars(manifesto_matched, "cmp_code")
table_class <- table(actual_class, predicted_class)
table_class
```

While this is already better (we have to pay attention to the diagonal), the large number of codes still makes this hard to read. So, as before, we can better visualise these results - here with the help of a heatmap. To do this, we first transform our table into a data frame that gives us all the possible combinations of codes and their occurrence. We put this into the command and also use a scaling gradient that gets darker when the value in a cell is higher:

```
table_class <- as.data.frame(table_class)

ggplot(data = table_class, aes(x = predicted_class, y = actual_class)) +
 geom_tile(aes(fill = Freq)) +
 scale_fill_gradient(high = "black", low = "white", name="Value")+
 xlab("Predicted Class")+
 ylab("Actual Class")+
 scale_y_discrete(expand = c(0,0)) +
 theme_classic()+
 theme(axis.text.x = element_text(angle = 90))+
 theme(axis.text.x = element_text(vjust = 0.40))
```



Here, we can see that a high number of cases are on the diagonal, which indicates that the algorithm did a good job. Yet, it also classified a large number of sentences into the 503 and 504 categories, while they belonged to any of the other categories.

Besides this, we can also summarize how good the algorithm is through Krippendorff's α. To do so, we take the predicted codes, transform them from factors to numeric values, and store them in an object. Then, we bind them together with the actual codes and place them into a data frame. Finally, we transpose the data frame (so that rows are now columns) and make it into a matrix:

```
predict <- as.numeric(as.character(predicted_class))
reliability <- as.data.frame(cbind(actual_class, predict))
reliability_t <- t(reliability)
reliability <- as.matrix(reliability_t)
```

Then, we load the `kripp.boot` package, and calculate the nominal version of Krippendorff's α, as we are working with nominal codes:

```
library(kripp.boot)
kripp.boot(reliability, iter = 500, method = "nominal")
```

As an alternative, we can use the `DescTools` package:

```
library(DescTools)
KrippAlpha(reliability, method = "nominal")
```

Here we see that the number of subjects was 2780 (the number of sentences in

the test set), the number of coders 2 (the actual and the predicted codes), and the value of α 0.318 with an interval between 0.297 and 0.337. While this might not look particularly encouraging, when we realise that Mikhaylov et al. (2012) estimate the agreement among trained coders by the Manifesto Project to be between 0.350 and 0.400, then 0.305 is quite a good score for a simple model!

Chapter 11

Unsupervised Methods

While supervised models often work fine for text classification, one disadvantage is that we need to set specifics for the model. As an alternative, we can not specify anything and have R find out which classifications work. There are various algorithms to do so, of which we here will focus on Latent Dirichlet Allocation (LDA); a 'seeded' version of LDA that uses information from other sources to improve the results of the LDA; and a Structural Topic Model.

11.1 Latent Dirichlet Allocation

Latent Dirichlet Allocation, or LDA, relies on the idea is that each text is a mix of topics, and each word belongs to one of these. To run LDA, we will use the `topicmodels` package, and use the inaugural speeches as an example. First, we will use the `convert` function to convert the data frequency matrix to a data term matrix as this is what `topicmodels` uses:

```
library(topicmodels)
inaugural_dtm <- convert(data_inaugural_dfm, to = "topicmodels")
```

Then, we fit an LDA model with 10 topics. First, we have to define some a priori parameters for the model. Here, we will use the Gibbs sampling method to fit the LDA model (Griffiths & Steyvers, 2004) over the alternative VEM approach (Blei et al., 2003). Gibbs sampling performs a random walk over the distribution so we need to set a seed to ensure reproducible results. In this particular example, we set five seeds for five independent runs. We also set a burn-in period of 2000 as the first iterations will not reflect the distribution well, and take the 200th iteration of the following 1000:

```
burnin <- 2000
iter <- 1000
thin <- 200
```

```
seed <- list(42,5,24,158,2500)
nstart <- 5
best <- TRUE
```

The LDA algorithm estimates topic-word probabilities as well as topic-document probabilities that we can extract and visualize. Here, we will start with the topic-word probabilities called `beta`. To do this, we will use the `tidytext` package which is part of the tidyverse family of packages. Central to the logic of tidyverse packages is that it does not rely on a document term matrix but represents the data in a long format (Welbers et al., 2017, p. 252). Although this makes it less memory efficient, this lends itself to effective visualisation. The whole logic of these packages is that it works with data which has columns (variables) and rows with single observations. While this is the logic most people know, but it is not always the quickest (and is also not used by `quanteda`). Yet, it always allows you to look at your data in a way most will understand. First, we run the LDA and have a look at the first 10 terms:

```
inaugural_lda10 <- LDA(inaugural_dtm, k=10,
                       method="Gibbs",
                       control=list(burnin=burnin,
                                    iter=iter,
                                    thin=thin,
                                    seed=seed,
                                    nstart=nstart,
                                    best=best))

terms(inaugural_lda10, 10)
```

```
##       Topic 1      Topic 2    Topic 3     Topic 4     Topic 5   Topic 6
##  [1,] "government" "nations"  "women"     "business"  "us"      "freedom"
##  [2,] "justice"    "world"    "love"      "race"      "can"     "liberty"
##  [3,] "progress"   "peace"    "many"      "may"       "new"     "every"
##  [4,] "system"     "free"     "much"      "necessary" "nation"  "americans"
##  [5,] "laws"       "freedom"  "days"      "made"      "america" "still"
##  [6,] "greater"    "peoples"  "story"     "passed"    "must"    "rights"
##  [7,] "economic"   "may"      "moment"    "proper"    "world"   "care"
##  [8,] "prosperity" "people"   "like"      "tariff"    "people"  "came"
##  [9,] "public"     "strength" "americans" "feeling"   "time"    "states"
## [10,] "many"       "united"   "thank"     "increase"  "one"     "ideals"
##       Topic 7      Topic 8    Topic 9      Topic 10
##  [1,] "can"        "life"     "man"        "republic"
##  [2,] "must"       "upon"     "change"     "never"
##  [3,] "government" "men"      "earth"      "war"
##  [4,] "upon"       "shall"    "done"       "may"
##  [5,] "law"        "purpose"  "believe"    "must"
##  [6,] "shall"      "good"     "seek"       "civilization"
```

