Introduction to Quantitative Text Analysis

Kostas Gemenis and Bastiaan Bruinsma

Contents

Welcome!

Welcome to our textbook on quantitative text analysis! This book originated as a collection of assignments and lecture slides 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. We have updated the version you see today with the help of a grant from the Learning Development Network 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? Over the last twenty years, developments have made research using quantitative text analysis a fascinating proposition. First, the massive 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.

However, quantitative text analysis can be daunting for someone unfamiliar with quantitative methods or programming. Therefore, in this book, we aim to guide you through it, combining theoretical explanations with step-by-step explanations of the code. We also designed several exercises to practise for those with little or no experience in text analysis, R, or quantitative methods. Ultimately, we hope this book is informative, engaging, and educational.

This book has 8 chapters, each of which focuses on a specific aspect of quantitative text analysis in R. While we wrote them with the idea that you will read them in sequence, you are, of course, free to skip to any chapter you like.

---

1. **Chapter** 1 will discuss all the preliminaries and tell you how to install

R, download and write packages, and what to do when encountering an error.

2. **Chapter** 2 discusses the basics and background of quantitative text analysis and looks at the various steps of the average analysis. Besides, we briefly discuss the concepts of validity, reliability and validation and why they are essential for quantitative text analysis.

3. **Chapter** 3 looks at how to import textual data into R. We will look at how R handles data in general, how we can import files such as .txt, .pdf, and .csv, and how we can use an API, JSON, or databases such as MySQL or SQLite.

4. **Chapter** 4 shows the different ways we can use to understand what the texts in our data are about. We will also introduce the idea of the *corpus* as the central element of most QTA analyses and the *Data Frequency Matrix* (*DFM*), which we derive from it. We will also look at pre-processing, descriptives and various text statistics such as keyness and entropy.

5. **Chapter** 5 introduces dictionaries, including classical dictionary analysis, sentiment analysis and dictionary analysis using VADER.

6. **Chapter** 6 looks at methods to scale texts on one or more dimensions. We look at three of the most popular methods to do so: *Wordscores*, *Wordfish*, and *Correspondence Analysis*.

7. **Chapter** 7 discusses methods for supervised analysis, where we train an algorithm on a particular set of texts to classify others. We look at *Support Vector Machines (SVMs)*, *Regularised Logistic Regression*, and *Naive Bayes (NB)*. In addition, we also discuss how to evaluate and validate our classifications and several other points we should keep in mind.

8. **Chapter** 8 is about unsupervised methods, where we have an algorithm to find categories and structure in our texts. Here, we look at the most popular way to do so – *Latent Dirichlet Analysis (LDA)* – its *seeded LDA* form, the *Structural Topic Model* derived from it, and *Latent Semantic Analysis (LSA)*. Again, we will also look at various ways to validate our findings.

If you want to learn more about text analysis, there is no shortage of good introductions available these days. Which book is best for you, therefore, depends mainly 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 (Krippendorff, 2019). 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 (Grimmer et al., 2022). 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 discuss here in passing) (Neuendorf, 2016). These texts, among others, form the bedrock of the field and offer different perspectives on the theory and practice of making sense of texts using quantitative methods.

Chapter 1

Preliminaries

As all the analyses in this book will be carried out using R, this chapter will discuss what R is, how to install it and which packages are required. If you already have a functioning R environment on your system, you can safely skip this chapter, though be sure to check Section 1.3 to ensure that you have all the necessary packages.

1.1 Installing R

R is free, open-source, maintained by a large community, available for all major operating systems, including Windows, macOS, and Linux, and receives regular updates. In its most basic form, it runs in a simple command-line interface (CLI) that, while functional, can be less than intuitive. As a result, integrated development environments (IDEs) are popular, providing a graphical interface with helpful features such as code highlighting, auto-completion, built-in help, plotting windows, and file management. Several IDEs are available for R, including Jupyter (popular for notebooks), R Commander, and RStudio, the latter being the most popular and the one we recommend you use. In fact, we wrote this book with RStudio in mind, and wrote the book using it.

The way you install R and RStudio depends on your system. Note that these instructions may change in the future (especially when it comes to dependencies).

1.1.1 Windows

Install R:

- Go to the CRAN download page for Windows.
- Click the link to download the latest version of R.
- Run the downloaded `.exe` installer file.

- Accept the default installation settings regarding components and the destination folder, as this makes it easier for other programs (like RStudio) to find R. Modern R installers for Windows are 64-bit versions (as most modern Windows systems are 64-bit). If you want a 32-bit version, R Version 4.2.0 is the last that supported it, which you can download from the page containing all the previous releases.

Install RStudio:

- Go to the RStudio Desktop download page.
- Scroll down to the *Installers for Supported Platforms* section.
- Download the version for Windows.
- Run the downloaded `.exe` installer file. Again, it is best to accept the default installation settings. RStudio should automatically detect your R installation.

A Note on File Paths in Windows: Unlike Windows, which uses backslashes (\) in file paths (e.g., `C:\Users\Documents\data.csv`), R (like most Unix systems) uses forward slashes (`/`) (e.g., `C:/Users/Documents/data.csv`). Remember to use forward slashes when specifying file paths within R code, regardless of how they appear in your Windows file explorer.

1.1.2 Linux

Install R:

In most cases, R is already part of your Linux distribution. You can check this by opening a terminal and typing `R`, which will launch it in the terminal if installed. If R is not part of your system, you can install it using your distribution's package manager (such as APT or snap). However, the version of R in these default repositories may not be the latest. To get the latest version (recommended for compatibility with the latest packages), it's often best to add the official CRAN repository to your system's update sources (see here for detailed instructions on how to do so). Alternatively, if you prefer a GUI, you can use the Synaptic Package Manager and look for the `r-base-dev` and `r-base` packages, select them, and install them.

Install RStudio:

- Go to the RStudio Desktop download page.
- Scroll down to the *Installers for Supported Platforms* section.
- Download the installer package appropriate for your distribution (e.g., `.deb` for Debian/Ubuntu, `.rpm` for Fedora/CentOS).
- Install the downloaded file using your distribution's package installer (e.g., double-clicking the `.deb` file or using `sudo install rstudio-*-amd64.deb` in the terminal, followed by `sudo apt --fix-broken install` if there are dependency issues).

Once installed, you should find RStudio in your application menu or by typing `rstudio` in the terminal.

1.1.3 macOS

Install R:

- Go to the CRAN download page for macOS.
- Download the latest `R-x.y.z.pkg` file.
- Run the downloaded `.pkg` installer file. Follow the prompts, and accept the default installation location.
- Go to the CRAN macOS tools page.
- Download and install the recommended **Clang** compiler tools.
- Download and install the recommended **GNU Fortran** compiler tools.
- You may also need to install the **Xcode Command Line Tools**. Open the Terminal application (found in `/Applications/Utilities/`) and run `xcode-select --install`. Install all of these *after* installing R but *before* attempting to install packages that might require compilation (especially those from GitHub).

Install RStudio:

- Go to the RStudio Desktop download page.
- Scroll down to the *Installers for Supported Platforms* section.
- Download the recommended version for macOS (`.dmg` file).
- Open the downloaded `.dmg` file and drag the RStudio application icon into your Applications folder.
- You can then launch RStudio from your Applications folder or Launchpad.

1.1.4 Cloud

An alternative to installing R and RStudio on your computer is a cloud-based service like **Posit Cloud** (formerly RStudio Cloud). This service provides access to an RStudio environment directly through your web browser.

1. **Access Posit Cloud:** Go to https://posit.cloud/.
2. **Sign Up/Log In:** Click the *Sign Up* or *Login* button. You can sign up for a free tier account using your Google or GitHub credentials or an email address.
3. **Create a Project:** Once logged in, you'll typically start by creating a *New Project*, which allocates a virtual machine instance running R and RStudio for you.
4. **Working in the Cloud:** The interface is essentially the same as the desktop version of RStudio. You can write code, install packages, and manage files within your project.

Posit Cloud can be a good option for getting started quickly, working on projects that don't require extensive computing resources, or collaborating with others.

However, a local installation is a better choice if you have adequate hardware, especially for more intensive tasks or long-term projects.

1.2 Installing Packages

One of the strengths of R is its packages. Being bare bones on its own, these packages extend R's capabilities to virtually any task imaginable. These packages generally come in three forms: those with an official stable release hosted on the *Comprehensive R Archive Network* (CRAN), those still in development, often found on platforms like GitHub, and those we write ourselves. We will look at each of these in turn.

1.2.1 CRAN

CRAN is the principal repository for stable, official R packages and is essentially a distributed network of servers hosting R distributions, contributed packages, documentation, and related materials. Before a package appears here, it has gone through a review process that helps ensure that it is stable, free of critical bugs, adheres to certain coding standards, includes essential documentation (such as README and NEWS files), and has a unique version number. Also, many of the packages here come with an accompanying article in journals like *The R Journal* or *The Journal of Statistical Software*, which provide the logic behind the package, practical explanations of the code, and illustrative examples.

The standard function for installing packages from CRAN is `install.packages()`. To install a single package, provide its name within double quotes:

```
install.packages("ggplot2")
```

You can install multiple packages simultaneously by providing a character vector of package names using `c` and separating them with commas. Note that we also include `dependencies = TRUE` here. Doing so makes R automatically check for any other packages (dependencies) we need for our package to work correctly. We highly recommend including this to avoid errors caused by missing dependencies:

```
install.packages(c("dplyr", "tidyr", "readr"), dependencies = TRUE)
```

By default, R installs packages in a standard library location on our system. Use `.libPaths()` to see where this is. The default location is usually fine, but you can specify another location using the `lib` argument if needed.

We should also keep our packages updated, as newer versions often provide us with new features, bug fixes, and performance enhancements. To check for and install updates for all our installed packages, run:

```
update.packages()
```

This will scan all installed packages, compare their versions with those available from CRAN, and present us with a list of packages we can update. In RStudio, we can also click the *Update* button in the *Packages* tab, which will open a window listing all packages with available updates. It is good practice to update packages regularly (e.g. monthly) to minimise the chance of encountering bugs already fixed in newer versions. To see which packages are outdated without starting the update process, type `old.packages()` in the console.

1.2.2 GitHub

If a package is still under active development or has not yet gone through the CRAN submission process, we can download it from platforms like GitHub. In addition, GitHub also hosts the latest versions of official packages before their official release, giving us access to the latest features and bug fixes, with the caveat that they may be less stable or more prone to problems than their CRAN counterparts. We cannot install these packages directly, but must use the `devtools` package to install them, which also allows us to install packages from other sources such as Bioconductor or GitLab:

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

As we are now installing packages from "source" (that is, they are not on CRAN), we need to have the necessary tools to build them before we can use them (compilers, libraries, etc.). What you need varies by operating system:

- **Windows**: From the CRAN RTools page, download the latest recommended version of RTools that matches your R version and run it, making sure that you add RTools to your system's PATH.
- **macOS**: You need the Xcode Command Line Tools. To install this, open the Terminal application (found in `/Applications/Utilities/`) and enter `xcode-select --install`. A software update window should appear. Follow the prompts to install the tools and accept the license agreement.
- **Linux**: Depending on our Linux flavour, you need several development libraries and compilers. To install them, run the following in your console:

  ```
  sudo apt-get update
  ```

  ```
  sudo apt-get upgrade
  ```

  ```
  sudo apt install build-essential libcurl4-gnus-dev libxml2-dev
  libssl-dev
  ```

  These commands update the package lists, upgrade existing software, and install essential build tools (`build-essential`) and libraries (`libcurl`, `libxml2`, `libssl`) needed for compiling R packages from source. The

package names can differ slightly between Linux distributions (like Fedora, CentOS, etc.).

Once you have `devtools` and the necessary build tools, you can use `install_github()` to install the packages. You can find the *github_username* and *repository_name* on the GitHub page that hosts the package:

```
library(devtools) # Loads the devtools package
install_github("github_username/repository_name", dependencies = TRUE)
```

1.2.3 Writing Packages

If no package does exactly what we need, we can write it ourselves. This can be a good idea if there are certain pieces of code that we need to run multiple times, or that we want to share with others. As the name suggests, packages are collections of *functions*, pieces of code that we define using the `function()` command:

```
my_function_name <- function(argument1, argument2, ...) {
  result <- # Commands to be executed using the arguments
  return(result)   # Returns the result
}
```

Functions are invaluable for making your code modular and repeatable, and you will use them often while using R. However, once you have a collection of related functions (and possibly data) that you use frequently or want to share with others, you should consider organising them into a *package*. This involves structuring your code and documentation in a particular directory layout and including specific metadata files (such as `DESCRIPTION` and `NAMESPACE`) that aim to make your code easily installable, loadable, and discoverable by others. For more information, see Hadley Wickham's book R Packages or the guide by Hilary Parker.

1.3 Required Packages

R has several packages for text analysis, including `tm`, `tidytext`, `RTextTools`, `corpus` and `koRpus`. While each of these packages offers its own special features that may be useful in certain contexts, we will primarily rely on the `quanteda` package here (Benoit et al., 2018). We do so because it is efficient, has a logical design, and communicates well with other packages. Although already hosted on CRAN, it is still under active development (see https://quanteda.io/) and has a well-maintained website with extensive documentation, tutorials and vignettes. To install it, run:

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

The main idea of `quanteda` is that the package itself contains the basic tools,

and other "helper" packages provide more specialised tasks. These are either already released on CRAN or are still under development:

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

library(devtools)

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

Besides `quanteda`, we need several other packages before we can start. Note that writing `devtools::` is another way of telling R to load this package and run the command that follows it (which can be useful if we have several packages with the same commands):

```
# Install from GitHub

devtools::install_github("mikegruz/kripp.boot", dependencies = TRUE)
devtools::install_github("matthewjdenny/preText", dependencies = TRUE)

# Install from CRAN
install.packages(c(
  "ca",            # Correspondence Analysis
  "caret",         # Machine Learning
  "combinat",      # Combinatorics
  "coop",          # Cosine Similarity
  "DescTools",     # Descriptive Statistics
  "ggdendro",      # Dendrograms
  "FactoMineR",    # Correspondence Analysis
  "factoextra",    # Visualisations for FactoMineR
  "Factoshiny",    # Shiny app for FactoMineR
  "Hmisc",         # Collection of useful functions
  "httr",          # Tools for working with URLs and HTTP
  "irr",           # For Krippendorff's alpha
  "jsonlite",      # Tools for working with JSON
  "lsa",           # Latent Semantic Analysis
  "manifestoR",    # Access Manifesto Project data
  "readr",         # Read .csv files
  "readtext",      # Read .txt files
  "reshape2",      #  Reshape Data
  "R.temis",       # Text Mining
  "rvest",         # Scrape Web Pages
  "seededlda",     # Semi-supervised Latent Dirichlet Allocation
  "stm",           # Structural Topic Models
```

```
  "tidyverse",     # Tools for data science
  "topicmodels",   # Topic Models
  "magick",        # Advanced Graphics
  "vader"          # Vader Sentiment Analysis
), dependencies = TRUE)
```

After successfully installing a package, we can find it in RStudio under the Packages tab. To use it, we can either select it there or run the `library()` command:

```
library(quanteda)
```

```
## Package version: 4.3.0
## Unicode version: 15.1
## ICU version: 74.2
```

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

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

1.4 Troubleshooting

It is perfectly normal to encounter errors when writing and running R code. First of all, **do not be discouraged!** Errors are just R's way of telling us that it could not understand or execute a command, and the error messages (often displayed in red text in the console) give us clues as to what went wrong. Sometimes the message is straightforward, telling us about a missing package or that we made a typo. Other times, it may be more cryptic. In this case, after carefully reading the error message (which, although technical, is essential as it is our primary source of information), we can ask ourselves the following questions:

1. **Are the necessary packages installed?** If the error says something like `Error: could not find function "some_function"`, this is a strong indication that the package containing some_function is not on our system. We can solve this by installing these packages using `install.packages()` or `devtools::install_github()`.

2. **Are the required packages loaded?** Even if we have installed all the required packages, we still need to load them with `library(packageName)` before R can use them. Forgetting to do this is one of the most common mistakes.

3. **Are the commands, function names and object names spelt correctly?** Remember that R is case-sensitive and `mydata` is different from `myData`. Typos in function names (`intall.packages` instead of `install.packages`) are also common.

4. **Is the data in the correct format?** Many R functions expect input data in a particular structure (e.g. a data frame, a vector, a matrix, or an object class such as a `quanteda` corpus or DFM). Use functions such as `class()`, `str()`, `summary()` or `dplyr::glimpse()` to inspect a data object and confirm that it matches what the function expects.

5. **Are the packages up-to-date?** Sometimes bugs are present in older versions of packages. Updating packages with `update.packages()` can fix these problems. We can also check if the bug occurred after updating R, as older packages may not be compatible with the latest version of R.

If none of this works, it is time to look for help online. Copying the exact error message and pasting it into a search engine is often the quickest way to find solutions, as many other R users have probably encountered and solved the same problem. If that does not work, there are several other options:

- **Stack Overflow:** A widely used question-and-answer site for programmers. Search for any problem or error message. If you need to ask a new question, try to make it as easy as possible for others to help by providing a clear description, the code you ran, the exact error message, and a reproducible example.
- **RStudio Community**: This is mainly for problems with RStudio itself, although there are also many threads for package issues.
- **Package documentation and vignettes**: The official documentation for packages (accessible via `?functionName` or `help(packageName)`) and longer tutorials called "vignettes" (accessible via `vignette(package = "packageName")`) often contain solutions or examples that clarify how to use functions correctly.
- **Package websites and GitHub pages**: Many packages have dedicated websites or detailed READMEs on GitHub that provide additional information and troubleshooting tips.

When asking for help online, the most helpful thing you can do is to provide a reproducible example. This example, or reprex, is a small, self-contained piece of code that produces the same error or unexpected behaviour you see when run by someone else. This allows others to understand your problem immediately, without needing access to your data or the whole project. The reprex package makes it easy to create these examples in three steps:

1. Install and load the `reprex` package
2. Write the minimal code that causes your problem.
3. Wrap your code inside the `reprex()` function and run it. The result (code, output, and error message) will be copied to your clipboard and formatted for pasting into forums like Stack Overflow.

```
# 1. Install and load the reprex package.
```

```r
install.packages("reprex")
library(reprex)

# 2. Write the code that demonstrates your problem. Example: You are trying to use a f

my_problem_code <- {
  df <- data.frame(a = 1:3, b = c("X", "Y", "Z"))
  dplyr::filter(df, a > 1)   # The issue occurs here as dplyr::filter requires library
}

# 3. Run the problematic code inside the reprex() function.

reprex({
  df <- data.frame(a = 1:3, b = c("X", "Y", "Z"))
  dplyr::filter(df, a > 1)
})
```

You can also include your `sessionInfo()` output at the end of your reprex or question. This command shows your version of R, your operating system, and the versions of all packages currently loaded, helping others to identify possible compatibility issues.

Finally, before you ask, it is always a good idea to see if the problem has already been discussed. Searching Stack Overflow or other forums before posting can save time and help keep the community tidy.

Chapter 2

Background

Before we look at the different applications of Quantitative Text Analysis (QTA), it is a good idea to discuss the basics. This will help us later to understand what we are doing, why, and what QTA can do. In this chapter, we will discuss the basic concepts of QTA, the standard workflow, the concepts of validity and reliability and why they are so important to us.

2.1 Concepts

First of all, let us define what Quantitative Text Analysis actually *is*. While there are various definitions, all agree that with QTA, we focus on *text as data*. Rather than taking the word as it is, we transform it (and other textual features) into numerical representations, which we then use as the input for various applications. Although this transformation means that we have to abstract the meaning of a word into numerical values, by doing so, we can easily examine (very) large collections of documents – better known as corpora – and can test pre-defined hypotheses, generalise findings across large datasets, synthesise information, and uncover broad trends that might not be apparent from close reading alone. This ability to work with large datasets is one of the reasons for the explosive growth and interest in QTA in recent years.

We can divide the different applications of QTA into four groups, each with its own focus and preferred techniques:

Describe: What are the characteristics of our corpus? In other words, how many documents do we have? What is their word count and vocabulary size? What are the most common terms, and how are they distributed? And what about the metadata - who are the authors of our texts, and when were they written? This descriptive information is crucial for familiarising ourselves with the data, identifying areas for further research and spotting anomalies early on.

Explore: What are the main ideas, opinions, perspectives and themes embedded in our texts? Methods such as topic modelling can reveal the underlying thematic structure of a corpus; dictionary-based approaches, machine learning classifiers or sentiment analysis can quantify emotional tone; and readability statistics can provide insights into the complexity and accessibility of the language used. All of these can improve our understanding of the content of the texts and reveal unexpected relationships between them.

Measure: Can we measure or create (latent) concepts with our texts? For example, we can use codes from dictionary-based approaches to measure the political left-right position of a party manifesto, or use the frequency of topics in the texts to reveal the agenda priorities of authors or organisations.

Predict: How can our data predict future events or test hypotheses about causal effects? Because we see our texts as data, we can use our texts as part of statistical models, such as a regression, to predict outcomes. For example, we could analyse how much a new party manifesto is likely to change the behaviour of voters or legislators, or whether debates on social media affected a previous election.

Note that in most cases, a single research study will combine them to answer its research question. They also build on each other. For example, a descriptive analysis usually precedes an exploratory one, and both are needed to measure or predict.

Often, *Quantitative* Text Analysis is contrasted with *Qualitative* Text Analysis, which focuses on the in-depth interpretation and nuanced understanding of texts, typically working with smaller numbers of documents. Compared to its quantitative counterpart, it emphasises deep immersion in the texts in order to understand their meaning, context and subtleties, allowing us to explore complex phenomena, develop new theoretical insights and provide rich and thick descriptions of our texts. Central to this approach are techniques such as coding to identify key concepts, thematic analysis to uncover recurring patterns of meaning, discourse analysis to study language in its social context, and narrative analysis to understand stories and accounts. Thus, while QTA often focuses on the 'what' aspect of a text, the qualitative approach focuses on answering the 'how' and 'why' questions related to a text.