```
##  [7,] "congress"   "great"    "economy"    "lies"
##  [8,] "country"    "without"  "generation" "order"
##  [9,] "people"     "problems" "union"      "understanding"
## [10,] "states"     "duty"     "small"      "sound"
```

Here, we can see that the first topic is most concerned with words referring to peace and freedom, the second with references to the people, the third with businesses, as so on. While we can interpret our topics this way, a better way might be to visualise the results. For this, we will use the `tidy` command to prepare the dataset for visualisation. Then, we tell the command to use the information from the `beta` column, which contains the probability of a word occurring in a certain topic:

```
library(tidytext)
library(dplyr)
library(ggplot2)

inaugural_lda10_topics <- tidy(inaugural_lda10, matrix="beta")
```

If we would look into the dataset now, we would see that it has 63130 observations with 3 variables. These are the number of the topic, the word (the term) and the `beta` - the chance that the word occurs in that topic. We now want to visualise only the top ten words for each topic in a bar plot. Also, we want the graphs of each of these ten topics to appear in a single graph. To make this happen, we first have to select the top ten words for each topic. We do so again using a pipe (which is the `%>%` command). This pipe transports an output of a command to another one before saving it. So here, we take our data set and group it by topic using the `group_by` command. This command groups the dataset into 10 groups, each for every topic. What this allows us is to calculate things that we otherwise calculate for the whole data-set but here calculate for the groups instead. We then do so and select the top 10 terms (based on their beta value), using `top_n`. We then ungroup again (to make R view it as a single data-set), and use the `arrange` function to ensure the data-set sorts the topics in an increasing and the beta values in a decreasing fashion. Finally, we save this into a new object:

```
inaugural_lda10_topterms <- inaugural_lda10_topics %>%
 group_by(topic) %>%
 top_n(10, beta) %>%
 ungroup() %>%
 arrange(topic, -beta)
```

If we now look at the data set, we see that it is much smaller and has the topics ordered. Yet, before we can plot this we have to ensure that (seen from top to bottom), all the beta for the first topic come first, then for the second topic, and so on. To do so, we use the `mutate` command, and redefine the term variable so that it is re-ordered based first on the term and then on the beta value. The result is a data frame with first the first topic, then the second topic etc., and

with the beta values ordered within each topic. We then make the figure, with the terms on the horizontal axis and the beta values and the vertical axes, and have the bars this generates coloured by topic. Also, we switch off the legend (which we do not need) and use the `facet_wrap` command to split up the total graph (which would have 107 bars otherwise - 107 bars and not a 100 because some terms had the same value for beta). We set the options for the scales to be `free` as it might be that the beta values for some topics are larger or smaller than for the others. Finally, we flip the graphs and make the x-axis the y-axis and vice versa, as this makes the picture more clear:

```
inaugural_lda10_topterms %>%
 mutate(term=reorder(term, beta)) %>%
 ggplot(aes(term, beta, fill=factor(topic))) +
 geom_col(show.legend=FALSE) +
 facet_wrap(~ topic, scales="free") +
 coord_flip()+
 theme_minimal()
```

What is clear here is that looking at only the words in each topic only says so much. In the first topic, the term 'peace' is more important than anything else,

and so is 'us' in topic number 2. Also, in topic number ten, we see that both 'first' and 'need' are of equal importance.

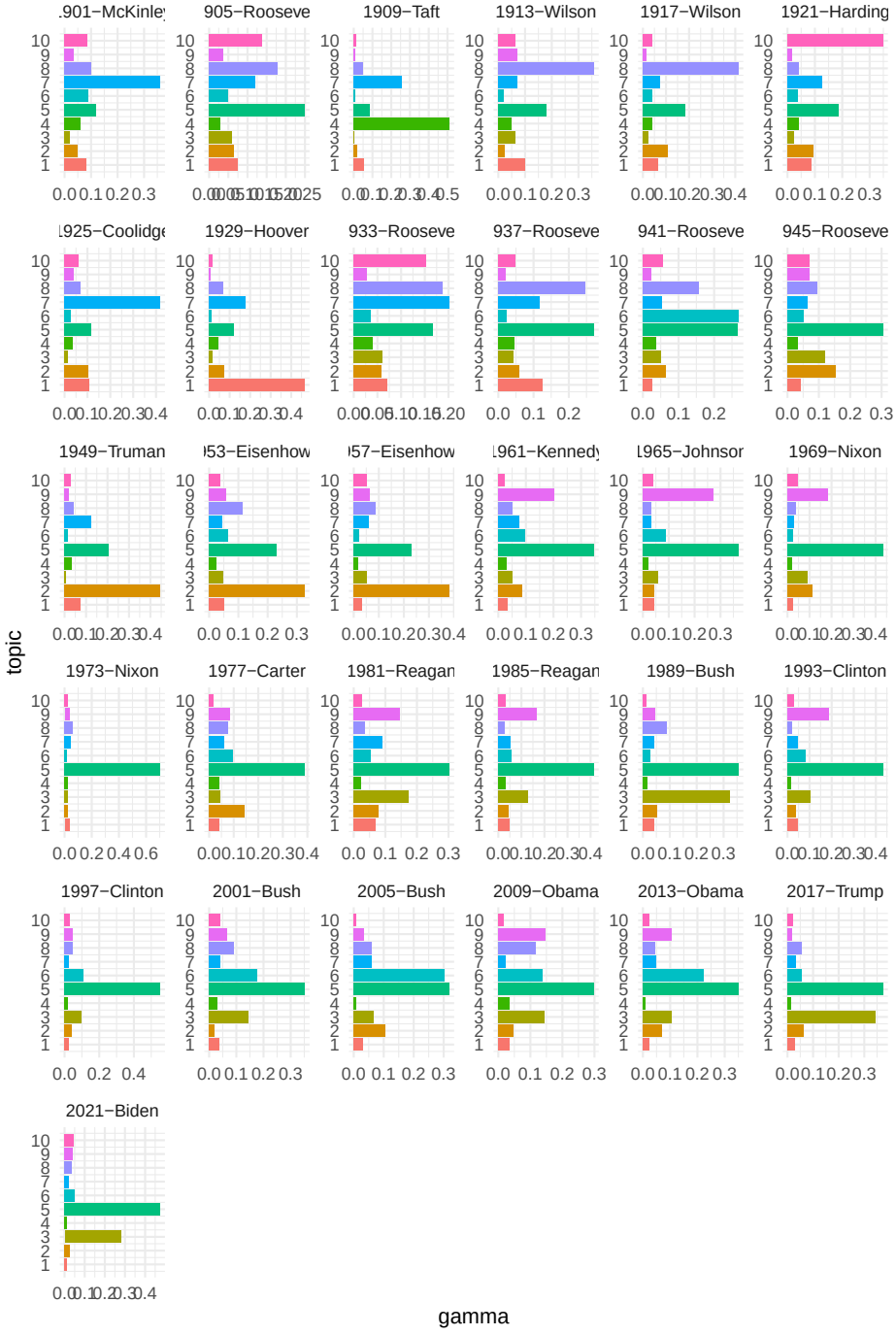
Another question we can ask is how much of each topic is in each of the documents. Put in another way: do certain documents talk more about certain topics than others? To see this, we first generate a new data frame with this information, known as the `gamma` value for each document:

```
inaugural_lda10_documents <- tidy(inaugural_lda10, matrix="gamma")
```

We then go through similar steps to make the data set ready for use and prepare the graph. For the graph, the only steps we do different are to force R to label each topic on the axis (as otherwise it will treat it as a continuous variable and come up with useless values such as 7.5), and to give it a different look (using the `theme_classic()` command):

```
inaugural_lda10_toptopics <- inaugural_lda10_documents %>%
 group_by(document) %>%
 top_n(10, gamma) %>%
 ungroup() %>%
 arrange(topic, -gamma)
```

```
inaugural_lda10_toptopics %>%
 mutate(term=reorder(topic, gamma)) %>%
 ggplot(aes(topic, gamma, fill=factor(topic))) +
 geom_col(show.legend=FALSE) +
 scale_x_continuous(breaks = c(1,2,3,4,5,6,7,8,9,10))+
 facet_wrap(~ document, scales="free") +
 coord_flip()+
 theme_minimal()
```



Here, we see that in 1929 Hoover talked most often about topic 9 (focusing on government), Biden in 2021 focused on words like 'us' and 'people', while in

1945 Roosevelt seemed to favour both the people and topics referring to peace. Again, our exact conclusions of course depend on how we interpret the topics.

11.2 Seeded Latent Dirichlet Allocation

An alternative to the above approach is one known as seeded-LDA. This approach uses seed words that can steer the LDA in the right direction. One origin of these seed words can be a dictionary that tells the algorithm which words belong together in various categories. To use it, we will first load the packages and set a seed:

```
library(seededlda)
library(quanteda.dictionaries)

set.seed(42)
```

Next, we need to specify a selection of seed words in dictionary form. While we can construct a dictionary ourselves, here we choose to use the Laver and Garry dictionary we saw earlier. We then use this dictionary to run our seeded LDA:

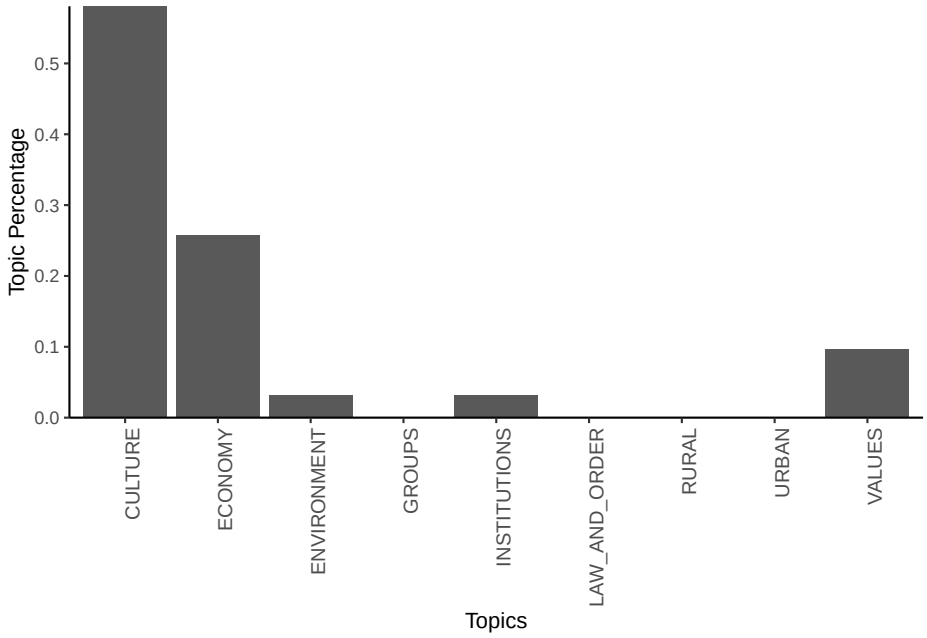
```
dictionary_LaverGarry <- dictionary(data_dictionary_LaverGarry)
seededmodel <- textmodel_seededlda(data_inaugural_dfm, dictionary = dictionary_LaverGa
terms(seededmodel, 20)
```