However, rather than seeing them as opposing methodologies, we should see them as complementary. That is, we can use the two approaches to inform and enrich each other in many ways. For example, a deep qualitative understanding of the context and nuances of a text is invaluable in developing a robust coding scheme or specialised dictionaries that we can use in a quantitative analysis, while quantitative methods can efficiently scan large corpora to find overarching patterns, anomalies or specific subsets of texts that deserve more focused, in-depth qualitative study. As a result, combining the two approaches can lead to more convincing and interesting research and help to address the shortcomings of the other.

Table 2.1: Table 1 - Differences between Quantitative and Qualitative Text Analysis

Aspect	Quantitative Text Analysis	Qualitative Text Analysis
Goal	To measure, count, and identify patterns, frequencies, and statistical relationships. To test hypotheses.	To understand, interpret, and explore meanings, themes, and context. To generate hypotheses or theories.
Data	Numerical data derived from text (e.g., word counts, frequencies, coded categories represented numerically).	Textual data (e.g., interview transcripts, documents, open-ended survey responses, field notes).
Approach	Objective; aims for generalizability; deductive (tests pre-defined hypotheses).	Subjective; focuses on depth and richness of specific cases; inductive (develops understanding from the data).
Methods	Statistical analysis, content analysis (frequency-based), computational linguistics, automated sentiment analysis, topic modelling.	Thematic analysis, discourse analysis, narrative analysis, interpretative phenomenological analysis, grounded theory, close reading.
Analysis	Statistical tests, identifying correlations, creating visualisations of numerical patterns.	Coding (assigning labels to text segments), identifying themes, interpreting meanings, and building narratives.
Size	Large datasets to ensure statistical significance and generalizability.	Small, purposefully selected datasets to allow for in-depth analysis.
Focus	Breadth of information across many texts; identifying *what* and *how much/often*.	Depth of understanding within texts; exploring *why* and *how*.
Outcomes	Summaries, statistical significance, generalisable findings, identification of trends.	Rich descriptions, in-depth understanding of context, identification of themes, and development of concepts or theories.

For now, let us return to QTA. As we have already noted, the increased availability of (digital) texts and advances in computing power have led to a sharp increase in the popularity of QTA. As a result, there has also been a growing interest in its theoretical underpinnings, such as the work of Grimmer & Stewart

(2013) and Grimmer et al. (2022). They have sought to identify what QTA can (and cannot) do, and how best to do it. Overall, they stress the importance of six points:

1. **Theory**: Although QTA is data-driven, theory is inevitable. Thus, any QTA needs some form of theoretical guidance to tell us which texts to choose, how many features to extract, and what questions to ask. Indeed, the question of which method to use depends on our theoretical framework, and any findings we make will only make sense within a particular theoretical context.
2. **Models are models**: Language is complex. Models, no matter how elaborate and well-constructed, will always be simplifications. As such, we must always remember that we are not working with a text, but with a model of that text.
3. **Augmentation**: Computers - and thus QTA methods - are very good at organising, calculating and processing large amounts of text quickly, but not so good at understanding, interpreting and reasoning, which is where humans excel. So, a good QTA does not replace us, but complements us.
4. **No best method**: Our method will always depend on our research question, theoretical framework, and the type of data we are working with. This means that it is perfectly acceptable to choose a simpler method over a more complicated and complex one, as long as it is the right choice for our problem.
5. **Iteration**: A QTA analysis is rarely straightforward. Instead, the process is often iterative, where we explore the data, apply methods, evaluate results, and frequently return to earlier steps to try other ideas.
6. **Validation**: Because we are dealing with models, validation is essential to avoid misleading conclusions. We should validate as often and in as many ways as possible, including comparing automated results with human judgment, checking the face validity of results, and testing the predictive power of our measures.

2.2 Workflow

Now that we have seen what QTA is and what we can do with it, let us turn to how we actually go about it. Typically, whatever our objective (describe, explore, measure or predict), we will have a workflow consisting of 10 steps, from defining our research question to interpreting the results, although the exact implementation of each step will vary from project to project:

1. **Questioning:** We determine what we want to know. What questions do we want to answer? What are our theoretical assumptions?

2. **Selecting:** We define the scope of the corpus by setting clear criteria for which texts to include and which not to include. We do this by clearly defining our target audience and arguing why we include one text and not

another.

3. **Collecting:** We collect our selected texts to build our corpus. This may involve manual collection, web scraping or using existing text archives and databases. We also need to ensure the texts are in a usable format (e.g. .txt, .pdf, .doc) and have the correct versions.

4. **Cleaning:** We check our corpus for errors or inconsistencies that may have occurred during collection or conversion. Common problems include incorrect character encoding (e.g. displaying "™" instead of "Ü"). Raw text data is often messy and requires careful inspection and cleaning to avoid later errors. Remember that even text contained in databases usually involves some form of cleaning.

5. **Transforming:** We transform our corpus into a structured format suitable for analysis. The most common choice is the document-feature matrix (DFM), where rows represent documents, columns represent features (typically words, but can be n-grams or other units), and cell values indicate the frequency of each feature in each document.

6. **Pre-processing:** We refine our DFM by removing (or at least reducing) noise and irrelevant features. This can include removing stopwords, numbers, and punctuation and applying stemming (reducing words to their root form) or lemmatisation (reducing words to their dictionary form). We can also use weights to emphasise or de-emphasise certain terms.

7. **Describing:** We describe what our data look like. This may involve calculating descriptive statistics (e.g. word frequencies, document lengths) or creating visualisations (e.g. word clouds, distributions). This allows us to understand the structure of the data, identify patterns and check for any remaining problems.

8. **Modelling:** We select and apply the method or model of our choice based on our research question. This could be topic modelling, sentiment analysis, classification, or supervised learning techniques. We often have to iterate until we find the correct parameters to achieve optimal performance.

9. **Interpretating:** We analyse the results using tables, graphs and other visualisations and ask ourselves what our results *mean*. We then relate our findings to our theoretical framework and try to answer our research question.

10. **Validating:** We validate our results. Do the results make sense? Are they logical and expected, and if not, why? We need to validate not only here but at every stage, and we may need to revisit a previous step, e.g., refine corpus selection, improve data cleaning, adjust pre-processing steps, or try different models.

While the time spent on each step may vary, and some steps may occasionally be

combined or adapted, this workflow provides a general roadmap for quantitative text analysis. We will follow a similar approach in this book, going through these steps in the next chapters.

2.3 Validation

Of all the steps involved in the QTA workflow, validation is often the one that receives the least attention. However, given that each method is based on some form of machine learning, it is also the most important. In fact, Grimmer & Stewart (2013) advised to "validate, validate, validate!". The reason is simple. With the large amount of data and the fact that we are simply counting numbers, it is very easy to find evidence for something, especially when it is (very) large data. We should therefore be very sure of what we are doing and whether it makes sense. For this reason, it is a good idea to take a closer look at what validation actually means and how to go about it.

Validation involves two related concepts: reliability and validity. Here, *reliability* refers to our measure's *consistency* (or *stability*). If we have a measure with high reliability, this means that we should get the same results each time we run it - it is stable and not prone to random error or variation. *Validity,* on the other hand, refers to the accuracy or appropriateness of our measure. In other words, if our measure is valid, we measure what we want to measure. If you want to think about it in statistical terms, reliability is the proportion of non-random variance, while validity is the proportion of variance that the observed scores share with the true scores.

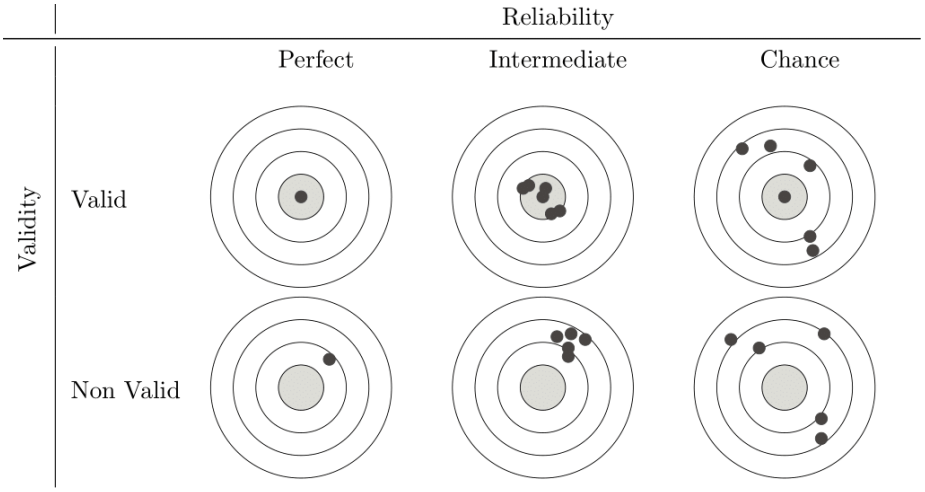


Figure 2.1: Validity and Reliability

Note that reliability and validity are not mutually exclusive. There is no point in having a highly valid measure that is not reliable, or a highly reliable measure

that is not valid. Figure 2.1, after Krippendorff (2019), shows this nicely by comparing our measure with a target, with the hits (the black dots) representing individual measures. What we are aiming for is at the top left: a measurement that hits the target perfectly every time. However, as our reliability decreases, more and more often we are not only hitting the target, but also hovering around it. As a result, high validity but low reliability means that sometimes we hit the target, but whether we do so is a matter of chance. At the same time, high reliability but low validity means that we hit the same spot every time, but always miss the target. So we never measure what we want to measure. So we want our measure to be both valid and reliable if it is to be of any use. Now, let's look at both concepts in a little more detail.

2.3.1 Validity

Of the two concepts, validity is the more difficult. To understand it a little better, we can divide it into three subtypes: content, criterion, and construct validity (Carmines & Zeller, 1979). Each of these focuses on a different aspect of validity and has different issues associated with it.

Content Validity	*Does our measure cover all the aspects or dimensions of the concept we are studying?* For example, if we design a coding scheme to measure 'political ideology', have we included all the different attitudes to existing economic, social and foreign policy issues? To see if this is the case, we often rely on expert judgement and our own theoretical understanding of the concept.
Criterion Validity	*How well does our measure correlate with other established measures of the same concept?* For example, we might test whether a sentiment score we get from social media texts agrees with the results of a public opinion poll on the same topic. Another way might be to use our measure to predict a future outcome and then check that the prediction is correct. For example, we could see if the amount of attention given to specific policies in legislative debates could help us predict future budget allocations for those policies.
Construct Validity	*How well does our measure operationalise our concept?* If it does, we would expect the measure to behave in the same way as the concept. For example, if we develop a measure of "economic uncertainty" from news articles, we would expect it to be negatively correlated with measures of consumer confidence.

So, how do we put this into practice? One aspect that generally makes validity more difficult than reliability is that there is very little we can measure. Instead, we have to argue and prove, using a variety of methods, that our measure and overall analysis are indeed valid.

The most common approach is to compare against a *gold standard*. For example, suppose we have access to a human-coded dataset for a subset of our data. In this case, we can compare the output of our (computational) method with these human judgments. Indeed, this is what we will do in Chapter 7, where we will use this to calculate metrics such as accuracy, precision, recall and F1 scores. Related to this is when we use our data to predict other (external) data. For example, we might build a model to explain the overall economy of a country, and then relate this to the country's actual GDP. Somewhat more complicated is when we focus on our analysis's actual *meaning*. This involves asking whether the results generated by our method are conceptually meaningful and make sense to people. In sentiment analysis, for example, we could manually review examples where our model gives strong positive or negative sentiment scores and see if they make sense. Preferably, we would have more than one person do this to avoid being too lenient on the model.

Ideally, we would use as many validation options as possible (a technique also known as triangulation). This way, with each validation, we build confidence and strengthen our argument that our approach is valid.

2.3.2 Reliability

Reliability, as we saw above, is whether our measurement measures the same thing each time we do it, or, more broadly, whether our analysis would lead us to the same conclusions each time. One thing that helps us here is that reliability is something we can measure (unlike validity). It often comes in three forms: stability, reproducibility and accuracy. The first, *Stability*, refers to how consistent a measurement is over time. For example, if we asked a coder to code the same text on different occasions, we would expect them to code it the same way (also known as within-coder consistency). The second, *Reproducibility* (also known as inter-coder reliability), extends this to multiple coders. This means that independent coders, given the same coding instructions, should produce the same code for the same text. Finally, the third, *Accuracy* (which we also use for validity), compares our coders' codes to a known standard or "true" value. The better the comparison, the more reliable our measure.

The first type - stability - is easy to measure: just repeat the analysis and compare the results. The second - reproducibility - is more complicated. This is because there are several things we want to be able to take into account, such as the ability to account for the categories our coders actually use, a standardised scale for interpretation, appropriateness to the level of measurement of our data (e.g. nominal, ordinal, interval, ratio), correction for chance agreement, and the ability to handle missing data. There are several measures of reproducibility, each with its own strengths and limitations. The simplest and most straightforward is *Percentage Agreement*, where we divide the number of codes the coders agree on by the total number of codes coded. However, this does not consider agreement that could occur purely by chance, therefore overestimates reliability.

Another way of doing this is to use Pearson's correlation coefficient (r), as we would assume that we are correlating the codes of one or more coders. However, Pearson's r measures linear association, not agreement. Thus, two coders can be in perfect disagreement and still show a strong positive or negative correlation if their disagreements follow a consistent linear pattern.

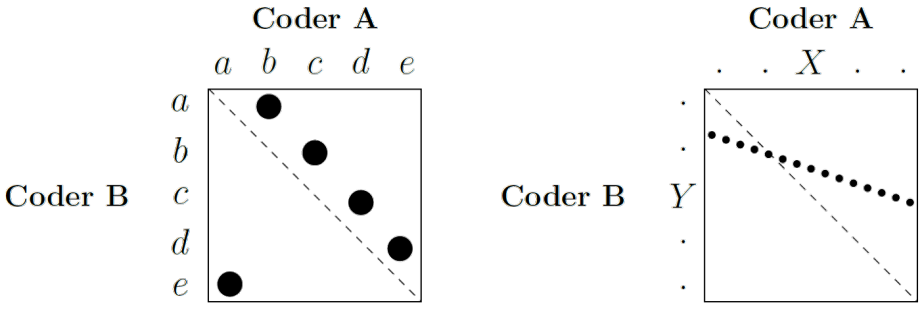


Figure 2.2: Perfect agreement between two coders

Consider Figure 2.2, which is an example adapted from Krippendorff (2019) to illustrate this. Suppose we have two coders, A and B, each assigning sentences to five categories, labelled 'a' to 'e'. If, for example, whenever Coder A assigns 'a', Coder B assigns 'e'; and whenever Coder A assigns 'b', Coder B assigns 'a', and so on, they are in perfect disagreement (the example on the right). However, if we calculated Pearson's r, we might still find a high correlation. This is because Pearson's r only looks at the distances between the values, regardless of their location. For there to be a correlation, it is only necessary for increases or decreases in the category values assigned by one coder to be mirrored by similar directional changes in the other's assignments, a condition that can be met even when there is complete disagreement about the actual categories assigned. For this reason, we generally do not recommend it. Instead, we have the following options:

Cohen's κ is useful for assessing agreement between *two coders* on *nominal* (categorical) data. It improves on percent agreement because it corrects for chance agreement, based on each coder's individual marginal distributions of codes.

Scott's π is similar to Cohen's κ, but we use it when we assume that the two coders are drawing from the same underlying distribution of codes. Consequently, it calculates the chance agreement based on the pooled marginal distribution of codes. Like Cohen's κ, we use it with two coders and nominal data.

Fleiss' κ is an extension of Scott's π . We use it when assessing the agreement between *multiple coders* (more than two) on *nominal* data. A key requirement for Fleiss' κ is that each unit (e.g. document, sentence) must be coded by the same number of coders, although it does not necessarily have to be the exact

same set of coders for each unit.

Since each of these three measures has its drawbacks, we will use **Krippendorff's** α here. It improves on the other measures by handling any number of coders, allowing for missing data, and applying to any level of measurement - nominal, ordinal, interval, and ratio. In addition, it calculates the random agreement based on the observed data, rather than assuming any distribution.

We can calculate Krippendorff's α using the formula:

$$\alpha = 1 - \frac{D_o}{D_e}$$

Where D_o is the disagreement we observe between the coders, determined by the distance function we choose to be appropriate for the level of measurement of our data. D_e represents the disagreement we would expect by chance, calculated from the distribution of codes assigned by our coders. Thus, if we obtain a α value of 1.0, this indicates perfect agreement; 0.0 indicates agreement at the level of chance alone; and a value below 0.0 indicates systematic disagreement between our coders. For interpretation, we follow Krippendorff's suggestion that a $\alpha \geq 0.800$ indicates good reliability, while values between $0.667 \leq \alpha < 0.800$ may allow us to draw tentative conclusions. We usually consider alpha values below 0.667 to indicate poor reliability.

However, even α has its limitations, the most problematic being when coders agree on only a few categories and use those categories very often. This inflates the value of α, making it higher than it should be.

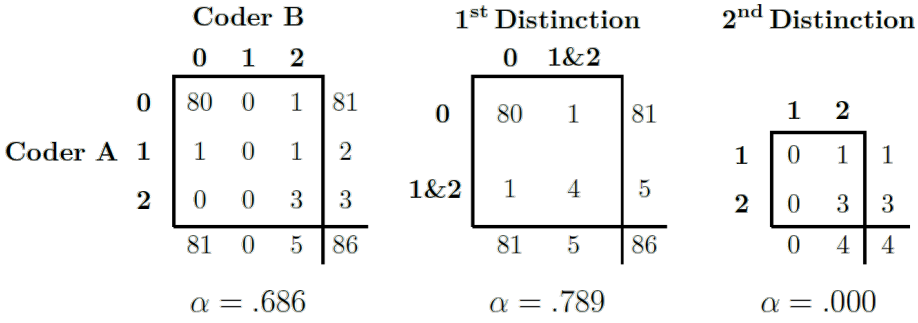


Figure 2.3: Inflation caused by use of a limited number of categories

Figure 2.3 (based on Krippendorff (2019)) illustrates this point. Imagine a coding task with three categories (0, 1 and 2). Category 0 indicates that the coders could not assign a more specific code, while categories 1 and 2 represent meaningful codes. If, out of a large number of cases (e.g. 86), both coders assign category 0 to the majority (e.g. 80 cases), this leaves very few cases for us to observe agreement or disagreement on the meaningful categories 1 and 2. If

we then calculate α over all three categories, we get a moderate value (0.686). However, if we then collapse categories 1 and 2 into a single category 'meaningful code', distinguishing only between 'meaningful code' and 'no meaningful code' (category 0), the agreement on this broader distinction suddenly becomes very high, leading to a higher α (0.789). On the other hand, if we remove the dominant 0 category and calculate α only on categories 1 and 2 (for the few cases where they were used), the resulting α could be very low (almost 0). This could happen if a coder did not use one of these categories at all, even if they agreed on the other meaningful category in the remaining cases. This shows how our choice of categories to include in the calculation, and their observed distribution, can significantly influence the resulting α value.

Finally, α depends on our chosen metric (e.g. nominal, ordinal, interval, ratio). If we use an inappropriate metric, such as a nominal metric for data that is actually ordinal, we may ignore valuable information about the ordered nature of the categories, leading us to misunderstand the actual level of agreement achieved by our coders.

To calculate α in R, we use the `irr` package, which provides the `kripp.alpha()` function. To see how this works, we simulate a case where 12 coders code 10 sentences into 3 categories:

```
library(irr)  # Load the library
set.seed(24)  # Setting a seed makes our example reproducible

# We create a matrix with 10 coders (rows) coding 12 sentences (columns) into 3
# categories (1, 2, or 3)

reliability_matrix <- matrix(sample(1:3, 10 * 12, replace = TRUE), nrow = 10, ncol = 12)

# Now, we calculate Krippendorff's alpha, specifying the data and method (level
# of measurement). For this example, we assume nominal data.

k_alpha_nominal <- kripp.alpha(reliability_matrix, method = "nominal")
print(k_alpha_nominal)
```

```
##  Krippendorff's alpha
##
##  Subjects = 12
##    Raters = 10
##     alpha = 0.0106
```

When we run `kripp.alpha`, the output typically includes the calculated α value, the number of units (which it refers to as subjects), the number of coders (raters), and the level of measurement we specified for the calculation (e.g. "nominal"). We then compare this resulting α value with established thresholds (e.g. 0.67 or 0.80, as Krippendorff suggests) to assess our coding reliability. For a more nuanced understanding of the stability of our estimate, we can obtain bootstrapped

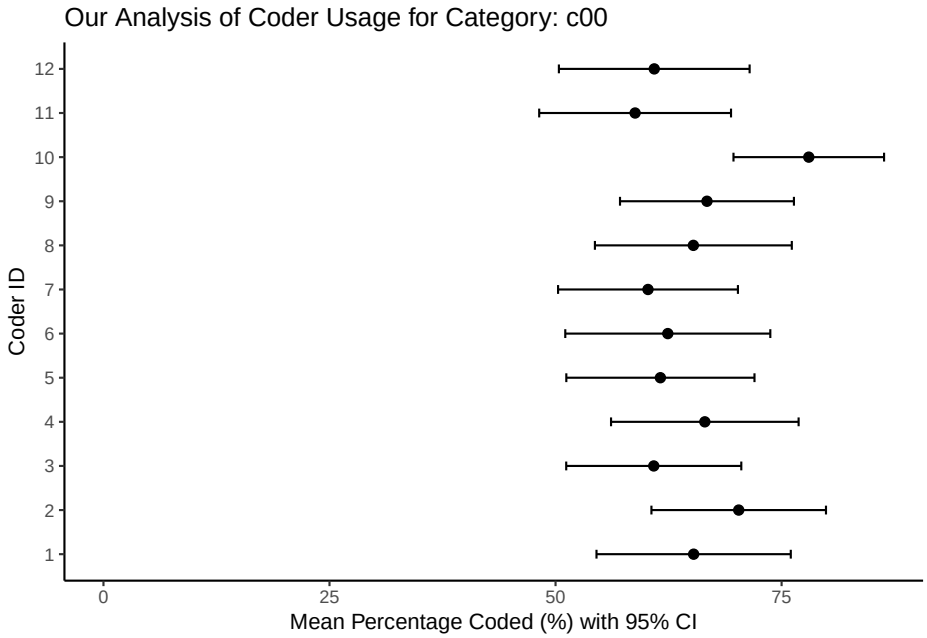
confidence intervals for α using packages such as `kripp.boot` (see here).