```
##       CULTURE     ECONOMY       ENVIRONMENT     GROUPS      INSTITUTIONS
##  [1,] "people"    "work"        "civilization"  "women"     "president"
##  [2,] "us"        "government"  "production"    "race"      "administration"
##  [3,] "new"       "can"         "productive"    "day"       "executive"
##  [4,] "america"   "upon"        "republic"      "story"     "continue"
##  [5,] "nation"    "great"       "population"    "thank"     "office"
##  [6,] "world"     "must"        "war"           "back"      "business"
##  [7,] "must"      "may"         "order"         "bless"     "congress"
##  [8,] "can"       "shall"       "produce"       "president" "policy"
##  [9,] "time"      "economic"    "tasks"         "schools"   "legislation"
## [10,] "let"       "justice"     "planet"        "around"    "law"
## [11,] "one"       "opportunity" "products"      "yes"       "modern"
## [12,] "today"     "children"    "making"        "across"    "rule"
## [13,] "now"       "country"     "conditions"    "hand"      "authority"
## [14,] "every"     "united"      "productivity"  "left"      "race"
## [15,] "make"      "war"         "relations"     "friends"   "necessary"
## [16,] "americans" "progress"    "promote"       "began"     "agencies"
## [17,] "american"  "men"         "maintained"    "lost"      "make"
## [18,] "years"     "never"       "understanding" "racial"    "proper"
## [19,] "together"  "economy"     "leadership"    "young"     "reforms"
## [20,] "spirit"    "made"        "normal"        "founding"  "voices"
##       LAW_AND_ORDER   RURAL          URBAN        VALUES
##  [1,] "force"         "public"       "man"        "freedom"
```

```
##  [2,] "determined"    "law"           "price"      "history"
##  [3,] "forces"        "party"         "sides"      "human"
##  [4,] "men"           "permanent"     "begin"      "peace"
##  [5,] "determination" "toward"        "growth"     "free"
##  [6,] "day"           "enforcement"   "loyalty"    "world"
##  [7,] "court"         "agricultural"  "covenant"   "nations"
##  [8,] "counsel"       "relations"     "sick"       "rights"
##  [9,] "every"         "direction"     "compassion" "principles"
## [10,] "something"     "nation"        "final"      "past"
## [11,] "evil"          "establishment" "heal"       "help"
## [12,] "mind"          "islands"       "globe"      "life"
## [13,] "conviction"    "countrymen"    "goals"      "faith"
## [14,] "terror"        "stability"     "passed"     "strength"
## [15,] "necessity"     "independence"  "call"       "live"
## [16,] "life"          "feed"          "understood" "know"
## [17,] "dealing"       "provided"      "suffer"     "peoples"
## [18,] "body"          "civilization"  "trying"     "security"
## [19,] "democratic"    "agriculture"   "mountains"  "humanity"
## [20,] "determine"     "ideals"        "assure"     "leadership"
```

Note that using the dictionary has ensured that we only use the categories that occur in the dictionary. This means that we can look at which topics are in each inaugural speech and which terms were most likely for each of the topics. Let us start with the topics first:

```
topics <- topics(seededmodel)
topics_table <- ftable(topics)
topics_prop_table <- as.data.frame(prop.table(topics_table))

ggplot(data=topics_prop_table, aes(x=topics, y=Freq))+
 geom_bar(stat="identity")+
 labs(x="Topics", y="Topic Percentage")+
 scale_y_continuous(expand = c(0, 0)) +
 theme_classic()+
 theme(axis.text.x = element_text(size=10, angle=90, hjust = 1))
```



Here, we find that Culture was the most favoured topic, followed by the Economy and Values. Finally, we can then have a look at the most likely terms for each topic, sorted by each of the categories in the dictionary:

```
terms <- terms(seededmodel)
terms_table <- ftable(terms)
terms_df <- as.data.frame(terms_table)
head(terms_df)
```

```
##   Var1    Var2    Freq
## 1    A CULTURE  people
## 2    B CULTURE      us
## 3    C CULTURE     new
## 4    D CULTURE america
## 5    E CULTURE  nation
## 6    F CULTURE   world
```

Here, we find that in the first cluster (denoted as 'A'), the word 'people' was most likely (from all words that belonged to Culture). Thus, within this cluster, talking about culture often contained references to the people. In the same way, we can make similar observations for the other categories.

11.3 Structural Topic Model

Besides LDA, various other methods for unsupervised classification exist, such as hierarchical clustering, k-means, and various other mixed membership models.

Each of them has its specific advantages and problems, and it often depends on the goal of the researcher to decide which method to use. One new and flexible method is the Structural Topic Model or STM. In R, we can find this method in the `stm` package (Roberts et al., 2019).

One of the outstanding features of stm is *topical prevalence*. This means that we can include covariates to help identify the correct model and better understand the topics the model generates (Roberts et al., 2014). For example, we can add information on time to study how topics change over the years; actors on how they differ between different authors; and any other possible variable to see how they differ between them. One of the main advantages of STM is that, unlike in LDA, we are not required to set any parameters in advance. In LDA, these parameters - α (the degree of mixture of topics a document has) and β (the degree of mixture of words that a topic has) - have to be set beforehand based on previous knowledge. Yet, this knowledge is not always present and we often need several iterations before we settle upon a correct number. In STM, we use the metadata to set these parameters.

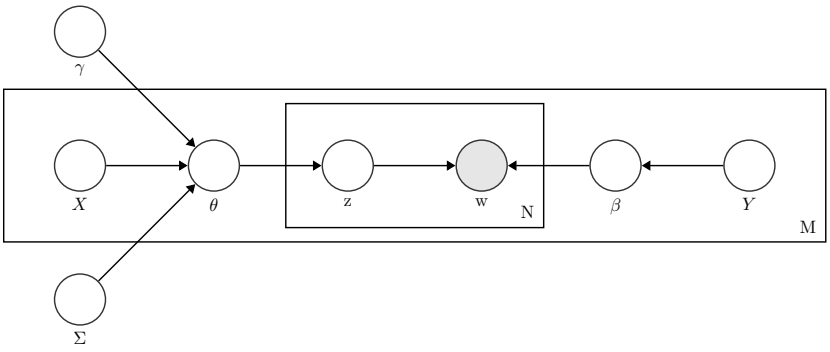


Figure 11.1: Plate diagram for a Structucal Topic Model.

Figure 11.1 shows stm in the form of a plate diagram. Here, X refers to the prevalence metadata; γ, the metadata weights; Σ, the topic covariances; θ, the document prevalence; z, the per-word topic; w, the observed word; Y, the content metadata; β, the topic content; N, the number of words in a document; and M, the number of documents in the corpus.

To run stm in R, we have to load the package, set a seed, convert our dfm to the stm format and place our documents, vocabulary (the tokens) and any other data in three separate objects (for later convenience):

```
library(stm)
library(quanteda)

set.seed(42)

data_inaugural_stm <- convert(data_inaugural_dfm, to = "stm")

documents <- data_inaugural_stm$documents
vocabulary <- data_inaugural_stm$vocab
meta <- data_inaugural_stm$meta
```

The first thing we have to do is find the number of topics we need. In the `stm` package, we can do this by using a function called `searchK`. Here, we specify a range of values that could include the 'correct' number of topics, which we then run and collect. Afterwards, we then look at several goodness-of-fit measures to assess which number of topics (which k) has the best fit for the data. These measures include exclusivity, semantic coherence, held-out likelihood, bound, lbound, and residual dispersion. Here, we run this for 2 to 15 possible topics.

In our code, we specify our documents, our tokens (the vocabulary), and our meta-data. Moreover, as our prevalence, we include parameters for `Year` and `Party`, as we expect the content of the topics to differ between both the Republican and Democratic party, as well as over time:

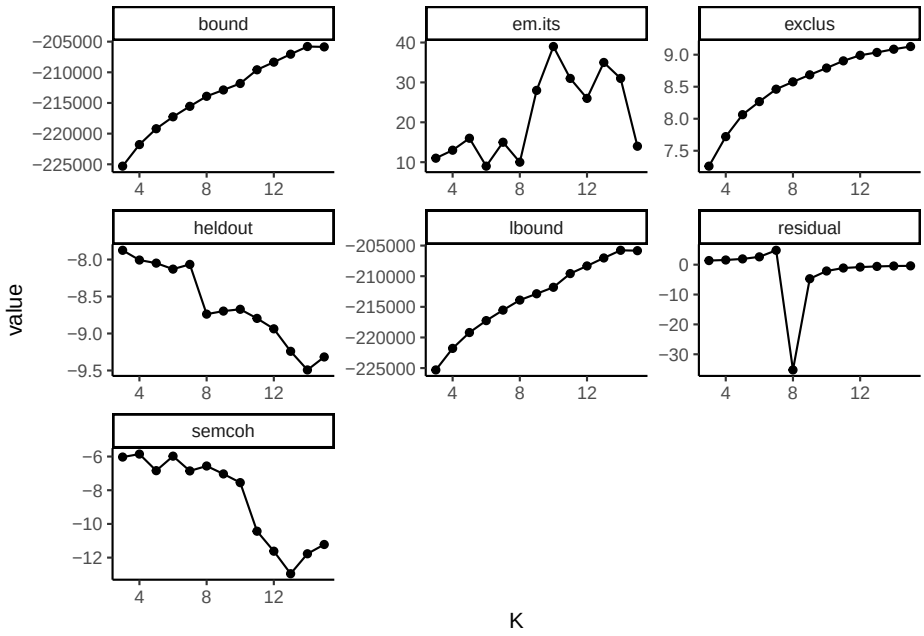
```
k <- c(3,4,5,6,7,8,9,10,11,12,13,14,15)
```

```
findingk <- searchK(documents, vocabulary, k, prevalence =~ Party + s(Year), data = met
```

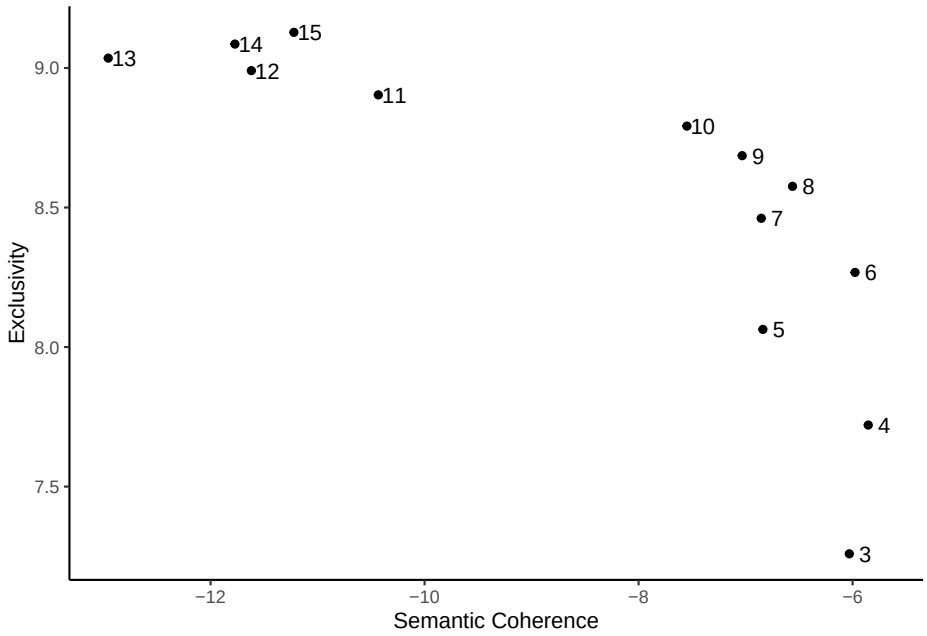
```
findingk_results <- as.data.frame(matrix(unlist(findingk$results), nrow=length(unlist(
names <- names(findingk$results)
names(findingk_results) <- names
```

Looking at `findingk_results` we find various values. The first, exclusivity, refers to the occurrence that when words have a high probability under one topic, they have a low probability under others. Related to this is semantic coherence which happens when the most probable words in a topic should occur in the same document. Held-out (or held-out log-likelihood) is the likelihood of our model on data that was not used in the initial estimation (the lower the better), while residuals refer to the difference between a data point and the mean value that the model predicts for that data point (which we want to be 1, indicating a standard distribution). Finally, bound and lbound refer to a model's internal measure of fit. Here, we will be looking for the number of topics, that balance the exclusivity and the semantic coherence, have a residual around 1, and a low held-out. To make this simpler, we visualise our data. In the first graph we plot all the values, while in the second, we only look at the exclusivity and the semantic coherence (as they are the most important):