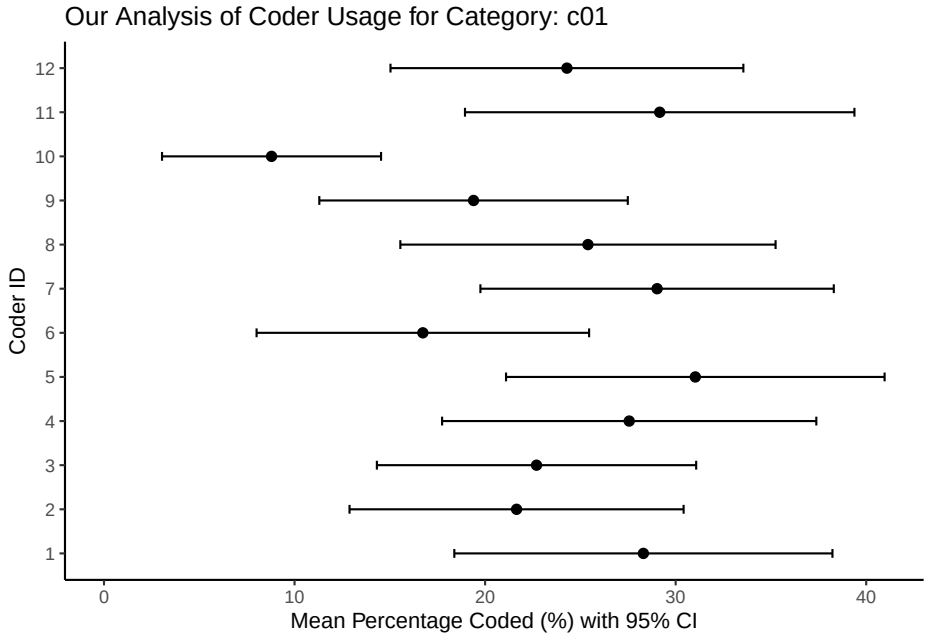
In addition, we can also visualise our reliability. One way of doing this, adapted from Benoit et al. (2009) and Lowe & Benoit (2011), is to use bootstrapping to estimate and visualise the uncertainty around each coder's distribution of codes across the different categories. To do this, we first obtain the number of times each coder used each specific category. Then, for each coder, we use their observed coding patterns (i.e., the proportion of times they used each category) to repeatedly resample their codings, typically using a multinomial distribution. From these numerous bootstrapped samples, we compute summary statistics such as the mean percentage and standard error for each category for each coder. Finally, we plot these mean percentages along with their confidence intervals. This allows us to visually represent the consistency of each coder and identify any significant variation in their application of the coding scheme. As before, let's simulate the coding output of 12 coders across 3 categories and see how this works:

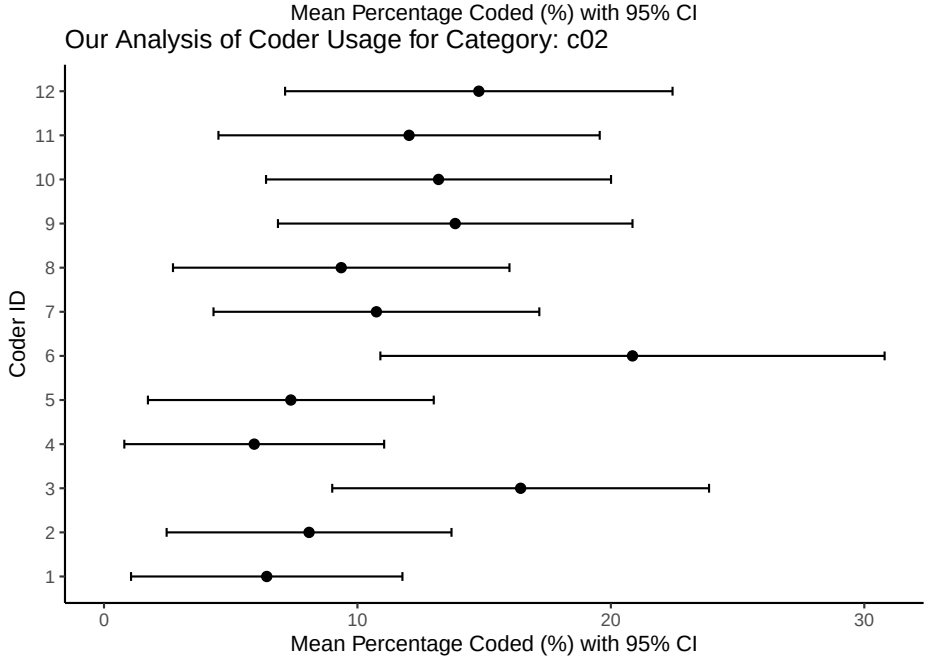
```
library(dplyr)
library(tidyr)
library(ggplot2)

set.seed(48)

# Create placeholder data to simulate coder output. This tibble will have coder IDs an

data_uncertainty <- tibble(
  coderid = 1:12,
  c00 = rpois(12, 50), # Simulating counts for category 0
  c01 = rpois(12, 20), # Simulating counts for category 1
  c02 = rpois(12, 10)  # Simulating counts for category 2
) %>%
  mutate(n = c00 + c01 + c02) # Total codes per coder

category_cols_id <- c("c00", "c01", "c02")

# Now, we perform the bootstrap
n_coders <- nrow(data_uncertainty)
n_repl <- 2000 # We set the number of bootstraps
n_categories <- length(category_cols_id)

# We prepare an array to store our bootstrap results: coder x category x replicate
bootstrap_results_array <- array(
  NA,
  dim = c(n_coders, n_categories, n_repl),
  dimnames = list(data_uncertainty$coderid, category_cols_id, 1:n_repl)
)
```

```r
# We loop through each coder to resample their codings.
for (coder_idx in 1:n_coders) {
  observed_counts <- as.numeric(data_uncertainty[coder_idx, category_cols_id])
  total_codes_n <- data_uncertainty$n[coder_idx]

  observed_probs <- observed_counts / total_codes_n
  # We ensure probabilities sum to 1
  if (abs(sum(observed_probs) - 1) > 1e-6) {
    observed_probs <- observed_probs / sum(observed_probs)
  }

  # We perform multinomial resampling
  resampled_counts_matrix <- rmultinom(n = n_repl, size = total_codes_n, prob = observed_probs)
  bootstrap_results_array[coder_idx, , ] <- resampled_counts_matrix

}

# We convert counts to percentages
bootstrap_percentages_array <- sweep(
  bootstrap_results_array,
  MARGIN = c(1, 3),
  data_uncertainty$n,
  FUN = "/"
) * 100
# Handle potential NaN if a coder had 0 codes for 'n'
bootstrap_percentages_array[is.nan(bootstrap_percentages_array)] <- 0

# Calculate summary statistics
mean_perc <- apply(bootstrap_percentages_array, c(1, 2), mean, na.rm = TRUE)
# Use the SD of the bootstrapped means as an estimate of the standard error
sd_perc <- apply(bootstrap_percentages_array, c(1, 2), sd, na.rm = TRUE)

mean_perc_df <- as.data.frame.table(mean_perc, responseName = "mean_p") %>%
  rename(coderid = Var1, category = Var2)
sd_perc_df <- as.data.frame.table(sd_perc, responseName = "se_p") %>%
  rename(coderid = Var1, category = Var2)

vis_data <- full_join(mean_perc_df, sd_perc_df, by = c("coderid", "category")) %>%
  mutate(
    lower_ci = mean_p - 1.96 * se_p, # 95% CI lower bound
    upper_ci = mean_p + 1.96 * se_p, # 95% CI upper bound
    lower_ci = pmax(0, lower_ci), # Ensure CI doesn't go below 0%
    upper_ci = pmin(100, upper_ci), # Ensure CI doesn't exceed 100%
    coderid = factor(coderid)        # Treat coderid as a factor for plotting
  )
```

```r
# Finally, we plot the three categories:

categories_to_plot <- category_cols_id[1:min(3, length(category_cols_id))]
plots_list <- list() # To store plots if we generate multiple

for (cat_to_plot in categories_to_plot) {
  plot_data_subset <- filter(vis_data, category == cat_to_plot)

  p <- ggplot(plot_data_subset, aes(x = mean_p, y = coderid)) +
    geom_point(size = 2) +
    geom_errorbarh(aes(xmin = lower_ci, xmax = upper_ci),
                   height = 0.2,
                   na.rm = TRUE) +
    scale_x_continuous(name = "Mean Percentage Coded (%) with 95% CI", limits = c(0, NA
    scale_y_discrete(name = "Coder ID") +
    ggtitle(paste("Our Analysis of Coder Usage for Category:", cat_to_plot)) +
    theme_classic()

  plots_list[[cat_to_plot]] <- p
  print(p) # Display the plot
}
```







In these plots, the horizontal bars represent the 95% confidence intervals we derived from the bootstrap resampling process. When we see shorter bars around a coder's mean percentage, this indicates high consistency (low uncertainty) in that coder's use of that category relative to their overall coding activity. On the other hand, longer bars indicate greater uncertainty, which could be due to the

coder's less frequent or less consistent use of that particular category. When comparing between coders, overlapping confidence intervals indicate that the coders used the category at statistically similar rates. However, non-overlapping intervals may indicate systematic differences in how a particular coder interpreted or applied a category compared to their peers. Graphs such as these can help us identify specific categories or coders that contribute most to disagreement, and we can use them to improve our coder training or refine our coding scheme.

Chapter 3

Text in R

No analysis is possible unless we have some data to work with. In this chapter, we will look at seven different ways of getting textual data into R: a) using .txt files, b) using .pdf files, c) using .csv files, d) using an API, e) using web scraping, f) using structured files such as JSON and XML, and g) importing from a database. However, before we look at these methods, let us first look at how R understands text and how we can work with it.

3.1 Basics

R treats text as a *string* or *character vector*, making it one of R's basic data structures, the others being logical, integer, real (numeric), complex, and raw. A character vector can contain a single string (such as a single word or phrase), while more complex character vectors can contain multiple strings, which could represent a collection of sentences, paragraphs, or even entire documents. Because character vectors are vectors, we can perform many of the same operations on them as we can on other vector types, such as calculations or checking their properties. For example, the `nchar()` function returns the number of characters *within* each string element, while `length()` returns the number of *elements* (or individual strings) contained in the vector.

Let's start by defining a character vector containing a single string and examining its properties using these two functions. Note that in R, we must enclose our strings in either double quotes (") or single quotes (‘ ’):

```
vector1 <- "This is the first of our character vectors"
nchar(vector1)  # Number of characters in the string
length(vector1)  # Number of elements (strings) in the vector
```

As you can see from the output of the above code, the `nchar()` function tells us that the string has 42 characters. However, `length()` tells us the vector

contains only 1 element. This is because `vector1` contains only a single string, even though that string is quite long. To illustrate a vector with multiple strings, we use the `c()` function (short for "combine"). When used with strings, `c()` combines multiple individual strings into a single character vector. Let's create a vector with three different strings and see how `length()` and `nchar()` behave:

```
vector2 <- c("This is one example", "This is another", "And so we can continue.")
length(vector2)  # Number of elements in vector
nchar(vector2)  # Returns characters for each element
sum(nchar(vector2))  # Total number of characters in all elements
```

When we run this code, `length(vector2)` will return 3 because the vector contains three separate strings. As a result, `nchar(vector2)` now has multiple elements (3), and `nchar()` returns a vector of results, one for each string. To get the total number of characters in all three strings, we can wrap (or nest) `nchar(vector2)` inside the `sum()` function. Note that R typically evaluates commands from the inside out. So, it will first calculate the number of characters for each string in `vector2` (producing an intermediate numeric vector), after which the `sum()` function will calculate the sum of this intermediate vector.

The next step is to modify our vectors. For example, we can extract specific parts of a string and create substrings using the `substr(text, start_position, end_position)` function. To do so, we specify the starting and ending character positions. Note that the positions are counted from the beginning of the string, starting at 1 (and thus not 0):

```
substr(vector1, 1, 5)  # Extracts characters from position 1 to 5
substr(vector1, 7, 11)  # Extracts characters from position 7 to 11
```

Another thing we can do is combine multiple strings into a single one. The first way to do this is to use the `paste()` function and concatenate multiple strings into a single one. By default, `paste()` concatenates strings with a space in between, but we can change this using the `sep` argument. Another approach is to use a vector of multiple strings. In this case, we still use `paste()`, but now with the `collapse` argument:

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

paste(vector2, collapse = "")
```

We can also change the text itself, for example, changing its case (lowercase or uppercase) using `tolower()` (which converts all characters to lowercase) and `toupper()` (which converts them to uppercase):

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

Another (and powerful) feature of R is its ability to find specific patterns within strings. These functions are especially powerful when combined with regular expressions (see for more on that here):

- `grep(pattern, x)`: This function searches for the specified `pattern` within each string element of the character vector `x`. It returns a vector of the *indices* (positions) of the elements in `x` that contain a match for the pattern.
- `grepl(pattern, x)`: Similar to `grep`, but instead of returning indices, it returns a *logical vector* of the same length as `x`. Each element in the resulting vector is `TRUE` if the corresponding string in `x` matches the pattern and `FALSE` otherwise. This is useful for filtering or sub setting.
- `sub(pattern, replacement, x)`: This function finds the specified `pattern` in each string element of `x` and replaces it with the `replacement` string. Importantly, `sub()` only replaces the *first* occurrence of the pattern found within each string.
- `gsub(pattern, replacement, x)`: This function is identical to `sub()`, but with one key difference: it replaces *all* occurrences of the pattern found within each string element, not just the first one.

Let's see these pattern-matching and replacement functions in action:

```
text_vector <- c("This is a test sentence.", "Another test here.", "No match in this one.")

grep("test", text_vector)  # Find elements containing the exact word 'test'
grepl("test", text_vector)  # Check which elements contain the word 'test'
sub("test", "sample", text_vector)  # Replace the first instance of 'test' with 'sample' in each
gsub(" ", "_", text_vector)  # Replace all spaces ' ' with underscores '_'
```

The opposite of pasting strings together is splitting them apart. The `strsplit(x, split)` function is designed to break down the elements of a character vector `x` into smaller pieces based on a specified `split` pattern (often a single character like a space, comma, or dash). Because each string in the input vector `x` might be split into a different number of resulting pieces, `strsplit()` returns a *list*. Each element of this list corresponds to an original string from `x`, and within each list element is a character vector containing the substrings that resulted from the split.

```
sentence <- "This sentence will be split into words"
strsplit(sentence, " ")  # Splits the single string by spaces, returns a list with one element (c

dates <- c("2023-01-15", "2024-11-01")
strsplit(dates, "-")  # Splits each date string by the dash, returns a list with two elements (ve
```

While basic functions like `print()` and `cat()` are sufficient for simple output, `sprintf()` provides much finer control over how numbers, strings, and other data types are formatted within a string, similar to the `printf` function found in C. You construct a format string containing placeholders (like `% d' for`

integers,%sfor strings,%.2ffor floating-point numbers with two decimal places), andsprintf()‘ replaces these placeholders with the values of subsequent arguments, respecting the specified formatting rules. This is particularly useful for creating consistent output or messages.

```
my_var <- "Hello"
print(my_var)  # Prints the variable's value, often with quotes for strings
cat(my_var, "world!\n")  # Concatenates and prints, useful for console output

value <- 42.567
sprintf("The value is %.2f", value)  # Formats' value' to 2 decimal places within the
sprintf("Integer: %d, String: %s", 100, "example")  # Inserts an integer and a string
```

While base R provides these essential tools for working with text, specialised packages such as `quanteda`, `tm`, `stringr`, or `tidytext` offer more comprehensive, efficient, and often more user-friendly functions for complex text processing and analysis tasks. These packages typically build upon the fundamental vector concepts and functions available in base R, providing extended capabilities that include more powerful regular expression handling, tokenisation, stemming, stop-word removal, and advanced text manipulation tools.

3.2 Import .txt

Plain text (.txt) files are a standard, simple format for storing text due to their lack of formatting, small size, and cross-platform compatibility. The `readtext` package provides a convenient way to read text files. It automatically creates a data frame with text content and associated metadata, such as filenames. First, ensure the `readtext` package is installed (`install.packages("readtext")`) and loaded. Then, specify the directory containing your .txt files. It is good practice to use relative paths or R projects to manage file locations.

```
library(readtext)

txt_directory <- "folder/txt"  # Address of folder with the TXT files

# The pattern '*' reads all files Encoding specifies the encoding the TXT
# documents are in

data_texts <- readtext(paste0(txt_directory, "/*.txt"), docvarsfrom = "filenames",
    encoding = "UTF-8")
```

In the last line, we set `docvarsfrom = "filenames"`. This creates a variable that stores the filenames alongside the text, allowing us to identify them later easily. In addition to `readtext`, base R functions like `readLines()` or `scan()` can also read .txt files, often line by line or into a single character vector, which may require further processing to organise by document.

3.3 Import .pdf

Importing PDF files is a bit more challenging than importing TXT files, as they often contain not only text but also complex formatting, images, and tables. To import them into R, we again use the `readtext` package (though, in this case, it will use `pdftools` in the background). Note that this only works with readable PDFs (where text can be selected/copied), not image-based scans. In case you have an image-based scan (basically a picture stored as a PDF), you first need to use Optical Character Recognition (OCR) to retrieve the text from it (there are various ways to do this, with `tesseract` being the most popular – see for more on that here).

```
library(readtext)

pdf_directory <- "folder/pdf"  # Address of folder with the PDF files
data_pdf_texts <- readtext(paste0(pdf_files, "/*.pdf"), docvarsfrom = "filenames")
```

Also, remember that PDF text extraction is imperfect. Formatting, tables, headers/footers, and multi-column layouts can lead to jumbled text. Manual inspection and cleaning are often required.

3.4 Import .csv

Sometimes, text data comes pre-processed as a document-term matrix (DTM) or term-frequency matrix stored in a CSV file. A DTM typically has documents as rows, terms (or words) as columns, and cell values representing the word counts. There are two main ways we can import CSV files: using R's inbuilt `read.csv()` or the `read_csv` function from the `readr` package:

```
data_dtm <- read.csv("your_dtm_file.csv")  # In case the first row is NOT the column names
data_dtm <- read.csv("your_dtm_file.csv", header = TRUE)
data_dtm <- readr::read_csv("your_dtm_file.csv", col_names = FALSE)  # In case the first row are
data_dtm <- readr::read_csv("your_dtm_file.csv")
```

Remember that importing a pre-computed matrix means you inherit the pre-processing choices made when it was created. Also, take into account that in some cases, the CSV is not delimited by a comma but by a semicolon (;) or tab. In that case, we have to import it as a delimited object:

```
data_dtm <- read_delim(NULL, delim = ";", escape_double = FALSE)
data_dtm <- read_delim(NULL, delim = "\t", escape_double = FALSE)
```

3.5 Import from an API

Application Programming Interfaces (APIs) provide structured ways to request and receive data directly from web services (e.g., social media platforms, news

organisations, databases). When available, using an API is generally more reliable and efficient than web scraping. There are some considerations to keep in mind:

- **Registration/Authentication**: Most APIs require registration to obtain an API key or token for authentication.
- **Rate Limits**: APIs usually limit the requests allowed within a specific period.
- **Terms of Service**: Always review the API's terms of service regarding data usage and restrictions.
- **API Changes & Restrictions**: APIs can change. Notably, access to platforms like Twitter/X and Facebook has become significantly restricted and often requires payment or enhanced verification. For instance, the `Rfacebook` package is no longer actively maintained. Always check the current status and documentation.
- **R Packages**: Specific R packages often exist to simplify interaction with popular APIs (e.g., `rtweet` for Twitter/X, `RedditExtractoR` for Reddit, `WikipediR` for Wikipedia, `manifestoR` for the Manifesto Project corpus). If no dedicated package exists, you can use general HTTP packages like `httr` or `httr2` combined with `jsonlite` to handle requests and responses.

To demonstrate how this works, let us have a look at the New York Times Movie Reviews API (requires registering for an API key at https://developer.nytimes.com/):

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

nyt_api_key <- "[YOUR_API_KEY_HERE]"  # Replace '[YOUR_API_KEY_HERE]' with your actual

# Construct the API request URL
base_url <- "[https://api.nytimes.com/svc/movies/v2/reviews/search.json](https://api.ny
query_params <- list(query = "love", `opening-date` = "2000-01-01:2020-01-01", `api-key

response <- GET(base_url, query = query_params)  # Make the API request using httr::GE

# Parse the JSON content
content_json <- content(response, as = "text", encoding = "UTF-8")
reviews_list <- fromJSON(content_json, flatten = TRUE)
reviews_df <- as_tibble(reviews_list$results)  # Convert the relevant part of the list
```

This example retrieves movie reviews published between 2000 and 2020 containing the word "love". The response is in JSON format, which `jsonlite::fromJSON()` converts into an R list, which is subsequently transformed into a `tibble` (a type of data frame).

3.6 Import using Web Scraping

When an API is unavailable, web scraping—extracting data directly from the HTML structure of web pages—can be an alternative. Again, there are a few things to keep in mind:

- **Legality/Ethics**: Always check the website's `robots.txt` file (e.g., `www.example.com/robots.txt`) and Terms of Service before scraping. Many sites prohibit or restrict scraping. Respect website resources; avoid overly aggressive scraping that could overload servers.
- **Website Structure**: Scraping relies on the stability of a website's HTML structure. If the site changes, your scraper might break.
- **Static or Dynamic Content**: Simple websites with content loaded directly in the initial HTML are easier to scrape (using packages like `rvest`). Websites that load content dynamically using JavaScript after the initial page load often require browser automation tools, such as `RSelenium`.
- **Complexity**: Scraping can be complex and requires knowledge of HTML and CSS selectors (or XPath).