```
library(reshape2)

findingk_melt <- melt(findingk_results, id="K")
findingk_melt$variable <- as.character(findingk_melt$variable)
findingk$K <- as.factor(findingk_results$K)

ggplot(findingk_melt, aes(K, value)) +
 geom_point()+
 geom_line()+
 facet_wrap(~ variable, scales = "free")+
 theme_classic()
```




```
ggplot(findingk_results, aes(semcoh, exclus)) +
 geom_point()+
 geom_text(data=findingk_results, label=findingk$K, nudge_x = 0.15)+
 scale_x_continuous("Semantic Coherence")+
 scale_y_continuous("Exclusivity")+
 theme_classic()
```



Based on these graphs, we decide upon 8 topics. The main reason for this is that for this number of topics, there is a high semantic coherence given the exclusivity. We can now run our stm model, using spectral initialization and a topical prevalence including both the Party and the Year of the inauguration. Also, we have a look at the topics, and the words with the highest probability attached to them:

```
n_topics <- 8
output_stm <- stm(documents, vocabulary, K = n_topics, prevalence =~ Party + s(Year), c
```

```
labelTopics(output_stm)
```

```
## Topic 1 Top Words:
##       Highest Prob: free, peace, world, shall, freedom, must, faith
##       FREX: strive, free, peoples, everywhere, truth, man's, learned
##       Lift: abhorring, absorbing, abstractions, acquire, aggressor, amass, andes
##       Score: anguished, productivity, strive, trial, learned, europe, defines
## Topic 2 Top Words:
##       Highest Prob: us, new, world, let, can, people, america
##       FREX: let, century, together, new, weapons, voices, abroad
##       Lift: 200th, 20th, dawn, explore, micah, moon, music
##       Score: attempting, nuclear, let, celebrate, voices, abroad, dawn
## Topic 3 Top Words:
##       Highest Prob: us, must, world, government, people, america, can
##       FREX: civilization, republic, experiment, normal, relationship, order, industr
##       Lift: deliberate, inspiration, intention, regards, tremendous, unshaken, abnor
```

```
##       Score: accompanied, supreme, regards, deliberate, inspiration, unshaken, righteousness
## Topic 4 Top Words:
##       Highest Prob: us, america, nation, can, must, new, people
##       FREX: story, thank, president, defend, everyone, children, america
##       Lift: breeze, january, accounting, accumulate, activism, addiction, agitate
##       Score: allowing, story, breeze, talk, crucial, everyone, virus
## Topic 5 Top Words:
##       Highest Prob: freedom, nation, people, america, government, know, democracy
##       FREX: speaks, mind, democracy, liberty, seen, came, millions
##       Lift: paint, abreast, absence, admiration, agent, amount, aspires
##       Score: charta, speaks, paint, disaster, mind, defended, seen
## Topic 6 Top Words:
##       Highest Prob: us, must, nation, people, can, new, every
##       FREX: generation, journey, union, change, covenant, creed, enduring
##       Lift: demanded, mastery, span, storms, absolutism, abundantly, afghanistan
##       Score: abundantly, covenant, journey, mastery, storms, demanded, span
## Topic 7 Top Words:
##       Highest Prob: can, world, people, peace, nations, must, government
##       FREX: settlement, enforcement, countries, desire, party, international, property
##       Lift: abound, absurd, acceptance, accepts, accordingly, accountability, accounted
##       Score: abound, enforcement, contributed, settlement, property, major, eighteenth
## Topic 8 Top Words:
##       Highest Prob: upon, government, shall, can, must, great, may
##       FREX: army, interstate, negro, executive, tariff, business, proper
##       Lift: affected, amendments, antitrust, army, attention, avail, banking
##       Score: tariff, interstate, army, negro, policy, proper, business
```

Here, we see that the word `us` is dominant in most topics, making it a candidate for removal as a stop word in a future analysis. Looking closer, we find that the first topic refers to peace, the second, third and seventh to the world, the fourth and sixth to America, and the eighth to the government.

Finally, we can see whether there is any relation between these topics and any of the parameters we included. Here, let us look at any existing differences between the two parties:

```
est_assoc_effect <- estimateEffect(~Party, output_stm, metadata = meta, prior=1e-5)
```

While we can visualise this with the `plot.estimateEffect` option, the visualisation is far from ideal. Thus, let us use some data-wrangling and make the plot ourselves:

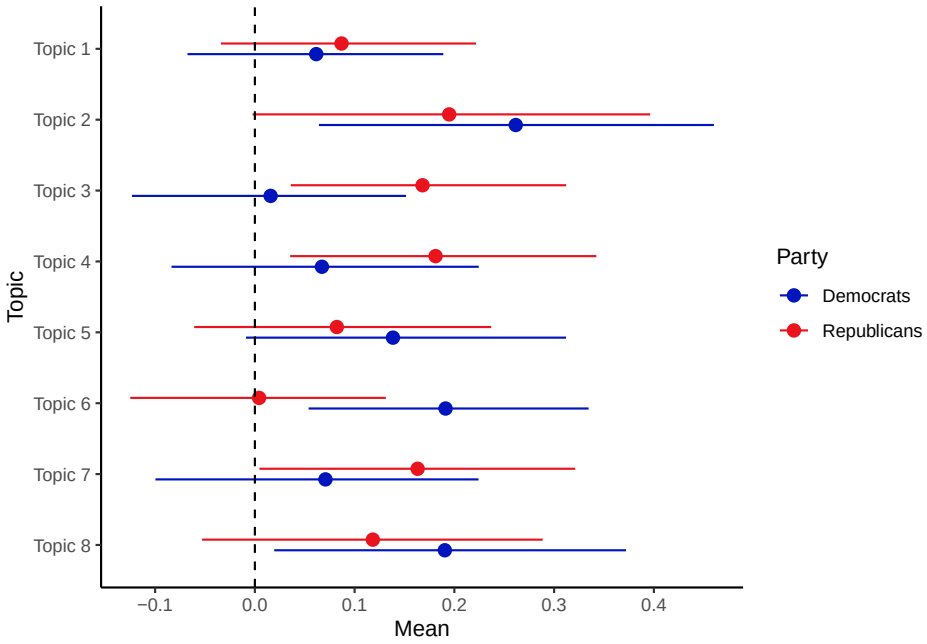
```
estimate_data <- plot.estimateEffect(est_assoc_effect, "Party", method = "pointestimate", model =
estimate_graph_means <- estimate_data$means
estimate_graph_means <- data.frame(matrix(unlist(estimate_graph_means), nrow=length(estimate_grap
estimate_graph_means <- data.frame(c(rep("Republicans", 8), rep("Democrats", 8)), c(estimate_grap

estimate_graph_cis <- estimate_data$cis
```

```
estimate_graph_cis <- data.frame(matrix(unlist(estimate_graph_cis), nrow=length(estimat
estimate_graph_cis <- data.frame(c(estimate_graph_cis$X1,estimate_graph_cis$X3), c(esti

Topic <- c("Topic 1", "Topic 2", "Topic 3", "Topic 4","Topic 5", "Topic 6", "Topic 7",

estimate_graph <- cbind(Topic, estimate_graph_means,estimate_graph_cis)
names(estimate_graph) <- c("Topic","Party","Mean", "min", "max")
estimate_graph$Party <- as.factor(estimate_graph$Party)
estimate_graph$Topic <- as.factor(estimate_graph$Topic)
estimate_graph$Topic <- factor(estimate_graph$Topic, levels=rev(levels(estimate_graph$T
```

Now, let us plot our intervals:

```
ggplot(estimate_graph, aes(Mean, Topic)) +
 geom_pointrange(aes(xmin = min, xmax = max, color = Party),
                 position = position_dodge(0.3))+
 geom_vline(xintercept = 0,
            linetype="dashed", size=0.5)+
 scale_color_manual(values = c("#0015BC", "#E9141D"))+
 theme_classic()
```

```
## Warning: Using `size` aesthetic for lines was deprecated in ggplot2 3.4.0.
## i Please use `linewidth` instead.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning
## was generated.
```



Here, we find that while the averages for the topic do seem to differ a little between both of the parties, all the intervals are overlapping, indicating that they are not that different.

Chapter 12

Texttricks

One of the main advantages of using R for quantitative text analysis is that it is (for a part) a general-purpose language. That is, we are not limited to any of the functions that the original designers chose to implement with it. Indeed, this is the reason for the existence of the all the packages we have been using. Indeed, writing a package in R is relatively simple, as is hosting your package and making it available to others.

The main logic of R packages rests on the idea of *functions*. We already saw an example of a function in Chapter 4, where we specified a function to read in a .pdf file and convert it to a .txt file. The way a function in R works it that you encapsulate the commands you require within the `function` command. Anything between the curly brackets will be treated as part of the function, while everything between the parentheses will be treated as the input of our function.

```
extract_pdf <- function(filename) {
  require(pdftools)
  print(filename)
  try({
    text <- pdf_text(filename)
  })
  title <- gsub("(.*)/([^/]*).pdf", "\\2", filename)
  txt_directory <- getwd()
  write(text, file.path(txt_directory, paste0(title, ".txt")))
}
```

- *Function Name* - The name of the function and what we need to type into the console in order to run the function
- *Arguments* - Arguments are placeholders. In our case, `filename` is an argument. This can be anything, and in our case it is the address of where our .pdf file is stored. When we run the function, anything we specify in

the place of `filename` will be treated as such by R.
- *Body* - Everything between the curly brackets that makes up the complete function
- *Return Value* - The output of the function. Can be specified by `return(value)`. Yet, in our case the return is in the `write()` command that writes the .txt file to our `txt_directory` folder.

Compiling this function (or many different functions) into a package, requires a little bit more work, but not much. A very nice introduction is given by Hilary Parker. Here, we placed the function above in the `texttricks` package. As this package is on GitHub, you can have a look at all its files at: https://github.com/SCJBruinsma/texttricks.

To install the package within `R`, use the `install_github` command that is part of the devtools package:

```
install.packages("devtools")
library(devtools)
devtools::install_github("SCJBruinsma/texttricks")
```

Albaugh, Q., Sevenans, J., Soroka, S., & Loewen, P. J. (2013). The Automated Coding of Policy Agendas: A Dictionary-based Approach. *6th Annual Comparative Agendas Conference, Antwerp, Belgium.*

Bakker, R., Vries, C. de, Edwards, E., Hooghe, L., Jolly, S., Marks, G., Polk, J., Rovny, J., Steenbergen, M. R., & Vachudova, M. A. (2012). Measuring party positions in europe: The chapel hill expert survey trend file, 1999-2010. *Party Politics*, *21*(1), 1–15. https://doi.org/10.1177/1354068812462931

Benoit, K., Laver, M., & Mikhaylov, S. (2009). Treating words as data with error: Uncertainty in text statements of policy positions. *American Journal of Political Science*, *53*(2), 495–513. https://doi.org/10.1111/j.1540-5907.2009.00383.x

Benoit, K., Watanabe, K., Wang, H., Nulty, P., Obeng, A., Müller, S., & Matsuo, A. (2018). Quanteda: An r package for the quantitative analysis of textual data. *Journal of Open Source Software*, *3*(30), 774. https://doi.org/10.21105/joss.00774

Blei, D. M., Ng, A. Y., & Jordan, M. I. (2003). Latent dirichlet allocation. *Journal of Machine Learning Research*, *3*(Jan), 993–1022.

Bruns, A. (2019). After the 'APIcalypse': Social media platforms and their fight against critical scholarly research. *Information, Communication & Society*, *22*(11), 1544–1566. https://doi.org/10.1080/1369118X.2019.1637447

Carmines, E. G., & Zeller, R. A. (1979). *Reliability and validity assessment*. Sage. https://doi.org/10.4135/9781412985642

Clarke, I., & Grieve, J. (2019). Stylistic variation on the donald trump twitter account: A linguistic analysis of tweets posted between 2009 and 2018. *PLOS ONE*, *14*(9), 1–27. https://doi.org/10.1371/journal.pone.0222062

Freelon, D. (2018). Computational Research in the Post-API Age. *Political Communication*, *35*(4), 665–668. https://doi.org/10.1080/10584609.2018.1477506

Griffiths, T. L., & Steyvers, M. (2004). Finding scientific topics. *Proceedings of the National Academy of Sciences*, *101*(suppl 1), 5228–5235. https://doi.org/10.1073/pnas.0307752101

Grimmer, J., Roberts, M. E., & Stewart, B. M. (2022). *Text as Data: A New Framework for Machine Learning and the Social Sciences*. Princeton University Press.

Grimmer, J., & Stewart, B. M. (2013). Text as data: The promise and pitfals of automatic content analysis methods for political texts. *Political Analysis*, *21*(3), 267–297. https://doi.org/10.1093/pan/mps028

Haselmayer, M., & Jenny, M. (2017). Sentiment analysis of political communication: Combining a dictionary approach with crowdcoding. *Quality & Quantity*, *51*(6), 2623–2646. https://doi.org/10.1007/s11135-016-0412-4

Hayes, A. F., & Krippendorff, K. (2007). Answering the call for a standard reliability measure for coding data. *Communication Methods and Measures*, *1*(1), 77–89. https://doi.org/10.1080/19312450709336664

Hutto, C., & Gilbert, E. (2014). VADER: A parsimonious rule-based model for sentiment analysis of social media text. *Proceedings of the International*

AAAI Conference on Web and Social Media, *8*(1), 216–225. https://doi.org/10.1609/icwsm.v8i1.14550

Krippendorff, K. (2018). *Content Analysis - an Introduction to Its Methodology* (4th ed.). Sage.

Lamprianou, I. (2020). Measuring and visualizing coders' reliability: New approaches and guidelines from experimental data. *Sociological Methods & Research*. https://doi.org/10.1177/0049124120926198

Laver, M., Benoit, K., & Garry, J. (2003). Extracting policy positions from political texts using words as data. *The American Political Science Review*, *97*(2), 311–331. https://doi.org/10.1017/S0003055403000698

Laver, M., & Garry, J. (2000). Estimating policy positions from political texts. *American Journal of Political Science*, *44*(3), 619–634. https://doi.org/10.2307/2669268

Lê, S., Josse, J., & Husson, F. (2008). Factominer: An r package for multivariate analysis. *Journal of Statistical Software*, *25*(1). https://doi.org/10.18637/jss.v025.i01

Lin, L. (1989). A concordance correlation coefficient to evaluate reproducibility. *Biometrics*, *45*, 255–268. https://doi.org/10.2307/2532051

Lind, F., Eberl, J.-M., Heidenreich, T., & Boomgaarden, H. G. (2019). When the journey is as important as the goal: A roadmap to multilingual dictionary construction. *International Journal of Communication*, *13*, 4000–4020.

Lowe, W. (2011). *JFreq: Count Words, Quickly*. http://www.conjugateprior.org/software/jfreq/

Lowe, W., & Benoit, K. (2011). Estimating uncertainty in quantitative text analysis. *Annual Meeting of the Midwest Political Science Association*.

Martin, L. W., & Vanberg, G. (2008). Reply to benoit and laver. *Political*

Analysis, *16*(1), 112–114. https://doi.org/10.1093/pan/mpm018

Merz, N., Regel, S., & Lewandowski, J. (2016). The manifesto corpus: A new resource for research on political parties and quantitative text analysis. *Research & Politics*, *3*(2), 2053168016643346. https://doi.org/10.1177/2053168016643346

Mikhaylov, S., Laver, M., & Benoit, K. (2012). Coder reliability and misclassification in the human coding of party manifestos. *Political Analysis*, *20*(1), 78–91. https://doi.org/10.1093/pan/mpr047

Munzert, S., Rubba, C., Meißner, P., & Nyhuis, D. (2014). *Automated data collection with r: A practical guide to web scraping and text mining*. John Wiley & Sons.

Perriam, J., Birkbak, A., & Freeman, A. (2020). Digital Methods in a Post-API Environment. *International Journal of Social Research Methodology*, *23*(3), 277–290. https://doi.org/10.1080/13645579.2019.1682840

Roberts, M. E., Stewart, B. M., & Tingley, D. (2019). stm: An R Package for Structural Topic Models. *Journal of Statistical Software*, *91*(2). https://doi.org/10.18637/jss.v091.i02

Roberts, M. E., Stewart, B. M., Tingley, D., Lucas, C., Leder-Luis, J., Gadarian, S. K., Albertson, B., & Rand, D. G. (2014). Structural topic models for open-ended survey responses. *American Journal of Political Science*, *58*(4), 1064–1082. https://doi.org/10.1111/ajps.12103

Slapin, J. B., & Proksch, S.-O. (2008). A scaling model for estimating time-series party positions from texts. *American Journal of Political Science*, *52*(3), 705–722. https://doi.org/10.1111/j.1540-5907.2008.00338.x

Volkens, A., Krause, W., Lehmann, P., Matthieß, T., Merz, N., Regel, S., & Weßels, B. (2019). *The Manifesto Data Collection. Manifesto Project (MRG/CMP/MARPOR)*. Berlin: Wissenschaftszentrum Berlin für Sozialforschung (WZB). https://doi.org/10.25522/manifesto.mpds.2019b

Welbers, K., Van Atteveldt, W., & Benoit, K. (2017). Text Analysis in R. *Communication Methods and Measures*, *11*(4), 245–265. https://doi.org/10.1080/19312458.2017.1387238

Young, L., & Soroka, S. (2012). *Lexicoder sentiment dictionary*. http://www.snsoroka.com/data-lexicoder/