To see how scraping works, let us scrape the page on the Cold War from Wikipedia using `rvest`:

```
# Load necessary libraries
library(rvest)
library(dplyr)
library(stringr)
library(tibble)

# Define the URL of the Wikipedia page
url <- "https://en.wikipedia.org/wiki/Cold_War"

# Read the HTML content from the page
html_content <- read_html(url)
```

Now, we need to identify the HTML elements containing the desired text. Using browser developer tools (often opened with F12 or Cmd+Shift+C) helps inspect the page structure. The main content paragraphs for Wikipedia articles are typically within <p>tags inside a main content `div`:

```
# Extract paragraph text from the content section
paragraphs <- html_content %>%
    html_nodes("#mw-content-text .mw-parser-output p") %>%
    html_text2()  # Extract text, attempting to preserve formatting like line breaks

coldwar_text_df <- tibble(paragraph = paragraphs)  # Convert to a tibble/data frame
coldwar_text_df <- coldwar_text_df %>%
    filter(nchar(trimws(paragraph)) > 0)  # Remove empty or whitespace-only paragraphs
```

```r
# Extract paragraph text from the content section
paragraphs <- html_content %>%
    html_nodes("#mw-content-text .mw-parser-output p") %>%
    html_text2()  # Extracts text while preserving formatting

# Convert to a tibble and remove empty/whitespace-only paragraphs
coldwar_text_df <- tibble(paragraph = paragraphs) %>%
    filter(nchar(trimws(paragraph)) > 0)

# Display the first few rows
print(head(coldwar_text_df))
```

For more complex scraping involving logins, button clicks, or dynamically loaded content, explore the `RSelenium` package, which programmatically controls a web browser. For more on web scraping, see Wickham et al. (2023), with the book being available here.

3.7 Import JSON and XML

JSON (JavaScript Object Notation) and XML (eXtensible Markup Language) are standard hierarchical data formats often obtained from APIs or other data sources. For JSON, we use the `jsonlite` package. Here, `fromJSON()` reads JSON files or text into R lists or data frames. Note that the path does not need to refer to a place on your computer but can also be an address to an online API:

```r
library(jsonlite)
my_data_df <- fromJSON("path/to/your/data.json", flatten = TRUE)  # Here, ``flatten =
```

For XML, we use the `xml2` package. It provides functions like `read_xml()` or `read_html()` to parse files/URLs, and `xml_find_all()` (with XPath expressions), `xml_text()`, `xml_attr()` etc., to navigate and extract data:

```r
library(xml2)
xml_doc <- read_xml("path/to/your/data.xml")
titles <- xml_find_all(xml_doc, ".//title") %>%
    xml_text()  # Find specific nodes using XPath and extract text (here <title>)
```

3.8 Import from Databases

If your text data is somewhere in a relational database (like PostgreSQL, MySQL, SQLite, etc.), you can connect directly from R using the `DBI` (Database Interface) package along with a specific backend package for your database type, such as `RPostgres`, `RMariaDB`, `RSQLite`, `odbc`. Here, let us try to import a PostgreSQL database file:

```r
library(DBI)
library(RSQLite)  # Load the specific backend

connection <- dbConnect(RPostgres::Postgres(), dbname = "your_db", host = "your_host",
    port = 5432, user = "your_user", password = "your_password")

print(dbListTables(connection))  # List available tables

query <- "SELECT doc_id, text_content FROM documents WHERE year = 2023;"  # Query the database us
results_df <- dbGetQuery(connection, query)

dbDisconnect(connection)
```

Chapter 4

Describe

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 expect to find in them. This chapter focuses on techniques for exploring and summarising text data, including keywords-in-context, visualisations, and text statistics. Before diving into these techniques, we will briefly discuss the concept of the *corpus*, which is central to working with text data in `quanteda` and the *DFM* (data-frequency matrix), which we derive from it. Throughout this chapter, we will use the example of the Manifesto Project corpus, specifically the UK manifestos, to illustrate these concepts and techniques. This data is part of the `quanteda.corpora` package as `data_corpus_ukmanifestos`.

4.1 Corpus and DFM

In `quanteda`, the primary object for storing and managing text data is the `corpus`. A `corpus` object holds your documents and any associated metadata, known as *document-level variables* or *docvars*. The key characteristic of a `corpus` object is that it remains immutable during your analysis. Instead of modifying the original corpus, you create derivative objects (like tokens or document-feature matrices) for analysis. This approach ensures reproducibility and allows you to easily return to your original data for different analyses or pre-processing steps.

Creating a corpus in `quanteda` is straightforward. As seen in the 3 chapter, one standard method is to use the `readtext` package, which reads various file formats and creates a data frame with document IDs and text. This data frame can be directly converted into a `corpus` object using the `corpus()` function.

Alternatively, you can create a corpus from a simple character vector, where each element represents a document. If the vector elements are named, these

names will be used as document identifiers (`doc_id`); otherwise, `quanteda` will generate default IDs.

Incorporating document-level variables (docvars) is highly recommended. These variables store crucial metadata about each document, such as the author, publication date, source, or other relevant categorical or numerical information. Docvars are essential for grouping, filtering, and conducting analyses that relate textual features to external characteristics of the documents. When you create a corpus from a data frame, `quanteda` automatically attempts to include other columns as docvars. You can add or modify docvars later using the `docvars()` function.

```
library(quanteda)  # Core text analysis functions
library(quanteda.corpora)  # Access to built-in text corpora
library(quanteda.textstats)  # Text statistics
library(quanteda.textplots)  # Visualizing text statistics
library(ggplot2)  # Plots
library(reshape2)  # For melting data frames

data(data_corpus_ukmanifestos)  # Load the UK political manifestos corpus

selected_parties <- c("Con", "Lab", "LD", "UKIP", "SNP", "PCy", "SF")  # Filter the co
data_corpus_ukmanifestos <- corpus_subset(data_corpus_ukmanifestos, Year > 1996 &
    Party %in% selected_parties)
```

The *document-feature matrix* (DFM) is the core data structure for many quantitative text analysis methods. It represents the corpus as a matrix, where documents are represented as rows and features (typically words or n-grams, after pre-processing) are represented as columns. The cell values are usually the counts of each feature in each document. The DFM is created from a `tokens` object (which identifies the individual words in a document) using the `dfm()` function.

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

head(data_dfm)
```

By default, `dfm()` counts the occurrences of each feature (term frequency). If you want to, you can apply different weighting schemes like TF-IDF using `dfm_tfidf()` to emphasise words that are more important to a document relative to the corpus. However, at this point, the DFM contains much information we do not need, such as words like 'the' and '1997'. Because of this, we rarely generate the DFM directly, but we first carry out some pre-processing, to which we will now turn.

4.2 Text Pre-processing

Raw text data is inherently complex and often contains noise that can obscure patterns relevant to your research question. Text pre-processing is the cleaning and normalising of text to make it suitable for quantitative analysis. These steps transform the raw text into a structured format, such as a document-feature matrix, by reducing variability and focusing on meaningful units of text.

Choosing the proper pre-processing steps is not trivial and heavily depends on your research question and the nature of your text data. As highlighted by Denny & Spirling (2018), different pre-processing choices can significantly impact the results of downstream analyses, particularly unsupervised methods like topic modelling or clustering. Understanding what each pre-processing step does and its potential consequences is crucial. The `preText` R package, developed by the authors of that paper, provides tools to evaluate the sensitivity of your results to different pre-processing pipelines.

Pre-processing is typically applied sequentially to the text. In `quanteda`, most pre-processing steps operate on a `tokens` object, transforming each document's list of word sequences before creating the final document-feature matrix.

4.2.1 Tokenisation and Initial Cleaning

The first step in pre-processing is *tokenisation*: splitting the continuous text into discrete units called tokens. These are usually individual words but can also be sentences, paragraphs, or characters. `quanteda`'s `tokens()` function is used for this and allows for initial cleaning during the tokenisation process. By default, `tokens()` splits on whitespace and keeps punctuation, symbols, and numbers:

```
data_tokens <- tokens(data_corpus_ukmanifestos)
head(data_tokens[[5]], 20)
```

```
##  [1] "Manifesto"   "May"         "1997"        "Contents"    ":"
##  [6] "*"           "A"           "New"         "Opportunity" "for"
## [11] "Peace"       "*"           "Unionists"   "*"           "Economy"
## [16] "*"           "Social"      "Justice"     "and"         "Economic"
```

The `head(data_tokens[[5]], 20)` argument here allows us to see the first 20 terms of the 5th object in our corpus (the 1997 Sinn Féin manifesto). As we can see, the raw tokens here include punctuation, numbers and symbols, as `tokens()` does not remove those unless we specify this:

```
data_tokens_cleaned <- tokens(
  data_corpus_ukmanifestos,
  what = "word",
  # Specify that we want to tokenise into words
  remove_punct = TRUE,
  # Remove punctuation marks like ., !?
  remove_symbols = TRUE,
```

```
  # Remove symbols like $, %, ^, &, *
  remove_numbers = TRUE,
  # Remove numerical digits (e.g., 123, 1997)
  remove_url = TRUE,
  # Remove URLs (web addresses).
  remove_separators = TRUE,
  # Remove separator characters like tabs, newlines, and multiple spaces.
  split_hyphens = FALSE,
  # If false, words such as matter-of-fact will not be split
  split_tags = FALSE,
  # If false, do not split social media tags
  include_docvars = TRUE,
  concatenator = "_",
  # The character to connect tokens that should stay together (e.g. we can write conse
  verbose = quanteda_options("verbose")
)

# Display the first 20 tokens of the first document after initial cleaning to observe
head(data_tokens_cleaned[[5]], 20)
```

```
##  [1] "Manifesto"   "May"         "Contents"    "A"           "New"
##  [6] "Opportunity" "for"         "Peace"       "Unionists"   "Economy"
## [11] "Social"      "Justice"     "and"         "Economic"    "Equality"
## [16] "Young"       "People's"    "Rights"      "Education"   "And"
```

```
ntoken(data_tokens_cleaned) # The number of tokens per document
```

```
##  UK_natl_1997_en_Con  UK_natl_1997_en_Lab   UK_natl_1997_en_LD
##                20796                17456                14080
##  UK_natl_1997_en_PCy   UK_natl_1997_en_SF UK_natl_1997_en_UKIP
##                16075                 5857                11839
##  UK_natl_2001_en_Con  UK_natl_2001_en_Lab   UK_natl_2001_en_LD
##                13142                28606                21101
##  UK_natl_2001_en_PCy   UK_natl_2001_en_SF  UK_natl_2001_en_SNP
##                 6542                 7061                10383
##  UK_natl_2005_en_Con  UK_natl_2005_en_Lab   UK_natl_2005_en_LD
##                 7606                23651                16033
##  UK_natl_2005_en_PCy   UK_natl_2005_en_SF  UK_natl_2005_en_SNP
##                 7289                20335                 2585
## UK_natl_2005_en_UKIP  UK_regl_2003_en_PCy
##                 8978                24971
```

```
ntype(data_tokens_cleaned) # The number of unique tokens per document
```

```
##  UK_natl_1997_en_Con  UK_natl_1997_en_Lab   UK_natl_1997_en_LD
##                 3043                 2930                 2756
##  UK_natl_1997_en_PCy   UK_natl_1997_en_SF UK_natl_1997_en_UKIP
```

```
##                 3215                 1741                 2774
##  UK_natl_2001_en_Con  UK_natl_2001_en_Lab   UK_natl_2001_en_LD
##                 2746                 4027                 3852
##  UK_natl_2001_en_PCy   UK_natl_2001_en_SF  UK_natl_2001_en_SNP
##                 1639                 2010                 2366
##  UK_natl_2005_en_Con  UK_natl_2005_en_Lab   UK_natl_2005_en_LD
##                 2027                 3903                 3207
##  UK_natl_2005_en_PCy   UK_natl_2005_en_SF  UK_natl_2005_en_SNP
##                 1904                 4256                 1008
## UK_natl_2005_en_UKIP  UK_regl_2003_en_PCy
##                 2434                 3937
```

The `remove_*` arguments in `tokens()` are powerful for initial cleaning, simplifying the token set by removing elements that might not be relevant to your analysis and reducing noise. Also note that even though we remove punctuation, this is not always the case, such as when we are dealing with a possessive apostrophe (as in People's). We could (if we wanted to) remove this as well, by adding `tokens_remove(pattern = "'s$", valuetype = "regex")`, though we do not recommend this as this would leave the `s` as a single character.

4.2.2 Lower-casing

Converting all text to lower-case is a standard normalisation step. It ensures that the same word is not counted as different features simply because of variations in capitalisation (e.g., "Party", "party", "PARTY"). This step is crucial for consistent term counting, and we generally recommend it unless your analysis requires preserving case information (e.g., for Named Entity Recognition). `quanteda` provides the `tokens_tolower()` function. Note that `keep_acronyms=FALSE` ensures that acronyms (like NATO) are also lower-cased. Set to `TRUE` if you want to preserve the capitalisation of detected acronyms:

```
data_tokens_lower <- tokens_tolower(data_tokens_cleaned, keep_acronyms = FALSE)
head(data_tokens_lower[[5]], 20)
```

```
##  [1] "manifesto"   "may"         "contents"    "a"           "new"
##  [6] "opportunity" "for"         "peace"       "unionists"   "economy"
## [11] "social"      "justice"     "and"         "economic"    "equality"
## [16] "young"       "people's"    "rights"      "education"   "and"
```

```
ntoken(data_tokens_lower)  # The number of tokens per document
```

```
##  UK_natl_1997_en_Con  UK_natl_1997_en_Lab   UK_natl_1997_en_LD
##                20796                17456                14080
##  UK_natl_1997_en_PCy   UK_natl_1997_en_SF UK_natl_1997_en_UKIP
##                16075                 5857                11839
##  UK_natl_2001_en_Con  UK_natl_2001_en_Lab   UK_natl_2001_en_LD
##                13142                28606                21101
##  UK_natl_2001_en_PCy   UK_natl_2001_en_SF  UK_natl_2001_en_SNP
```

```
##                 6542                 7061                10383
##  UK_natl_2005_en_Con  UK_natl_2005_en_Lab   UK_natl_2005_en_LD
##                 7606                23651                16033
##  UK_natl_2005_en_PCy   UK_natl_2005_en_SF  UK_natl_2005_en_SNP
##                 7289                20335                 2585
## UK_natl_2005_en_UKIP  UK_regl_2003_en_PCy
##                 8978                24971
```

```
ntype(data_tokens_lower)  # The number of unique tokens per document
```

```
##  UK_natl_1997_en_Con  UK_natl_1997_en_Lab   UK_natl_1997_en_LD
##                 3043                 2930                 2404
##  UK_natl_1997_en_PCy   UK_natl_1997_en_SF UK_natl_1997_en_UKIP
##                 2906                 1578                 2581
##  UK_natl_2001_en_Con  UK_natl_2001_en_Lab   UK_natl_2001_en_LD
##                 2487                 3554                 3275
##  UK_natl_2001_en_PCy   UK_natl_2001_en_SF  UK_natl_2001_en_SNP
##                 1504                 1789                 2122
##  UK_natl_2005_en_Con  UK_natl_2005_en_Lab   UK_natl_2005_en_LD
##                 1857                 3540                 2845
##  UK_natl_2005_en_PCy   UK_natl_2005_en_SF  UK_natl_2005_en_SNP
##                 1710                 3590                  919
## UK_natl_2005_en_UKIP  UK_regl_2003_en_PCy
##                 2211                 3492
```

After lower-casing, all tokens are uniform and ready for subsequent matching and counting. Notice in the counts that while the number of tokens has not changed (as we have not removed any words), the number of types (unique tokens) has, as now words like "AND", "And" and "and" are not considered as three different terms, but as a single one.

4.2.3 Stopword Removal

Stopwords are high-frequency words (e.g., "the", "a", "is", "in") that are generally considered less informative for distinguishing between documents or topics compared to more substantive terms (such as most verbs and nouns). Removing them can reduce the dimensionality of your DFM, allowing you to focus on more meaningful terms. However, stopwords should be eliminated with care, as in some contexts (e.g., authorship attribution, linguistic style analysis), they can be highly informative. Also, domain-specific "stopwords" might be essential concepts in your field and should not be removed (for example, you might not want to remove the word "we" if this word has become important during a political campaign).

`quanteda` includes built-in lists of stopwords for many languages, accessed via the `stopwords()` function. You can remove these or your custom lists using `tokens_select()`:

```
data_tokens_nostop <- tokens_select(data_tokens_lower, stopwords("english"), selection = "remove"
head(data_tokens_nostop[[5]], 20)
```

```
##  [1] "manifesto"   "may"         "contents"    "new"         "opportunity"
##  [6] "peace"       "unionists"   "economy"     "social"      "justice"
## [11] "economic"    "equality"    "young"       "people's"    "rights"
## [16] "education"   "training"    "farming"     "rural"       "development"
```

```
ntoken(data_tokens_nostop)  # The number of tokens per document
```

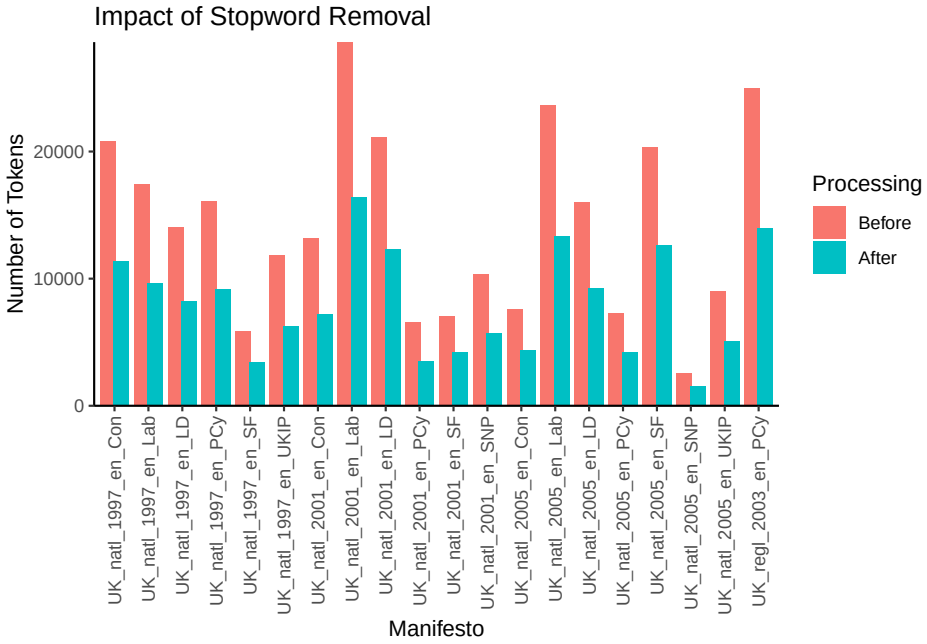
```
##  UK_natl_1997_en_Con  UK_natl_1997_en_Lab   UK_natl_1997_en_LD
##                11358                 9671                 8227
##  UK_natl_1997_en_PCy   UK_natl_1997_en_SF UK_natl_1997_en_UKIP
##                 9185                 3402                 6239
##  UK_natl_2001_en_Con  UK_natl_2001_en_Lab   UK_natl_2001_en_LD
##                 7179                16386                12338
##  UK_natl_2001_en_PCy   UK_natl_2001_en_SF  UK_natl_2001_en_SNP
##                 3508                 4212                 5692
##  UK_natl_2005_en_Con  UK_natl_2005_en_Lab   UK_natl_2005_en_LD
##                 4349                13366                 9263
##  UK_natl_2005_en_PCy   UK_natl_2005_en_SF  UK_natl_2005_en_SNP
##                 4203                12624                 1508
## UK_natl_2005_en_UKIP  UK_regl_2003_en_PCy
##                 5107                13955
```

```
ntype(data_tokens_nostop)  # The number of unique tokens per document
```

```
##  UK_natl_1997_en_Con  UK_natl_1997_en_Lab   UK_natl_1997_en_LD
##                 2934                 2827                 2306
##  UK_natl_1997_en_PCy   UK_natl_1997_en_SF UK_natl_1997_en_UKIP
##                 2801                 1486                 2472
##  UK_natl_2001_en_Con  UK_natl_2001_en_Lab   UK_natl_2001_en_LD
##                 2372                 3444                 3174
##  UK_natl_2001_en_PCy   UK_natl_2001_en_SF  UK_natl_2001_en_SNP
##                 1409                 1694                 2021
##  UK_natl_2005_en_Con  UK_natl_2005_en_Lab   UK_natl_2005_en_LD
##                 1742                 3426                 2714
##  UK_natl_2005_en_PCy   UK_natl_2005_en_SF  UK_natl_2005_en_SNP
##                 1612                 3485                  824
## UK_natl_2005_en_UKIP  UK_regl_2003_en_PCy
##                 2107                 3386
```

Note that this has removed several words, such as 'and' and 'a'. Before we move on, let us visualise the impact of this stopword removal on the total number of tokens per document. This helps us to better understand the reduction in text volume:

```r
# Calculate the number of tokens for each document before and after stopword removal.
tokens_before_stop <- ntoken(data_tokens_lower)
tokens_after_stop <- ntoken(data_tokens_nostop)

# Create a data frame for plotting the impact.
stopwords_data <- data.frame(
  Manifesto = docnames(data_tokens_lower),
  # Get document names
  Before = tokens_before_stop,
  After = tokens_after_stop
)

# Reshape the data from wide to long format for ggplot2 for more straightforward plott
stopwords_data <- melt(
  stopwords_data,
  id.vars = "Manifesto",
  variable.name = "Processing",
  value.name = "NumTokens"
)

ggplot(stopwords_data,
       aes(x = Manifesto, y = NumTokens, fill = Processing)) +
  geom_bar(stat = "identity", position = position_dodge(width = 0.8)) +
  scale_x_discrete(name = "Manifesto") +
  scale_y_continuous(name = "Number of Tokens", expand = c(0, 0)) +
  ggtitle("Impact of Stopword Removal") +
  theme_classic() +
  theme(axis.text.x = element_text(
    angle = 90,
    vjust = 0.5,
    hjust = 1
  ))
```



As expected, removing stopwords substantially reduces the total number of tokens across all documents. The number of words removed differs per document but can be substantial, for example, in the case of the 2001 Labour Party manifesto, which was reduced from 28,606 to 16,386 words — a reduction of around 42%. Finally, remember that `stopwords("english")` is nothing more than a character vector of stop words. Therefore, if we wish to remove stop words ourselves, we can do so by simply providing them as such:

```
data_tokens_nostop <- quanteda::tokens_select(data_tokens_nostop, c("may", "new",
    "manifesto"), selection = "remove", valuetype = "fixed")
```

4.2.4 N-grams and Collocations

While individual words (unigrams) are the most common features in text analysis, sometimes the meaning is better captured by sequences of words, known as **n-grams**. An n-gram is a contiguous sequence of n items from a given sample of text or speech. For example, "strong economy" is a 2-gram (or bigram), and "peace and prosperity" is a 3-gram (or trigram). Using n-grams can help us to capture phrases, multi-word expressions, and local context that single words miss. The `quanteda` package provides `tokens_ngrams()` to generate all possible n-grams of a specified size from a tokens object. Here, `# n = 2` specifies bigrams:

```
data_tokens_bigrams <- tokens_ngrams(data_tokens_nostop, n = 2)
head(data_tokens_bigrams[[5]], 10)
```

```
##  [1] "contents_opportunity" "opportunity_peace"    "peace_unionists"
##  [4] "unionists_economy"    "economy_social"       "social_justice"
##  [7] "justice_economic"     "economic_equality"    "equality_young"
## [10] "young_people's"
```

While `tokens_ngrams()` can generate n-grams, the problem is that it creates *all* possible numbers of n-grams, many of which are not meaningful or useful (such as 'opportunity_peace'). Therefore, it is a better idea first to identify **collocations** – n-grams that appear more frequently than we would expect by chance. These often represent meaningful phrases or concepts (for example, given that we are working with UK manifestos, we would expect combinations such as "Prime Minister" or "National Health Service"). We can identify these with the `textstat_collocations()` function, which calculates various association measures (like likelihood ratio, PMI, and chi-squared) to score potential collocations. Depending on your analysis, you might calculate collocations before stopword removal if you think that stopwords could be part of a collocation you would like to capture (for instance, if you are interested in capturing the term "we the people"). For this example, we will look for collocations after the stopwords are removed. Here, `# min_count` specifies the minimum number of times a collocation must appear to be considered. The standard (2) thus means that a combination appears more than a single time, and is thus not a random combination. Here we set ours at 5:

```
collocations <- textstat_collocations(data_tokens_nostop, min_count = 5, size = 2)
```

Now that we have found all the possible collocations, it is time to decide which ones we want to use. One aspect we can utilise for this is the Wald z-statistic, which calculates the likelihood that the two words would occur together at random. Here, we decide to include only those collocations with a $z > 3$, which means they are three standard errors away from the mean (and thus have a p-value of approximately 0.0027) of being likely:

```
collocations <- collocations[collocations$z > 3, ]
head(collocations, 20)
```

```
##                collocation count count_nested length   lambda        z
## 1        local authorities   145            0      2 6.227087 39.79911
## 2             young people   143            0      2 5.516777 39.77435
## 3           european union   114            0      2 7.258521 39.45265
## 4           health service    92            0      2 5.002104 37.77858
## 5         public services   105            0      2 4.125346 35.71296
## 6               long term    60            0      2 7.560533 35.69555
## 7              income tax    77            0      2 5.246884 35.28413
## 8             party wales    88            0      2 4.497066 34.41696
## 9         small businesses    52            0      2 6.810025 34.11001
## 10        public transport    77            0      2 4.700041 33.27666
## 11             council tax    68            0      2 5.103959 33.10773
## 12          private sector    51            0      2 5.603853 31.87116
```

```
## 13             cymru party    55           0      2 5.169383 31.42865
## 14            labour party    69           0      2 4.471125 31.34627
## 15            human rights    85           0      2 7.954503 30.72153
## 16                per year    58           0      2 4.732218 30.50192
## 17             rural areas    52           0      2 5.087455 30.47553
## 18 conservative government    74           0      2 4.876933 30.34761
## 19         next parliament    49           0      2 5.395004 30.31724
## 20           waiting times    39           0      2 8.122568 30.24217
```

As we can see, we now capture some relevant combinations such as "european union", "local authorities", and "income tax". We could remove all the terms we do not wish to from the data frame, but we will leave this for now. For now, we will "compound" them using `tokens_compound()`. This function replaces the sequence of individual tokens that form a collocation with a single, multi-word token (e.g., "national_health_service"). This ensures that the identified phrase is treated as a single feature from now on (as both the tokens object and the DFM will see `_` as the sign for a compound):

```
data_tokens_compounded <- tokens_compound(data_tokens_nostop, pattern = collocations,
    concatenator = "_")
head(data_tokens_compounded[[5]], 10)
```

```
##  [1] "contents"       "opportunity"    "peace"          "unionists"
##  [5] "economy"        "social_justice" "economic"       "equality"
##  [9] "young_people's" "rights"
```

Note that now the words "social justice" are combined into a single new term. Also, note that while we mainly calculate the collocations after initial lowercasing and punctuation removal, whether we do so before or after stopword removal and stemming depends on whether we see the stopwords and word endings as part of the collocation. For instance, stemming "social security" to "social secur" might be desired, but stemming before identifying the collocation could make it harder for us to identify the collocation.

4.2.5 Stemming and Lemmatisation

Stemming and lemmatisation are two techniques that reduce words to a common form, to group variations (e.g., "run", "running", "ran"). This can help us to treat words with similar meanings as equivalent features, reducing the overall vocabulary size and improving the signal-to-noise ratio in the data.

Stemming uses algorithmic rules to chop off suffixes (and sometimes prefixes) from words, often resulting in a truncated "stem" that may not be a real word (e.g., "university" -> "univers", "connection" -> "connect"). `quanteda` provides the `tokens_wordstem()` function, which uses the Porter stemmer by default for English. While fast, stemming can sometimes produce non-words or conflate words with different meanings.

Lemmatisation is a more linguistically informed process that uses a lexicon (a dictionary of words and their base forms) to convert words to their dictionary form, or *lemma* (e.g., "better" -> "good", "geese" -> "goose", "running" -> "run"). Lemmatisation generally produces valid words. It can be more accurate than stemming but typically requires external resources, such as lexicons. Also, it is computationally more intensive. `quanteda` does not have a built-in lemmatiser. Still, you can perform lemmatisation before creating a `corpus` or `tokens` object using other R packages (like `textstem` or `spacyr`).

Here, we will do some stemming using `tokens_wordstem()`:

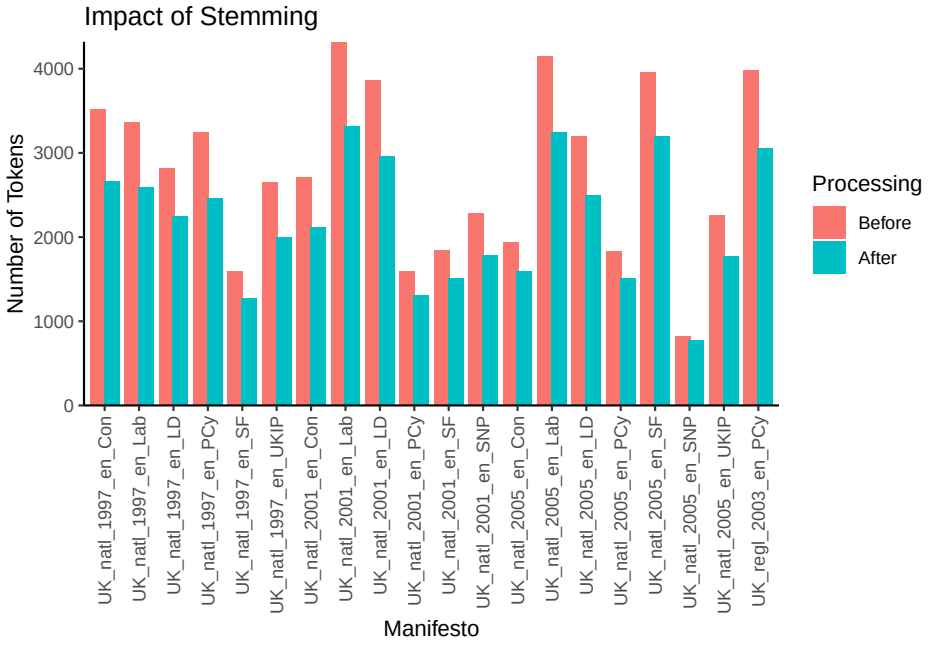
```
data_tokens_stemmed <- tokens_wordstem(data_tokens_compounded, language = "english")
head(data_tokens_stemmed[[5]], 20)
```

```
##  [1] "content"                "opportun"               "peac"
##  [4] "unionist"               "economi"                "social_justic"
##  [7] "econom"                 "equal"                  "young_peopl"
## [10] "right"                  "education_train"        "farming_rural_develop"
## [13] "environ"                "cultur"                 "women"
## [16] "irish_political_prison" "polic"                  "futur"
## [19] "opportun"               "peac"
```

Note how words like "economy", "culture", and "peace" have been reduced to their stems ("economi", "cultur", "peac"). As before, let's visualise this as well:

```
# Calculate the number of types (unique tokens)

types_before_stem <- ntype(data_tokens_compounded)
types_after_stem <- ntype(data_tokens_stemmed)

stemming_data <- data.frame(
  document = docnames(data_tokens_nostop),
  # Get document names
  Before = types_before_stem,
  After = types_after_stem
)

stemming_data <- melt(
  stemming_data,
  id.vars = "document",
  variable.name = "Processing",
  value.name = "NumTypes"
)

# Create a bar plot comparing each document's type counts before and after stemming

ggplot(stemming_data, aes(x = document, y = NumTypes, fill = Processing)) +
  geom_bar(stat = "identity", position = position_dodge(width = 0.8)) + # Use dodged b
```

```
  scale_x_discrete(name = "Manifesto") +
  scale_y_continuous(name = "Number of Tokens", expand = c(0, 0)) +
  ggtitle("Impact of Stemming") +
  theme_classic() +
  theme(axis.text.x = element_text(
    angle = 90,
    vjust = 0.5,
    hjust = 1
  ))  # Rotate x-axis labels
```



Again, removing the words from their stems can remove a significant number of unique features. For example, looking at the 2001 Labour manifesto shows that from the 4320 types before stemming, only 3315 remained afterwards.

4.2.6 Removing Sparse Features (DFM Trimming)

After tokenisation and linguistic pre-processing, we can generate our DFM. However, at this point, our DFM might still contain many features (terms) that appear infrequently across the corpus or in only a few documents. These *sparse features* can increase the dimensionality of our data without adding much useful information and can sometimes be noise (e.g., typos). Removing them can decrease processing time and also the performance of some models. We can use the `dfm_trim()` function for DFM trimming. We can set thresholds based on minimum term frequency (`min_termfreq`), maximum term frequency (`max_termfreq`), minimum document frequency (`min_docfreq`), or maximum

document frequency (`max_docfreq`), either as absolute counts or percentages:

```
data_dfm_untrimmed <- dfm(data_tokens_stemmed)
data_dfm_trimmed <- dfm_trim(data_dfm_untrimmed, min_docfreq = 2)  # Trim the dfm to r

# You could also trim by minimum term frequency (total occurrences across the
# corpus): data_dfm_trimmed_freq <- dfm_trim(data_dfm_untrimmed, min_termfreq =
# 10)

data_dfm_untrimmed
```

```
## Document-feature matrix of: 20 documents, 11,789 features (81.37% sparse) and 6 docv
##                       features
## docs                   conserv foreword administr elect sinc among success
##   UK_natl_1997_en_Con       19        1         3     7   28     3      29
##   UK_natl_1997_en_Lab       43        0         8    15    4     7      19
##   UK_natl_1997_en_LD         2        0         2    12    0     0       4
##   UK_natl_1997_en_PCy        1        1         2     6    0     5       8
##   UK_natl_1997_en_SF         0        0         2     6    5     1       3
##   UK_natl_1997_en_UKIP       3        0         4     8    6     1       6
##                       features
## docs                   british peacetim histori
##   UK_natl_1997_en_Con       32        1       8
##   UK_natl_1997_en_Lab       19        2       7
##   UK_natl_1997_en_LD         6        0       0
##   UK_natl_1997_en_PCy        7        0       3
##   UK_natl_1997_en_SF         9        0       2
##   UK_natl_1997_en_UKIP       2        0       3
## [ reached max_ndoc ... 14 more documents, reached max_nfeat ... 11,779 more features
```

```
data_dfm_trimmed
```

```
## Document-feature matrix of: 20 documents, 6,257 features (69.33% sparse) and 6 docva
##                       features
## docs                   conserv foreword administr elect sinc among success
##   UK_natl_1997_en_Con       19        1         3     7   28     3      29
##   UK_natl_1997_en_Lab       43        0         8    15    4     7      19
##   UK_natl_1997_en_LD         2        0         2    12    0     0       4
##   UK_natl_1997_en_PCy        1        1         2     6    0     5       8
##   UK_natl_1997_en_SF         0        0         2     6    5     1       3
##   UK_natl_1997_en_UKIP       3        0         4     8    6     1       6
##                       features
## docs                   british peacetim histori
##   UK_natl_1997_en_Con       32        1       8
##   UK_natl_1997_en_Lab       19        2       7
##   UK_natl_1997_en_LD         6        0       0
##   UK_natl_1997_en_PCy        7        0       3
```

```
##   UK_natl_1997_en_SF         9        0       2
##   UK_natl_1997_en_UKIP       2        0       3
## [ reached max_ndoc ... 14 more documents, reached max_nfeat ... 6,247 more features ]
```

Note that when we trim the DFM to remove features that appear in fewer than five documents, we reduce the sparsity (i.e., empty cells in the DFM) from 81.37% to 69.33%. We also go from 11,789 for features (types) to 6,257. Trimming significantly reduces the number of features in the DFM by removing those that occur very rarely. The specific thresholds you choose for trimming will depend on the size of your corpus, the nature of your text, and your analytical goals. One reason to use it (and why we set it to 2 here) is that 'unique' words are rarely very interesting. Besides, removing words that occur only a single time can be a good way of removing spelling mistakes. However, be mindful that overly aggressive trimming can remove potentially informative rare terms.

4.2.7 Additional Pre-Processing

Besides using the functions built into `quanteda`, we can also use R's more native functions to clean texts. For this, we have to clean our text while it is still in a data frame. There are several ways to do this, including R's `gsub`. However, we will use the more powerful and faster `stringr` package, part of the `tidyverse`, here. To use `stringr`, we first need to load the library. We must also ensure that our texts are stored as a data frame. For demonstration purposes here, we will take our original (and non-pre-processed) corpus and make it into a data frame using the `convert` function:

```
library(stringr)
data_corpus_df <- convert(data_corpus_ukmanifestos, to = "data.frame")
```

Now we can use `stringr` functions like `str_replace_all()` to clean up the text. `str_replace_all()` takes a character vector, a pattern to find and a replacement string. We will use regular expressions for the pattern.

```
# Remove URLs (http, https, ftp, ftps)
data_corpus_df$text <- str_replace_all(data_corpus_df$text, "http[s]?://[^ ]+", "")
data_corpus_df$text <- str_replace_all(data_corpus_df$text, "ftp[s]?://[^ ]+", "")  # Also includ

# Remove mentions (@username)
data_corpus_df$text <- str_replace_all(data_corpus_df$text, "@\\w+", "")

# Remove hashtags (#topic) - keep the topic word, remove the #
data_corpus_df$text <- str_replace_all(data_corpus_df$text, "#(\\w+)", "\\1")

# Remove punctuation
data_corpus_df$text <- str_replace_all(data_corpus_df$text, "[[:punct:]]", "")

# Remove numbers (optional, depending on analysis)
```

```
data_corpus_df$text <- str_replace_all(data_corpus_df$text, "[[:digit:]]", "")

# Remove retweet indicator 'RT' (ensure it's at the start of a potential tweet)
data_corpus_df$text <- str_replace_all(data_corpus_df$text, "^RT\\s", "")  # Use ^RT\\
# Remove extra whitespace (leading, trailing, and multiple spaces)
data_corpus_df$text <- str_trim(data_corpus_df$text)  # Remove leading/trailing whites
data_corpus_df$text <- str_replace_all(data_corpus_df$text, "[[:space:]]{2,}", " ")  #
# Convert to lower-case (important for dictionary matching)
data_corpus_df$text <- str_to_lower(data_corpus_df$text)
```

Note that in most cases, the functions in `quanteda` are enough. However, `str_replace_all()` can be practical before regular pre-processing, especially when dealing with problematic data.

4.2.8 Evaluating Pre-Processing

The selection and order of these pre-processing steps can substantially impact the resulting DFM and, as a result, the outcomes of your text analysis. For example:

- Removing stopwords might remove terms that are crucial in a specific context or field
- Stemming can group words that have different meanings (e.g., "organ" and "organise")
- The order of removing stopwords before or after calculating collocations affects the collocations we find

In other words, simply pre-processing our data can lead to different data sets and, therefore, to different conclusions later on during our analysis. To see what effect our pre-processing decisions had, we can use the `preText` package by Denny & Spirling (2018) to see what the impact of each step is on our analysis. `preText` works by applying all of the different possible combinations of pre-processing steps (7 in total, leading to $2^7 = 128$ possible combinations) to your corpus and then comparing the similarities between the different DFMs.

First, ensure the `preText` package is installed and loaded:

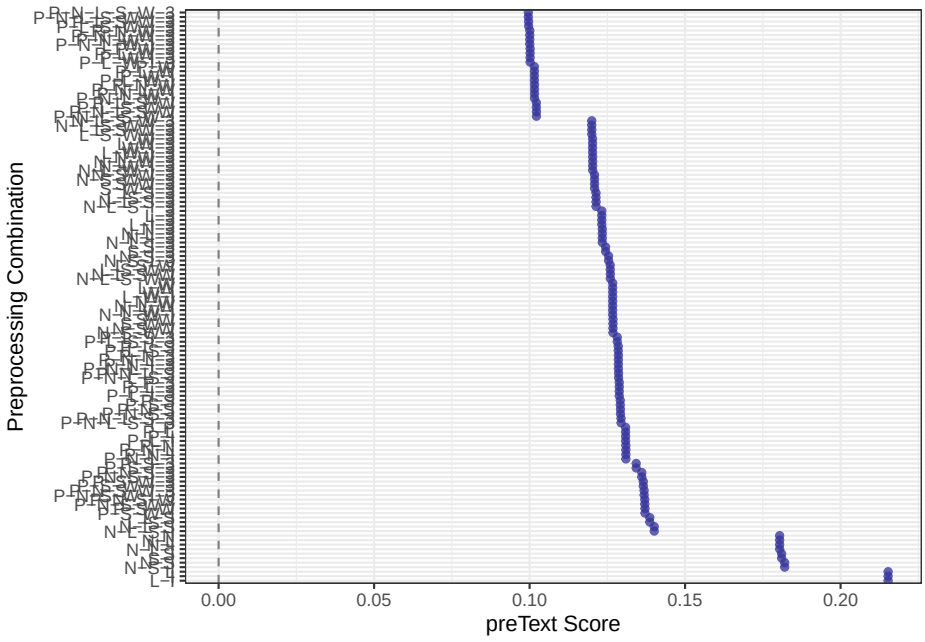
```
library(preText)
library(reshape2)
```

First, we run `factorial_preprocessing()`, which takes a corpus object (or other text formats) and applies a set of default pre-processing pipelines by systematically varying common pre-processing options. After this, we run the main `preText` command:

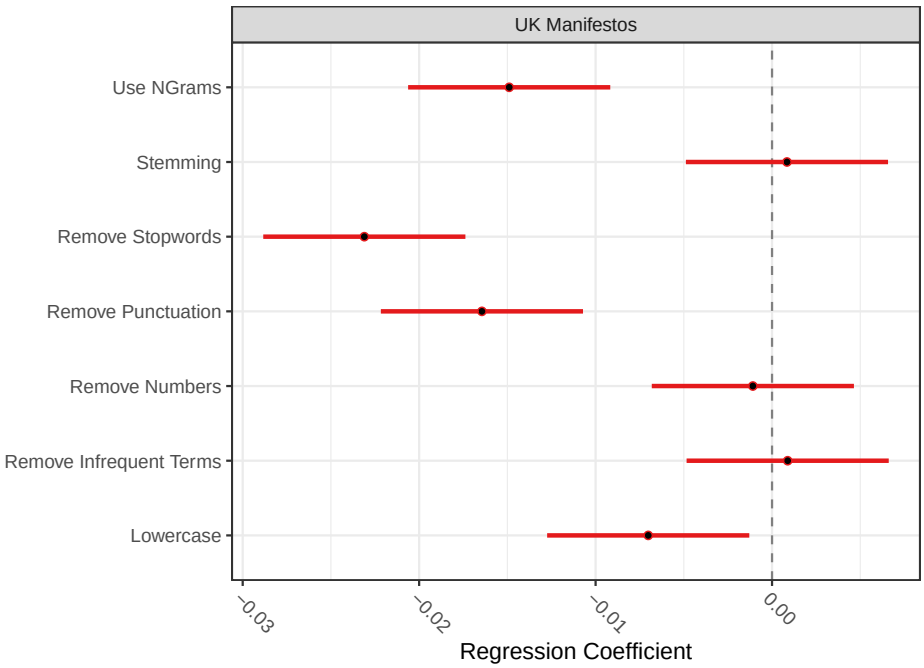
```
# use_ngrams = TRUE: Also consider including n-grams (sequences of words)
# infrequent_term_threshold = 0.01: Remove terms that appear in less than 1% of
# documents

preprocessed_documents <- factorial_preprocessing(data_corpus_ukmanifestos, use_ngrams = TRUE,
    infrequent_term_threshold = 0.01, parallel = FALSE, cores = 1, return_results = TRUE,
    verbose = TRUE  # Display progress and information
)

# Run the preText analysis, which compares the different pre-processed versions
# distance_method = 'cosine': Method to calculate distance between document
# representations num_comparisons = 50: Number of random document pairs to
# compare for robustness

preText_results <- preText(preprocessed_documents, dataset_name = "UK Manifestos",
    distance_method = "cosine", num_comparisons = 50, verbose = TRUE)
```

After running `preText`, we can visualise the results. Two commands are already part of the package and provide a good overview of the results:

```
preText_score_plot(preText_results)
```




```
regression_coefficient_plot(preText_results, remove_intercept = TRUE)
```



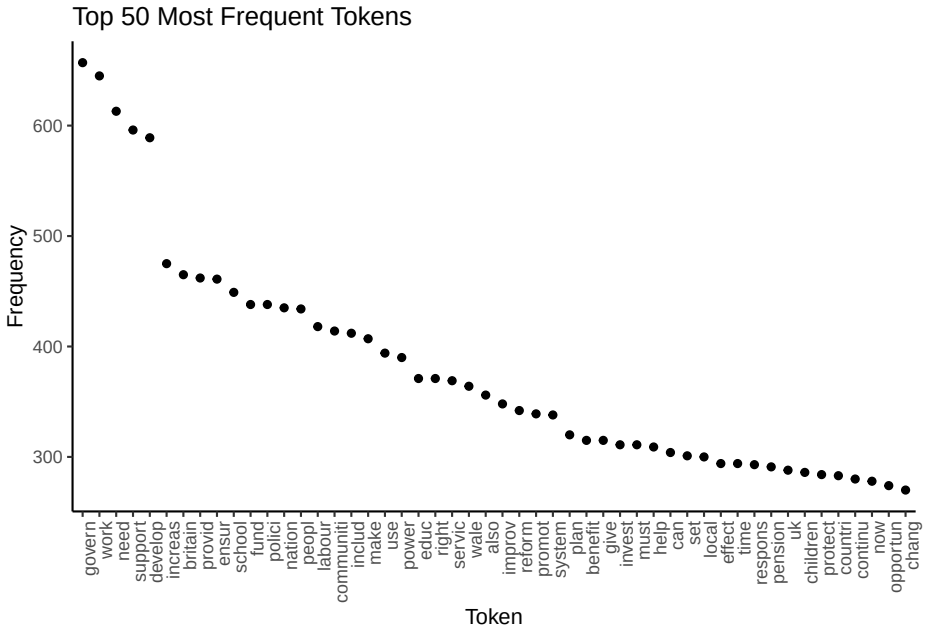
In the first plot, lower scores indicate more robust pre-processing pipelines where document similarities are less sensitive to changes. In the second plot, which is the more interesting of the two, we are looking for pre-processing steps that are significantly different from the null line. In this case, this is the use of n-grams, the removal of stopwords and punctuation and lowercasing. The fact that they are different means that this pre-processing step has led to a significantly different DFM, which is, therefore, more likely to yield significantly different results in further analyses. Note that this does not say whether this is good or bad: we might still prefer a "different" and clean DFM over an uncleaned one. The point is that we are aware of these changes and should consider the potential impacts of our pre-processing on our subsequent analysis.

4.3 Descriptives and Visualisations

Now that we have finished our DFM and assessed the effects of the pre-processing steps, it is time to examine our resulting DFM. For this, `quanteda` offers a variety of options. We start with a look at the most frequent tokens:

```
features <- topfeatures(data_dfm_trimmed, 50)  # Get the top 50 most frequent features
features_plot <- data.frame(term = names(features), frequency = unname(features))

features_plot$term <- with(features_plot, reorder(term, -frequency))  # Reorder the te

ggplot(features_plot, aes(x = term, y = frequency)) + geom_point() + theme_classic() +
```

```
    scale_x_discrete(name = "Token") + scale_y_continuous(name = "Frequency") + ggtitle("Top 50 M
    theme(axis.text.x = element_text(angle = 90, hjust = 1))
```



We can see that the words "govern", "work" and "need" are the most frequent, already indicating a little what our documents are about. Apart from this, a good use of a frequency graph is to spot stopwords you might wish to remove (for example, "also") and include them in the list of stopwords.

Another option for examining frequencies is to use word clouds. Not only are they a visually appealing way to visualise term frequencies, but word size also indicates frequency. While less precise than bar plots, we can use them for a quick impression of prominent terms. To avoid clutter, here, we include only terms that appear at least 20 times and set the maximum number of words in the word cloud at 100:

```
wordcloud_dfm_trim <- dfm_trim(data_dfm_trimmed, min_termfreq = 20)
textplot_wordcloud(wordcloud_dfm_trim, max_words = 100)
```

We can also compare docvars. Here, we first subset the DFM to include only Labour and Conservative manifestoes. Then we group them and then set `compare = TRUE`:

```
data_dfm_compare <- dfm_subset(data_dfm_trimmed, Party %in% c("Lab", "Con"))
wordcloud_dfm_trim <- dfm_group(data_dfm_compare, data_dfm_compare$Party)

textplot_wordcloud(wordcloud_dfm_trim, comparison = TRUE, max_words = 200, color = c("
    "darkred"))
```

Another thing we can do is examine Keywords in Context (KWIC), which displays how specific words or phrases are used by showing them along with their surrounding words. This is useful for understanding the different senses of a word, identifying typical phrases it appears in, and exploring its usage across different documents or contexts. We perform KWIC analysis on the `tokens` object as the order of the words is still essential at this point:

```
# pattern = 'econom*': The term to search for (using glob for wildcard
# matching). This will find 'econom', 'economi', 'economist', etc., after
# stemming.  valuetype = 'glob': Specifies that the pattern is a glob-style
# wildcard pattern.  window = 3: Show 3 words before and 3 words after the
# keyword.

kwic_output <- kwic(data_tokens_stemmed, pattern = "econom*", valuetype = "glob",
    window = 3)
head(kwic_output, 10)
```

```
## Keyword-in-context with 10 matches.
##    [UK_natl_1997_en_Con, 18]              europ becom success | economi |
```

```
##   [UK_natl_1997_en_Con, 67]        global free_market emerg | econom  |
##   [UK_natl_1997_en_Con, 89]            challeng turn around | econom  |
##  [UK_natl_1997_en_Con, 115]                lowest tax major | economi |
##  [UK_natl_1997_en_Con, 505]                 model fail just | econom  |
##  [UK_natl_1997_en_Con, 544]           opportun push forward | econom  |
##  [UK_natl_1997_en_Con, 580]             guis stakehold hard | econom  |
##  [UK_natl_1997_en_Con, 621] enterprise_centre_europ low tax | economi |
##  [UK_natl_1997_en_Con, 638]          public_spend choic two | econom  |
##  [UK_natl_1997_en_Con, 675]                 ensur grow less | economi |
##
##  countri brought knee
##  power rise east
##  fortun fewer_people_work work
##  europ mean continu
##  triumph triumph human
##  revolut began enter
##  evid show great
##  enterpris flourish state
##  philosophi clear year
##  whole economic_cycl time
```

The `kwic()` function output includes the document name (`docname`), the position of the keyword (`from`, `to`), the surrounding context (`pre`, `post`), and the keyword itself (`keyword`).

4.4 Text Statistics

Apart from graphics, we can also calculate a wide range of statistics about our texts, and the `quanteda.textstats` package offers a range of functions for this. We will use the pre-processed DFM (`data_dfm_trimmed`) and tokens (`data_tokens_stemmed`) for these calculations. We will go through the various options in turn.

4.4.1 Summary

`textstat_summary()` provides basic summary statistics for each document in the corpus, such as the number of characters, tokens, types, sentences, and paragraphs. Note that this command works on the original corpus, and not on the cleaned DFM:

```
corpus_summary <- textstat_summary(data_corpus_ukmanifestos)
head(corpus_summary)
```

```
##               document  chars sents tokens types puncts numbers symbols urls
## 1  UK_natl_1997_en_Con 131975  1188  23398  3174   2256     346       0    0
## 2  UK_natl_1997_en_Lab 111444   822  19372  3007   1787     114      15    0
```

```
## 3   UK_natl_1997_en_LD  90883   852  15988  2472   1766     104      38    0
## 4  UK_natl_1997_en_PCy 103411   765  17892  2969   1743      72       2    0
## 5   UK_natl_1997_en_SF  37998   254   6540  1629    620      55       8    0
## 6 UK_natl_1997_en_UKIP  72973   488  13103  2631   1195      61       8    0
##   tags emojis
## 1    0      0
## 2    0      0
## 3    0      0
## 4    0      0
## 5    0      0
## 6    0      0
```

4.4.2 Frequencies

`textstat_frequency()` provides detailed frequency counts for features in a DFM, including term frequency (total occurrences, like we already saw earlier) and document frequency (number of documents the term appears in). It can also group frequencies by document variables, allowing for comparison of term usage across different categories of documents.

```
feature_frequencies <- textstat_frequency(data_dfm_trimmed)
head(feature_frequencies, 10)
```

```
##    feature frequency rank docfreq group
## 1   govern       657    1      20   all
## 2     work       645    2      20   all
## 3     need       613    3      20   all
## 4  support       596    4      20   all
## 5  develop       589    5      19   all
## 6  increas       475    6      20   all
## 7  britain       465    7      16   all
## 8   provid       462    8      20   all
## 9    ensur       461    9      20   all
## 10  school       449   10      18   all
```

```
party_frequencies <- textstat_frequency(data_dfm_trimmed, groups = data_dfm_trimmed@docvars$Party
head(party_frequencies, 10)
```

```
##    feature frequency rank docfreq group
## 1  britain       141    1       3   Con
## 2   govern       124    2       3   Con
## 3   school        97    3       3   Con
## 4     give        87    4       3   Con
## 5     work        82    5       3   Con
## 6    peopl        79    6       3   Con
## 7      can        74    7       3   Con
## 8  continu        73    8       3   Con
```

```
## 9   labour        73    8       3   Con
## 10 pension        72   10       3   Con
```

4.4.3 Lexical diversity

Lexical diversity measures the variety of vocabulary in a text. `textstat_lexdiv()` calculates various measures like the Type-Token Ratio (TTR), which is the number of types (unique tokens) divided by the total number of tokens. A higher TTR generally indicates greater lexical diversity. This function operates on a `tokens` object:

```
corpus_lexdiv <- textstat_lexdiv(data_tokens_stemmed, measure = "TTR")
corpus_lexdiv
```

```
##                document       TTR
## 1   UK_natl_1997_en_Con 0.2733320
## 2   UK_natl_1997_en_Lab 0.3136331
## 3    UK_natl_1997_en_LD 0.3274956
## 4   UK_natl_1997_en_PCy 0.3124604
## 5    UK_natl_1997_en_SF 0.4374573
## 6  UK_natl_1997_en_UKIP 0.3505444
## 7   UK_natl_2001_en_Con 0.3433852
## 8   UK_natl_2001_en_Lab 0.2420945
## 9    UK_natl_2001_en_LD 0.2798416
## 10  UK_natl_2001_en_PCy 0.4446326
## 11   UK_natl_2001_en_SF 0.4259938
## 12  UK_natl_2001_en_SNP 0.3553472
## 13  UK_natl_2005_en_Con 0.4268849
## 14  UK_natl_2005_en_Lab 0.2891995
## 15   UK_natl_2005_en_LD 0.3209892
## 16  UK_natl_2005_en_PCy 0.4569697
## 17   UK_natl_2005_en_SF 0.2959165
## 18  UK_natl_2005_en_SNP 0.5817634
## 19 UK_natl_2005_en_UKIP 0.3882198
## 20  UK_regl_2003_en_PCy 0.2551293
```

4.4.4 Readability

Readability statistics estimate the difficulty of understanding a text based on characteristics like sentence length and the number of syllables per word. `textstat_readability()` calculates various standard scores (e.g., Flesch-Kincaid, Gunning Fog). This function works directly on a `corpus` object or a character vector:

```
corpus_readability <- textstat_readability(data_corpus_ukmanifestos, measure = "Flesch
head(corpus_readability)
```

```
##               document Flesch.Kincaid
## 1  UK_natl_1997_en_Con       10.79018
## 2  UK_natl_1997_en_Lab       12.70561
## 3   UK_natl_1997_en_LD       11.10189
## 4  UK_natl_1997_en_PCy       13.48201
## 5   UK_natl_1997_en_SF       14.27023
## 6 UK_natl_1997_en_UKIP       13.82250
```

4.4.5 Similarity and Distance

These functions calculate the similarity or distance between documents or features based on their representation in a DFM. They help quantify how alike or different texts or words are based on their shared vocabulary and term frequencies. Common measures include cosine similarity and Euclidean distance:

```
# method = 'cosine': Specifies the cosine similarity measure.  margin =
# 'documents': Calculate the similarity between documents (rows of the DFM).

corpus_similarties <- textstat_simil(data_dfm_trimmed, method = "cosine", margin = "documents")

corpus_similarties_matrix <- as.matrix(corpus_similarties)
corpus_similarties_matrix[1:5, 1:5]
```

```
##                     UK_natl_1997_en_Con UK_natl_1997_en_Lab UK_natl_1997_en_LD
## UK_natl_1997_en_Con           1.0000000           0.7569519          0.7502951
## UK_natl_1997_en_Lab           0.7569519           1.0000000          0.7528290
## UK_natl_1997_en_LD            0.7502951           0.7528290          1.0000000
## UK_natl_1997_en_PCy           0.6093673           0.6155356          0.6148482
## UK_natl_1997_en_SF            0.4817308           0.4968049          0.4681049
##                     UK_natl_1997_en_PCy UK_natl_1997_en_SF
## UK_natl_1997_en_Con           0.6093673          0.4817308
## UK_natl_1997_en_Lab           0.6155356          0.4968049
## UK_natl_1997_en_LD            0.6148482          0.4681049
## UK_natl_1997_en_PCy           1.0000000          0.4771206
## UK_natl_1997_en_SF            0.4771206          1.0000000
```

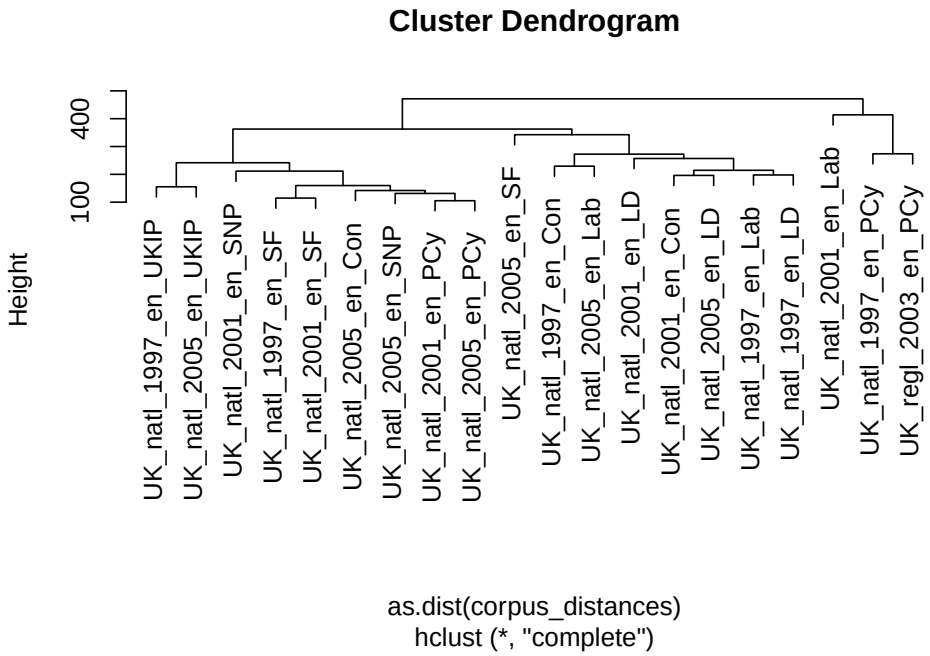
```
# method = 'euclidean': Specifies the Euclidean distance measure margin =
# 'documents': Calculate the distance between documents

corpus_distances <- textstat_dist(data_dfm_trimmed, margin = "documents", method = "euclidean")

# Convert the distance object to a matrix for inspection of pairwise distances

corpus_distances_matrix <- as.matrix(corpus_distances)
corpus_distances_matrix[1:5, 1:5]
```

```
##                     UK_natl_1997_en_Con UK_natl_1997_en_Lab UK_natl_1997_en_LD
```

```
## UK_natl_1997_en_Con              0.0000            230.1652            230.7423
## UK_natl_1997_en_Lab            230.1652              0.0000            197.8080
## UK_natl_1997_en_LD             230.7423            197.8080              0.0000
## UK_natl_1997_en_PCy            290.9605            262.1793            249.6117
## UK_natl_1997_en_SF             309.0372            257.2198            229.1244
##                      UK_natl_1997_en_PCy UK_natl_1997_en_SF
## UK_natl_1997_en_Con             290.9605            309.0372
## UK_natl_1997_en_Lab             262.1793            257.2198
## UK_natl_1997_en_LD              249.6117            229.1244
## UK_natl_1997_en_PCy               0.0000            266.7133
## UK_natl_1997_en_SF              266.7133              0.0000
```

If we would want to, we can also visualise the distances between the documents in the form of a dendrogram, by clustering the distances object:

```
plot(hclust(as.dist(corpus_distances)))
```



The results here are quite interesting, for one in that while both the 1997 and 2003 Plaid Cymru documents are very similar, they are clustered together with the 2001 Labour Party document, which is quite far from its 1997 and 2005 counterparts.

4.4.6 Keyness

Keyness statistics identify terms that are unusually frequent or infrequent in a target group of documents compared to a reference group. This is useful for identifying characteristic terms within a corpus subset and understanding

what distinguishes one set of texts from another. A common measure is the log-likelihood ratio or chi-squared statistic:

```
# Create a logical vector TRUE for documents from the Conservative party and
# FALSE for others. This vector defines the 'target' group.

data_dfm_trimmed@docvars$is_conservative <- data_dfm_trimmed@docvars$Party == "Con"

# Compute keyness statistics comparing the Conservative manifestos (target) to
# all other manifestos (reference) target =
# data_dfm_trimmed@docvars$is_conservative: Specifies the target group using
# the logical vector

keyness_conservative <- textstat_keyness(data_dfm_trimmed, target = data_dfm_trimmed@docvars$is_c

# View the most distinctive terms for the Conservative party (highest keyness
# scores) and the least distinctive terms (lowest keyness scores, which are
# characteristic of the reference group)

head(keyness_conservative, 20)
```

```
##                      feature      chi2            p n_target n_reference
## 1       conservative_govern 144.44686 0.000000e+00       38          14
## 2                   britain  94.85417 0.000000e+00      141         324
## 3  next_conservative_govern  93.74717 0.000000e+00       17           0
## 4                    famili  60.39675 7.771561e-15       69         137
## 5                     choic  54.78260 1.346701e-13       59         113
## 6                         n  52.41623 4.489742e-13       13           3
## 7                      keep  47.73544 4.877987e-12       47          85
## 8                   conserv  46.54836 8.938517e-12       61         131
## 9                    fallen  46.16172 1.088840e-11       19          14
## 10         time_common_sens  45.96519 1.203726e-11       10           1
## 11                   pound  45.15771 1.817879e-11       23          24
## 12                    give  44.16372 3.020273e-11       87         228
## 13                  contin  36.10094 1.873561e-09       10           3
## 14              foot_mouth  36.10094 1.873561e-09       10           3
## 15               lower_tax  35.07639 3.170212e-09        9           2
## 16                   widow  35.07639 3.170212e-09        9           2
## 17                 opt-out  34.50407 4.253600e-09        8           1
## 18                       w  34.50407 4.253600e-09        8           1
## 19              politician  30.71197 2.993130e-08       19          24
## 20                 continu  30.55484 3.245615e-08       73         207
```

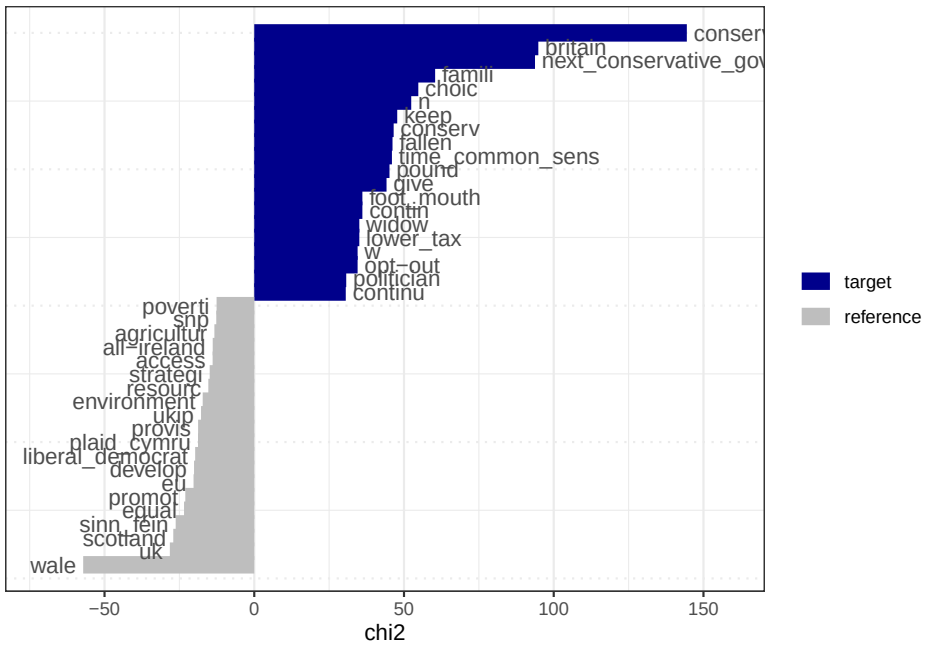
```
tail(keyness_conservative, 20)
```

```
##              feature      chi2            p n_target n_reference
## 6238         poverti -12.58534 3.887848e-04        3         109
```

```
## 6239              snp -12.69011 3.675955e-04        0          75
## 6240       agricultur -13.31662 2.630645e-04        1          91
## 6241     all-ireland -13.87526 1.953533e-04        0          82
## 6242           access -13.90699 1.920823e-04       10         186
## 6243         strategi -14.88492 1.142728e-04        7         164
## 6244          resourc -15.33557 9.000581e-05       12         214
## 6245      environment -17.22185 3.325894e-05        4         148
## 6246             ukip -17.77023 2.492533e-05        0         105
## 6247           provis -18.75683 1.484903e-05        5         168
## 6248      plaid_cymru -18.78653 1.461959e-05        0         111
## 6249 liberal_democrat -19.72714 8.932357e-06        1         129
## 6250          develop -20.11047 7.309549e-06       47         542
## 6251               eu -20.27349 6.712432e-06       12         247
## 6252           promot -23.02328 1.600513e-06       18         321
## 6253            equal -23.45134 1.281138e-06        2         163
## 6254        sinn_féin -26.24228 3.011560e-07        0         155
## 6255         scotland -27.04384 1.988927e-07        3         196
## 6256               uk -28.18851 1.100557e-07       10         278
## 6257             wale -57.12540 4.085621e-14        2         362
```

The output of `textstat_keyness` includes the feature, the keyness score, and the p-value. Positive keyness scores indicate terms that are unusually frequent in the target group. In contrast, negative scores indicate terms unusually infrequent in the target group (and thus characteristic of the reference group). Here, these are words refering to "britain", the family ("famili"), and choice ("choic"). We can of course also visualise this:

```
textplot_keyness(keyness_conservative)
```



4.4.7 Entropy

Entropy measures the randomness or evenness of feature distributions. Here, we can use it to assess the diversity of terms within documents (document entropy) or the evenness of a term's distribution across the corpus (feature entropy). High document entropy means a document uses a wide variety of terms relatively evenly, while low entropy means a few terms dominate. High feature entropy means a term is spread relatively evenly across documents, while low entropy means it's concentrated in a few documents.

```
# margin = 'documents': Calculate entropy for each document (rows of the DFM)

corpus_entropy_docs <- textstat_entropy(data_dfm_trimmed, margin = "documents")
corpus_entropy_docs <- as.data.frame(corpus_entropy_docs)
head(corpus_entropy_docs)
```

```
##                document   entropy
## 1  UK_natl_1997_en_Con 10.332738
## 2  UK_natl_1997_en_Lab 10.383004
## 3   UK_natl_1997_en_LD 10.186036
## 4  UK_natl_1997_en_PCy 10.233130
## 5   UK_natl_1997_en_SF  9.619364
## 6 UK_natl_1997_en_UKIP 10.068977
```

```
# margin = 'features': Calculate entropy for each feature (columns of the DFM)
```

```
corpus_entropy_feats <- textstat_entropy(data_dfm_trimmed, margin = "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
## 4      elect 4.140895
## 16   economi 4.129012
## 86      mean 4.122883
## 380    parti 4.114558
## 1254    forc 4.110995
## 479   polici 4.110578
## 130  increas 4.107211
## 582     mani 4.101916
## 782  greater 4.101321
## 688  societi 4.097388
```

4.5 Exercises

1. Using the UK Manifestos corpus (`data_corpus_ukmanifestos`), calculate each manifesto's readability scores (`textstat_readability`), perhaps using multiple measures. Add these scores to the document variables and plot a chosen readability score against the *Year* docvar. Is there a discernible trend in the readability of UK political manifestos over the selected period?

2. Calculate keyness statistics (`textstat_keyness`) to compare Labour Party manifestos with all other parties in the filtered corpus. What are the 20 most important terms for the Labour Party? Create a visualisation of these terms using `textplot_keyness`.

3. Using the cosine method (`margin = "features"`), explore the similarity of features (words) in the UK manifestos (`data_dfm`). Can you identify pairs of words that tend to appear in similar contexts? (Hint: examine the similarity matrix for high values).

4. Calculate the entropy (`textstat_entropy`) for features in the UK Manifestos DFM. Identify features with very low entropy (close to 0) and examine the documents in which they are concentrated using the `kwic()` function or by examining the DFM directly. What might this tell you about those documents or the use of those specific terms?

5. Apply `textstat_frequency` grouped by party to find the most frequent terms for the Conservative, Labour and Liberal Democrat parties. Create bar plots using the `ggplot2` package to visualise the frequencies of a few selected terms (e.g. 'econom', 'social', 'europ') across these three parties.

Compare this to the grouped frequency plot above, and add more terms for comparison.

Chapter 5

Dictionary Analysis

One of the simplest forms of quantitative text analysis is dictionary analysis. The idea here is to count the presence of pre-defined categories of words or phrases within texts to classify documents or measure the extent to which documents relate to particular topics or sentiments. By relying on a fixed set of terms and associated categories, dictionary methods provide a transparent and computationally efficient approach to text analysis. Unlike statistical or machine learning methods that learn patterns from data, dictionary methods are *non-statistical* and depend entirely on the quality and relevance of the dictionary used. A well-known application is measuring the tone of newspaper articles, speeches, children's writing, etc., using sentiment analysis dictionaries. These dictionaries categorise words as positive, negative or sometimes neutral, allowing a sentiment score to be calculated for a text. Another example is measuring the policy content of different documents, as illustrated by the Policy Agendas Project dictionary (Albaugh et al., 2013), which assigns words to various policy domains.

While straightforward, dictionary analysis has limitations. To begin with, it relies on the assumption that the meaning of a word is relatively stable across different contexts and documents. This can be a strong assumption, as the same word can have different meanings (polysemy), or its meaning can be negated or altered by surrounding words (e.g. sarcasm). Besides, they also cannot identify concepts or themes not explicitly included in the dictionary. Despite these limitations, dictionary analysis remains a valuable tool, especially for exploratory analysis, when validated dictionaries are available or combined with other methods.

Here, we will conduct three analyses using dictionaries: the first is a standard content analysis approach using a political dictionary, and the other two focus on sentiment. For the former, we will use the same political party manifestos we used in the previous chapter, while for the latter, we will use film reviews

and Twitter data.

5.1 Classical Dictionary Analysis

We start with the classical version of dictionary analysis. As for these dictionaries, we can either create them ourselves or use an off-the-shelf version. As with the data, `quanteda` provides access to several off-the-shelf dictionaries relevant to the social sciences:

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

We then apply one of these dictionaries to a document feature matrix (DFM), which is typically created from our corpus after appropriate pre-processing steps, such as tokenisation and lowercase and stopword removal, as we discussed in the previous chapter. As a dictionary, we will use the one created by Laver & Garry (2000), which is designed to estimate political positions from political texts. We first load this dictionary into R and then run it on the DFM using the `dfm_lookup` command. Here, you can use the DFM we created in the previous chapter, though, for this example, we make it from scratch (without any pre-processing):

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

data(data_corpus_ukmanifestos)
data_corpus_ukmanifestos

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

First, let us have a look at the dictionary:

```
data_dictionary_LaverGarry  # Display information about the dictionary object
```

```
## Dictionary object with 9 primary key entries and 2 nested levels.
## - [CULTURE]:
##   - people, war_in_iraq, civil_war
##   - [CULTURE-HIGH]:
##     - art, artistic, dance, galler*, museum*, music*, opera*, theatre*
##   - [CULTURE-POPULAR]:
##     - media
##   - [SPORT]:
##     - angler*
## - [ECONOMY]:
##   - [+STATE+]:
##     - accommodation, age, ambulance, assist, benefit, care, carer*, child*, class, c
```

```
##   - [=STATE=]:
##     - accountant, accounting, accounts, advert*, airline*, airport*, audit*, bank*, bargaining
##   - [-STATE-]:
##     - assets, autonomy, barrier*, bid, bidders, bidding, burden*, charit*, choice*, compet*, c
## - [ENVIRONMENT]:
##   - [CON ENVIRONMENT]:
##     - produc*
##   - [PRO ENVIRONMENT]:
##     - car, catalytic, chemical*, chimney*, clean*, congestion, cyclist*, deplet*, ecolog*, emi
## - [GROUPS]:
##   - [ETHNIC]:
##     - asian*, buddhist*, ethnic*, race, raci*
##   - [WOMEN]:
##     - girls, woman, women
## - [INSTITUTIONS]:
##   - [CONSERVATIVE]:
##     - authority, continu*, disrupt*, inspect*, jurisdiction*, legitimate, manag*, moratorium,
##   - [NEUTRAL]:
##     - administr*, advis*, agenc*, amalgamat*, appoint*, assembly, chair*, commission*, committ
##   - [RADICAL]:
##     - abolition, accountable, answerable, consult*, corrupt*, democratic*, elect*, implement*,
## - [LAW_AND_ORDER]:
##   - [LAW-CONSERVATIVE]:
##     - assaults, bail, burglar*, constab*, convict*, court, courts, custod*, dealing, delinquen
##   - [LAW-LIBERAL]:
##     - harassment, non-custodial
## [ reached max_nkey ... 3 more keys ]
```

Then, we apply it to the DFM:

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

```
## Document-feature matrix of: 101 documents, 20 features (17.23% sparse) and 6 docvars.
##                      features
## docs                  CULTURE.CULTURE-HIGH CULTURE.CULTURE-POPULAR
##   UK_natl_1945_en_Con                    5                       0
##   UK_natl_1945_en_Lab                    3                       0
##   UK_natl_1945_en_Lib                    5                       0
##   UK_natl_1950_en_Con                    2                       0
##   UK_natl_1950_en_Lab                    1                       0
##   UK_natl_1950_en_Lib                    2                       0
##                      features
## docs                  CULTURE.SPORT CULTURE ECONOMY.+STATE+ ECONOMY.=STATE=
##   UK_natl_1945_en_Con             0      14              66              119
##   UK_natl_1945_en_Lab             0      24              61              150
##   UK_natl_1945_en_Lib             0       8              43              106
```

```
##   UK_natl_1950_en_Con             0      11              74             217
##   UK_natl_1950_en_Lab             0      23              56             146
##   UK_natl_1950_en_Lib             0       3              44              99
##                       features
## docs                   ECONOMY.-STATE- ENVIRONMENT.CON ENVIRONMENT
##   UK_natl_1945_en_Con              61                          12
##   UK_natl_1945_en_Lab              67                          18
##   UK_natl_1945_en_Lib              42                           6
##   UK_natl_1950_en_Con              86                          22
##   UK_natl_1950_en_Lab              77                          15
##   UK_natl_1950_en_Lib              48                          11
##                       features
## docs                   ENVIRONMENT.PRO ENVIRONMENT GROUPS.ETHNIC
##   UK_natl_1945_en_Con                           0             0
##   UK_natl_1945_en_Lab                           1             0
##   UK_natl_1945_en_Lib                           4             0
##   UK_natl_1950_en_Con                           3             0
##   UK_natl_1950_en_Lab                           2             0
##   UK_natl_1950_en_Lib                           0             0
## [ reached max_ndoc ... 95 more documents, reached max_nfeat ... 10 more features ]
```

Here, we see that – for example – the 1945 Conservative Party manifesto – contained 5 words related to High Culture while it contained none for Popular Culture. Overall, the `dfm_lookup()` function takes a DFM and a dictionary object as input and returns a new DFM where the features are the categories defined in the dictionary and the values are the aggregated counts of terms belonging to each category within each document.

We can create our own dictionaries that better suit our research question or context. For this, we draw on our theoretical framework to develop different categories and their associated words. Another approach is using reference texts or expert knowledge to identify relevant category terms. We can also combine different dictionaries, as illustrated by Young & Soroka (2012), or integrate keywords from manual coding schemes (Lind et al., 2019). In addition, we can use techniques involving expert or crowd-coding assessments to refine dictionaries or determine the words that best fit different categories (Haselmayer & Jenny, 2017).

If we want to create our dictionary in `quanteda`, we use the `dictionary()` command. To do this, we specify the words in a named list. This list contains keys (the names of the categories) and the values, which are character vectors containing the words or phrases belonging to each category. We can use wildcard characters (such as `*` for glob matching) to include word variations. We then convert this list into a dictionary object. Here, we choose some words that we think will allow us to identify different political stances or issues:

```
dic_list <- list(
    economy = c("tax*", "invest*", "trade", "fiscal policy"), # Include a multi-word phrase
    war = c("army", "troops", "fight*", "military"),
    diplomacy = c("nato", "un", "international relations"),
    government = c("london", "commons", "downing street", "westminster")
)

# tolower = TRUE is often recommended unless you have a specific reason not to lowercase

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

```
## Dictionary object with 4 key entries.
## - [economy]:
##   - tax*, invest*, trade, fiscal policy
## - [war]:
##   - army, troops, fight*, military
## - [diplomacy]:
##   - nato, un, international relations
## - [government]:
##   - london, commons, downing street, westminster
```

If you compare the structure of `dic_list` and the resulting `dic_created` object with that of `data_dictionary_LaverGarry`, you will see that they have a similar structure, defining categories and associated terms. To then apply our created dictionary to the dfm, we use the same 'dfm_lookup' command:

```
# Ensure the dfm used here is preprocessed with tolower = TRUE if the
# dictionary is lowercased

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

```
## Document-feature matrix of: 101 documents, 4 features (16.58% sparse) and 6 docvars.
##                      features
## docs                  economy war diplomacy government
##   UK_natl_1945_en_Con      24   4         0          0
##   UK_natl_1945_en_Lab      13   5         0          0
##   UK_natl_1945_en_Lib      12   5         0          1
##   UK_natl_1950_en_Con      31   4         0          4
##   UK_natl_1950_en_Lab      11   1         0          0
##   UK_natl_1950_en_Lib      20   3         0          1
## [ reached max_ndoc ... 95 more documents ]
```

Note that `dfm_lookup()` matches individual features (words) in the dfm to the dictionary entries. This means that it will correctly match `"tax"`, `"taxes"`, `"taxation"` to the "economy" category if `"tax*"` is in the dictionary. How-

ever, if our dictionary contains multi-word expressions (like "fiscal policy" or "international relations" in our example dic_list), dfm_lookup() will not find them because the dfm loses word order information.

To correctly count multi-word expressions defined in a dictionary, we should apply the dictionary *before* creating the dfm directly to the tokens' object using thetokens_lookup()function. tokens_lookup() preserves the order of the tokens and can therefore match multi-word phrases. The output of tokens_lookup() is a tokens object where the original tokens are replaced by their dictionary categories. We can then convert the resulting token object into a dfm if necessary:

```
# Use tokens_lookup to handle multi-word expressions

dictionary_tokens <- tokens_lookup(data_tokens, dic_created, exclusive = FALSE)  # exc

dictionary_tokens_dfm <- dfm(dictionary_tokens)
```

Comparing dictionary_created_dfm and dictionary_created_dfm_from_tokens shows that the latter correctly identifies and counts the multi-word expressions defined in dic_created. Using tokens_lookup() with exclusive = FALSE means a token can be assigned to multiple categories if it matches entries in more than one. Setting exclusive = TRUE would assign a token to only one category (the first found match). Furthermore, while we can view the resulting dfm by calling it in the console or viewing it in the environment, we can also convert this dfm into a regular data frame for easier manipulation and visualisation. For this, we can use the convert command included in quanteda:

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

You can then use this data frame to normalise these raw counts and compare dictionary results across documents of different lengths by dividing the category counts by either the total number of tokens or the total number of dictionary words in each document.

5.2 Sentiment Analysis

The logic of dictionaries extends beyond simple words, as we saw above; we can also use them to provide measures related to scaling, such as the degree of positive or negative sentiment and look at whether a text expresses happiness, anger, positivity, negativity, etc. This can be particularly useful for analysing subjective content such as movie reviews, which we will look at in the first example.

5.2.1 Movie Reviews

Movie reviews often describe a film alongside an explicit opinion or rating. Here, we will use a sample from the Large Movie Review Dataset (`data_corpus_LMRD`), which contains reviews labelled as either positive or negative and which sometimes have ratings associated with them. As the dataset is large, we will work with a smaller sample of 30 reviews for demonstration purposes. We will sample the corpus using the `corpus_sample()` function and then preprocess it by tokenising, lowercasing, and removing stop words before creating a document feature matrix (DFM):

```
library(quanteda.classifiers)
library(quanteda)

# Load the large movie review dataset and sample 30 reviews
data(data_corpus_LMRD)
set.seed(42)  # Set seed for reproducibility
reviews <- corpus_sample(data_corpus_LMRD, 30)

reviews_tokens <- tokens(reviews, remove_punct = TRUE, remove_symbols = TRUE, remove_numbers = TR
    remove_url = TRUE)
reviews_tokens <- tokens_tolower(reviews_tokens)
reviews_tokens <- tokens_select(reviews_tokens, stopwords("english"), selection = "remove")
reviews_dfm <- dfm(reviews_tokens)
```

The next step is to load a sentiment analysis dictionary and apply it to our film review dfm. Here, we will use the Lexicoder Sentiment Dictionary (LSD2015), which is included in `quanteda.dictionaries`. This dictionary categorises words as positive or negative. We use the dictionary with `dfm_lookup()`:

```
library(quanteda.dictionaries)
data_dictionary_LSD2015

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

The resulting `results_dfm` has features corresponding to the categories in the LSD2015 dictionary (e.g., "positive" and "negative"), and the values are the number of words in each category found in each film review. The next step then is to convert the results into a data frame for easier analysis and display:

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

```
##                 doc_id negative positive neg_positive neg_negative
## 1 train/neg/6869_1.txt       19       31            0            0
## 2  test/neg/6694_1.txt        8        7            0            0
## 3 train/pos/6588_9.txt        8       12            0            0
```

```
## 4 train/pos/7415_8.txt        5       13            0            0
## 5  test/pos/2566_9.txt        8        9            0            0
## 6  test/neg/5680_2.txt       31       19            0            0
```

Often, movie reviews have an external rating (often in the form of stars or a positive/negative label). In that case, we can see if the dictionary-based sentiment is related to that rating. As the `data_corpus_LMRD` sample contains these ratings as document variables (docvars), we can extract this easily:

```
star_data <- docvars(reviews, field = "rating")

# Combine the rating with the dictionary sentiment scores

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

```
##   stars negative positive
## 1     1       19       31
## 2     1        8        7
## 3     9        8       12
## 4     8        5       13
## 5     9        8        9
## 6     2       31       19
```

Now, we can combine the positive and negative counts into a single sentiment score to compare dictionary-based sentiment with star ratings. For this, we take the ratio of positive words to the total number of sentiment words ($positive/(positive+negative)$) to avoid division by zero if there are no positive or negative words:

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

stargraph <- cbind(stargraph, sentiment_ratio)
head(stargraph)
```

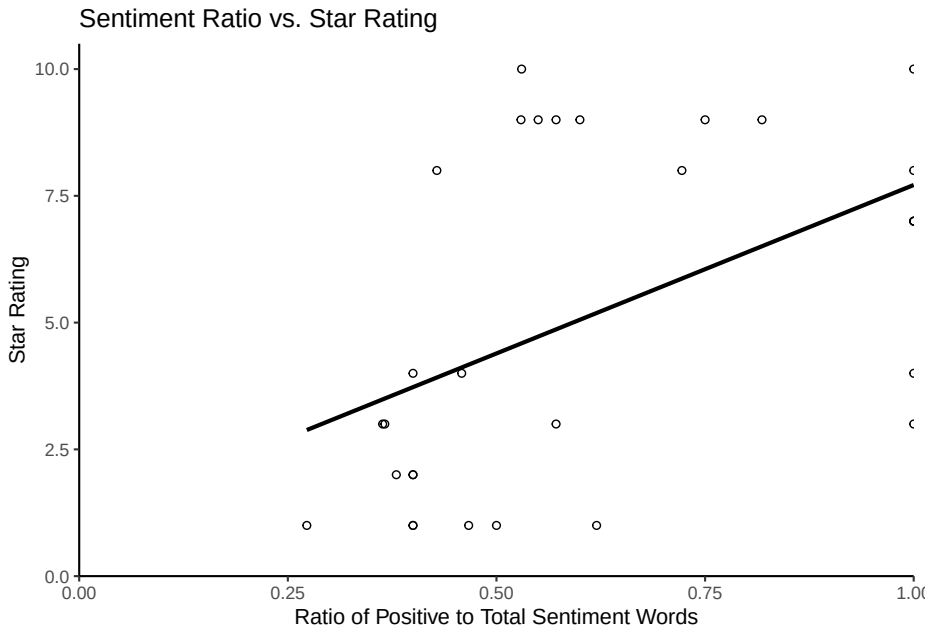
```
##   stars negative positive sentiment_ratio
## 1     1       19       31       0.6200000
## 2     1        8        7       0.4666667
## 3     9        8       12       0.6000000
## 4     8        5       13       0.7222222
## 5     9        8        9       0.5294118
## 6     2       31       19       0.3800000
```

Using `ggplot2`, we can plot the star ratings against these scaled sentiment measures to assess the relationship visually:

```
library(ggplot2)
```

```
ggplot(stargraph, aes(x = sentiment_ratio, y = stars)) + geom_point(shape = 1) +
    geom_smooth(method = lm, se = FALSE, color = "black") + scale_y_continuous(name = "Star Ratin
    limits = c(0, 10.5), expand = c(0, 0)) + scale_x_continuous(name = "Ratio of Positive to Tota
    limits = c(0, 1), expand = c(0, 0)) + ggtitle("Sentiment Ratio vs. Star Rating") +
    theme_classic()
```

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



Finally, we consider how to estimate the uncertainty around our dictionary-based sentiment scores, particularly the percentages of positive or negative words. For this, we use bootstrapping, a statistical technique that calculates the sampling variability of a statistic by resampling the observed data. In the context of text analysis and dictionary methods, bootstrapping can help us quantify the uncertainty in the estimated proportion of words falling into particular dictionary categories within each document. This is particularly useful for understanding the reliability of scores for shorter documents with limited word counts.

The following code demonstrates a bootstrapping approach to estimating confidence intervals for the percentage of positive and negative words in each review. This method involves resampling the word counts within each document based on a multinomial distribution derived from the observed counts. While the code may appear complex, it essentially simulates drawing new sets of words for each document many times based on the proportions of positive and negative words found initially. The core logic is that the `apply` function with `rmultinom` simulates drawing new counts for the negative and positive categories based on their

observed proportions and the total number of sentiment words in each document. We repeat this process `nrepl` several times to obtain a distribution of possible percentages for each document under resampling. The standard deviation of these simulated percentages estimates the standard error, which is then used to calculate confidence intervals.

```
library(ggplot2)  # For plotting
library(dplyr)

# Prepare data for bootstrapping: include doc_id and sentiment counts

reviews_bootstrap_data <- sentiment[, c("doc_id", "negative", "positive")]

# Remove rows with zero total sentiment words to avoid division by zero issues
# later
reviews_bootstrap_data <- reviews_bootstrap_data %>%
    filter(negative + positive > 0)

# Get the number of documents remaining
nman <- nrow(reviews_bootstrap_data)

# Set parameters for bootstrapping
nrepl <- 1000  # Number of bootstrap replications

# --- Perform Bootstrapping --- We will store the results of each bootstrap
# replication in a list
bootstrap_reps <- vector("list", nrepl)

for (i in 1:nrepl) {
    # For each document, simulate drawing word counts from a multinomial
    # distribution The number of trials is the total sentiment words in the
    # document The probabilities are the observed proportions of
    # negative/positive words
    boot_counts <- t(apply(reviews_bootstrap_data[, c("negative", "positive")], 1,
        function(x) {
            total_words <- sum(x)
            # Use rmultinom to draw new counts
            rmultinom(1, size = total_words, prob = x/total_words)[, 1]
        }))

    # Calculate the percentage of negative and positive words for this
    # replication
    total_sentiment_words <- apply(reviews_bootstrap_data[, c("negative", "positive")]
        1, sum)
    percent_negative <- boot_counts[, "negative"]/total_sentiment_words * 100
    percent_positive <- boot_counts[, "positive"]/total_sentiment_words * 100
```

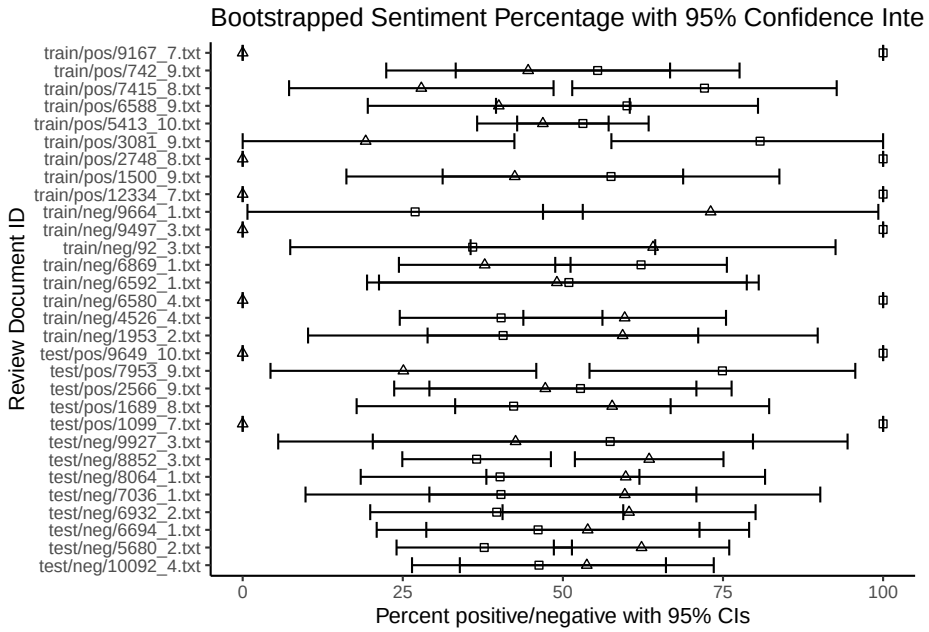
```
    # Store the percentages for this replication along with doc_id
    bootstrap_reps[[i]] <- data.frame(doc_id = reviews_bootstrap_data$doc_id, percent_negative =
        percent_positive = percent_positive)
}

# Aggregate Bootstrapping Results and combine results from all replications
# into a single data frame

all_bootstrap_results <- bind_rows(bootstrap_reps)

# Calculate the mean percentage and standard error (SD of replicates) for each
# document

summary_dataBS <- all_bootstrap_results %>%
    group_by(doc_id) %>%
    summarise(perNegative = mean(percent_negative), NegativeSE = sd(percent_negative),
        perPositive = mean(percent_positive), PositiveSE = sd(percent_positive),
        .groups = "drop"  # Avoid grouping warning
)

# Join with original counts for completeness
dataBS <- reviews_bootstrap_data %>%
    left_join(summary_dataBS, by = "doc_id")

# Calculate the 95% confidence intervals (using 1.96 * Standard Error)

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

# Ensure confidence intervals are within the valid range for percentages [0,
# 100]

dataBS$pos_hi <- pmin(dataBS$pos_hi, 100)
dataBS$pos_lo <- pmax(dataBS$pos_lo, 0)
dataBS$neg_hi <- pmin(dataBS$neg_hi, 100)
dataBS$neg_lo <- pmax(dataBS$neg_lo, 0)

head(dataBS)
```

```
##                 doc_id negative positive perNegative NegativeSE perPositive
## 1 train/neg/6869_1.txt       19       31    37.79400   6.836395    62.20600
## 2  test/neg/6694_1.txt        8        7    53.87333  12.858476    46.12667
## 3 train/pos/6588_9.txt        8       12    39.99000  10.435938    60.01000
```

```
## 4 train/pos/7415_8.txt        5       13    27.89444  10.535295    72.10556
## 5  test/pos/2566_9.txt        8        9    47.25294  12.038145    52.74706
## 6  test/neg/5680_2.txt       31       19    62.28200   6.986078    37.71800
##   PositiveSE   pos_hi   pos_lo    neg_lo   neg_hi
## 1   6.836395 75.60533 48.80667 24.394665 51.19333
## 2  12.858476 71.32928 20.92405 28.670720 79.07595
## 3  10.435938 80.46444 39.55556 19.535561 60.44444
## 4  10.535295 92.75473 51.45638  7.245266 48.54362
## 5  12.038145 76.34182 29.15230 23.658178 70.84770
## 6   6.986078 51.41071 24.02529 48.589288 75.97471
```

We can then produce a graph showing each review's estimated percentages of positive and negative words overlaid with their 95% confidence intervals:

```
ggplot() +
 geom_point(data = dataBS, aes(x = perPositive, y = doc_id), shape = 0) + # Plot mean p
 geom_point(data = dataBS, aes(x = perNegative, y = doc_id), shape = 2) + # Plot mean n
 geom_errorbarh(data = dataBS, aes(xmax = pos_hi, xmin = pos_lo, y = doc_id)) + # Error
 geom_errorbarh(data = dataBS, aes(xmax = neg_hi, xmin = neg_lo, y = doc_id)) + # Error
 scale_x_continuous(name = "Percent positive/negative with 95% CIs") +
 scale_y_discrete(name = "Review Document ID") +
 ggtitle("Bootstrapped Sentiment Percentage with 95% Confidence Intervals") +
 theme_classic()
```



Note that the fact that some documents are shorter than others and contain fewer dictionary words introduces more uncertainty to the estimates of the percentages. As can be seen from the overlapping confidence intervals for many

documents, the estimated rate of negative words is not statistically different from that of positive words at the 95% confidence level for these reviews. Based on this dictionary and bootstrapping method, the sentiment for these reviews appears to be mixed or uncertain. The width of the error bars provides a visual indication of this uncertainty for each document. While bootstrapping quantifies this uncertainty, its interpretation requires careful consideration of the underlying assumptions.

5.2.2 Twitter

Now, let us turn to another example of sentiment analysis using Twitter/X data. Due to its informal nature and use of slang, hashtags and emoticons, this type of text presents unique challenges, especially when cleaning. Here, we examine sentiment towards several major US airlines based on a dataset of tweets. In this case, researchers scraped data from Twitter and asked participants to classify the sentiment of each tweet as negative, positive or neutral, and if negative, to explain why. The data also includes information about the coders' confidence levels, the airline in question and the tweets' metadata. We can download this data from platforms such as Kaggle (e.g. the 'Airline Sentiment' dataset), but for ease of use in this example, we will load it directly from GitHub via a URL:

```
urlfile = "https://raw.githubusercontent.com/SCJBruinsma/qta-files/master/Tweets.csv"
tweets <- read.csv(url(urlfile), stringsAsFactors = FALSE)  # Use stringsAsFactors = FALSE to kee
head(tweets)
```

```
##       tweet_id airline_sentiment airline_sentiment_confidence negativereason
## 1 5.703061e+17           neutral                        1.0000
## 2 5.703011e+17          positive                        0.3486
## 3 5.703011e+17           neutral                        0.6837
## 4 5.703010e+17          negative                        1.0000     Bad Flight
## 5 5.703008e+17          negative                        1.0000     Can't Tell
## 6 5.703008e+17          negative                        1.0000     Can't Tell
##   negativereason_confidence        airline airline_sentiment_gold       name
## 1                        NA Virgin America                          cairdin
## 2                    0.0000 Virgin America                        jnardino
## 3                        NA Virgin America                       yvonnalynn
## 4                    0.7033 Virgin America                        jnardino
## 5                    1.0000 Virgin America                        jnardino
## 6                    0.6842 Virgin America                        jnardino
##   negativereason_gold retweet_count
## 1                                 0
## 2                                 0
## 3                                 0
## 4                                 0
## 5                                 0
## 6                                 0
##
```

```
## 1
## 2                                                                    @VirginAmerica plu
## 3                                                                     @VirginAmerica I
## 4          @VirginAmerica it's really aggressive to blast obnoxious "entertainment"
## 5                                                                                    @
## 6 @VirginAmerica seriously would pay $30 a flight for seats that didn't have this pl
##   tweet_coord             tweet_created tweet_location
## 1             2015-02-24 11:35:52 -0800
## 2             2015-02-24 11:15:59 -0800
## 3             2015-02-24 11:15:48 -0800      Lets Play
## 4             2015-02-24 11:15:36 -0800
## 5             2015-02-24 11:14:45 -0800
## 6             2015-02-24 11:14:33 -0800
##                user_timezone
## 1 Eastern Time (US & Canada)
## 2 Pacific Time (US & Canada)
## 3 Central Time (US & Canada)
## 4 Pacific Time (US & Canada)
## 5 Pacific Time (US & Canada)
## 6 Pacific Time (US & Canada)
```

After cleaning the text data in the data frame, we transform it into a `quanteda` corpus object, specifying that our text is in the `text` field. We then proceed with the standard `quanteda` preprocessing steps: transforming our corpus into a tokens object and removing stop words:

```
corpus_tweets <- corpus(tweets, text_field = "text")

data_tweets_tokens <- tokens(corpus_tweets, remove_punct = TRUE, remove_symbols = TRUE
    remove_numbers = TRUE, remove_url = TRUE, remove_separators = TRUE, split_hyphens =
    split_tags = FALSE)

data_tweets_tokens <- tokens_select(data_tweets_tokens, stopwords("english"), selectio

data_tweets_dfm <- dfm(data_tweets_tokens)
```

Now, we can apply our sentiment dictionary. As discussed earlier, we can do this in two ways: by applying it to the dfm using `dfm_lookup()` or to the tokens object using `tokens_lookup()`. Both should give similar results for single-word entries, but we have to use `tokens_lookup()` to correctly identify multi-word expressions. As the LSD2015 dictionary contains some multi-word expressions, using `tokens_lookup()` and then converting the result to a dfm is the preferred approach to ensure that all dictionary entries are captured:

```
results_tokens <- tokens_lookup(data_tweets_tokens, data_dictionary_LSD2015)

# Convert the resulting tokens object (with categories) to a dfm
```

```
results_dfm <- dfm(results_tokens)

# Convert the dfm to a data frame for analysis
results_df <- convert(results_dfm, to = "data.frame")
```

Now, let us see how well our dictionary-based sentiment matches the human-assigned sentiment labels in the original dataset. We recode the human-assigned `airline_sentiment` labels from our original dataset into numerical values for easier comparison (e.g. positive = 1, negative = -1, neutral = 0):

```
library(car)

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

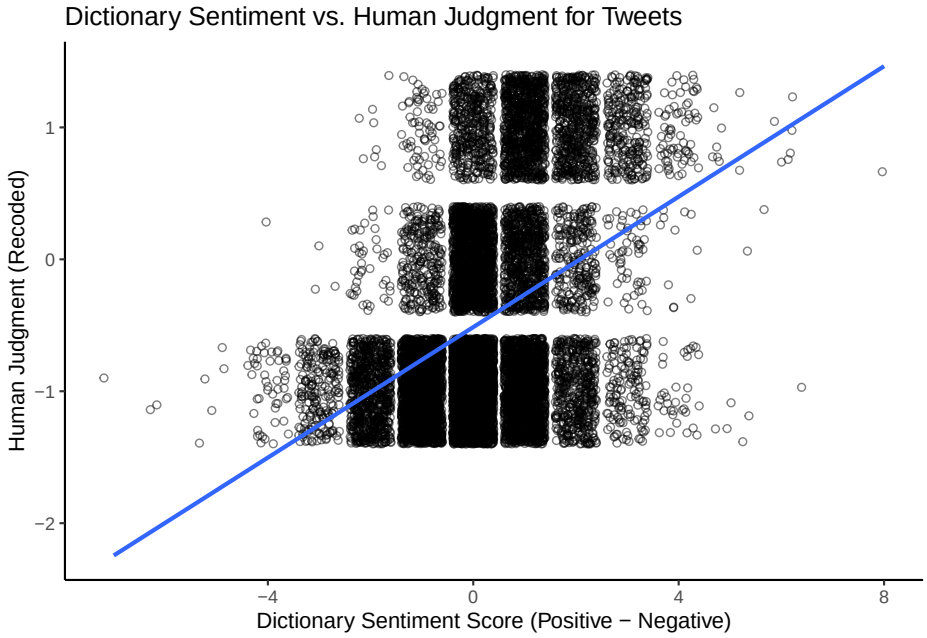
A quick look at the table shows how human-assigned sentiment is distributed. Perhaps not unexpected, negative tweets about airlines are more common than positive ones. We now want to combine this data with the output of our dictionary analysis to calculate an overall sentiment score for each tweet. One common method is subtracting the negative score from the positive score (positive minus negative). A higher resulting score indicates a more positive dictionary-based sentiment:

```
comparison_df <- as.data.frame(cbind(results_df$positive, results_df$negative, sentiment_numeric)
names(comparison_df) <- c("positive_dict", "negative_dict", "human_sentiment")

# Calculate the sentiment difference from dictionary counts
comparison_df$sentiment_difference_dict <- comparison_df$positive_dict - comparison_df$negative_d

head(comparison_df)
```

```
##   positive_dict negative_dict human_sentiment sentiment_difference_dict
## 1             0             0               0                         0
## 2             0             1               1                        -1
## 3             0             0               0                         0
## 4             1             3              -1                        -2
## 5             0             1              -1                        -1
## 6             0             1              -1                        -1
```

Finally, we can visualise the relationship between human-assigned and dictionary-based sentiment scores using a scatter plot. Since human sentiment is categorical (or comprises a small set of numerical values), adding jitter to the scores can help to visualise density. A simple linear regression line can illustrate the overall trend. We will use the results from the `tokens_lookup()` function, as this handles multi-word expressions correctly.

```r
library(ggplot2)

ggplot(comparison_df,
       aes(x = sentiment_difference_dict, y = human_sentiment)) +
  geom_jitter(shape = 1, alpha = 0.5) + # Add jitter and transparency
  geom_smooth(method = lm, se = FALSE) + # Add linear regression line
  scale_x_continuous(name = "Dictionary Sentiment Score (Positive - Negative)") +
  scale_y_continuous(name = "Human Judgment (Recoded)") +
  ggtitle("Dictionary Sentiment vs. Human Judgment for Tweets") +
  theme_classic()
```



This graph visually shows the correlation between dictionary-based and human-assigned sentiment scores. A positive slope suggests that human coders rate tweets with higher positive-minus-negative dictionary scores as more positive. The strength of this relationship (e.g., measured by the correlation coefficient or R^2 from the linear model) indicates how well the dictionary captures the sentiment as perceived by humans in that particular domain.

5.2.3 VADER

Another popular dictionary-based approach for sentiment analysis in social media contexts is VADER (Hutto & Gilbert, 2014) (Valence Aware Dictionary and sEntiment Reasoner). Unlike a simple dictionary lookup, VADER is a rule-based model that considers punctuation, capitalisation, emojis, and negation to determine sentiment intensity. It provides a continuous sentiment score ranging

from -1 (most negative) to +1 (most positive) and scores for the proportions of positive, negative and neutral sentiment. Unlike most dictionaries, which rely on the judgement of a single expert or small group, the VADER dictionary was developed and validated using crowdsourced human judgements.

We can use the `vader` package to use VADER in R; let's test it again using the airline tweet data. First, we reload the data and select a subset of tweets to work with to speed up processing, converting the text into a character vector:

```
urlfile = "https://raw.githubusercontent.com/SCJBruinsma/qta-files/master/Tweets.csv"
tweets_vader <- read.csv(url(urlfile), stringsAsFactors = FALSE)
# Select a sample of 1000 tweets for demonstration
set.seed(42)
tweets_sample_vader <- tweets_vader[sample(nrow(tweets_vader), 1000), ]
text_vader <- tweets_sample_vader$text  # Extract the text column
```

We then apply VADER to our tweets using the `vader_df()` function, which is designed to work with a character vector or data frame of text.

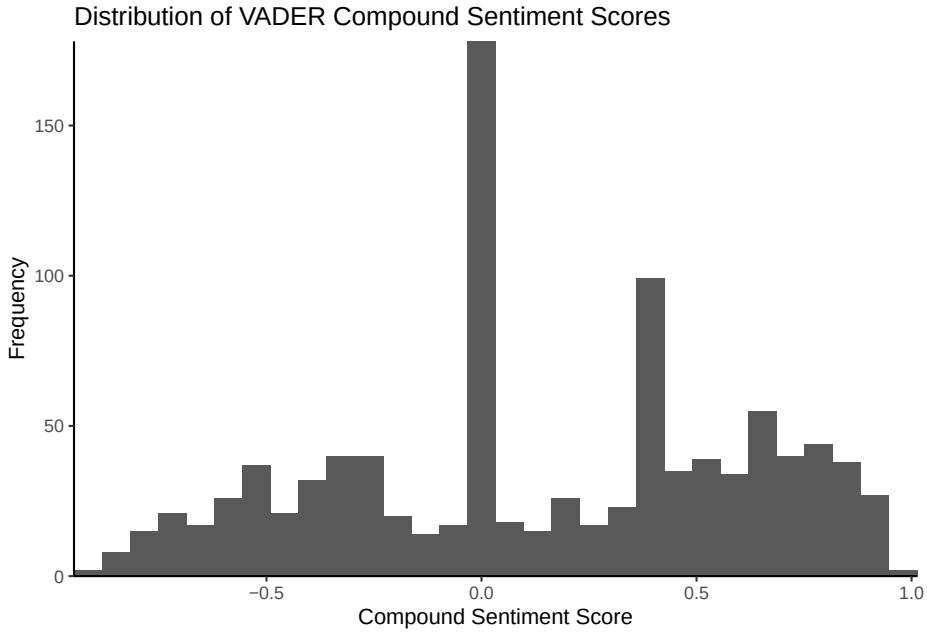
```
library(vader)

results_vader <- vader_df(text_vader)  # Apply vader_df to the extracted text vector
```

VADER then provides us with a data frame consisting of several variables. The most important ones are:

- `text`: The original text of the tweet.
- `compound`: A single, aggregated sentiment score ranging from -1 to +1. This is often the primary score we would use
- `pos`, `neg`, `neu`: The proportion of the text that falls into positive, negative, and neutral categories, respectively.
- `word_scores`: (If requested) Individual sentiment scores for each word.

To get a better idea of the output, we can look at the distribution of the `compound` sentiment scores using a histogram:

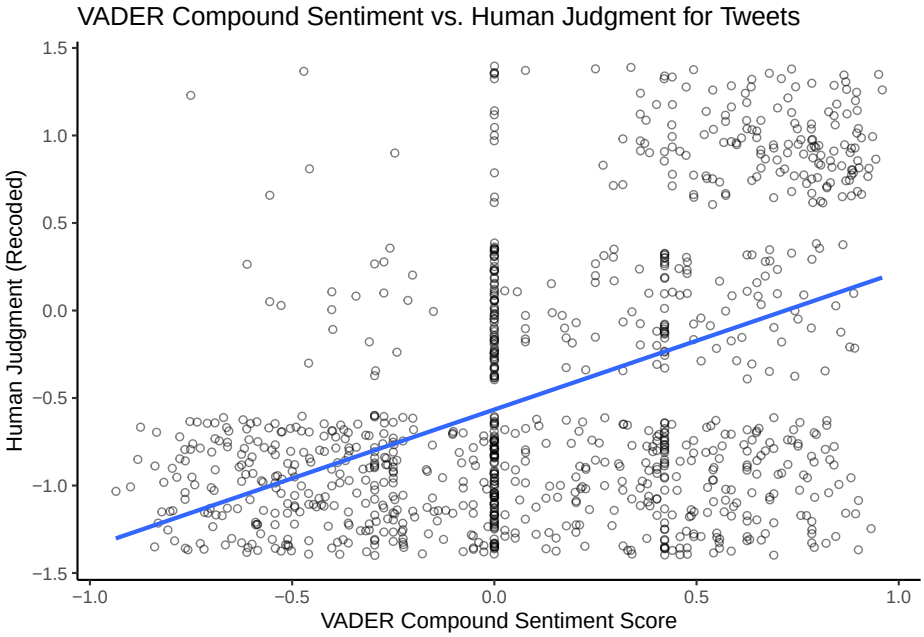
```
library(ggplot2)

ggplot(data = results_vader, aes(x = compound)) + geom_histogram(bins = 30) + scale_x_continuous(
    expand = c(0, 0)) + scale_y_continuous(name = "Frequency", expand = c(0, 0)) +
    ggtitle("Distribution of VADER Compound Sentiment Scores") + theme_classic()
```



The histogram illustrates the frequency of tweets across the range of composite sentiment scores. In this dataset, a significant proportion of tweets tend to cluster around a neutral score (0), potentially due to the presence of purely informative tweets or the absence of strong emotional language. Examining tweets with scores close to zero can help us understand why they are classified as neutral by VADER. Tweets such as '@JetBlue Counting on your flight 989 to get to DC!' may lack explicit positive or negative language and, therefore, receive a neutral or near-neutral composite score. As before, we can compare the VADER composite score with the human-assigned sentiment labels:

```
vader_comparison_df <- data.frame(vader_compound = results_vader$compound, human_sentir

vader_comparison_df <- na.omit(vader_comparison_df)

ggplot(vader_comparison_df, aes(x = vader_compound, y = human_sentiment)) +
    geom_jitter(shape = 1, alpha = 0.5) + # Add jitter and transparency
    geom_smooth(method = lm, se = FALSE) + # Add linear regression line
    scale_x_continuous(name = "VADER Compound Sentiment Score") +
    scale_y_continuous(name = "Human Judgment (Recoded)") +
    ggtitle("VADER Compound Sentiment vs. Human Judgment for Tweets") +
    theme_classic()
```

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



Comparing this plot with the previous one using LSD2015 provides us with some additional insight into which dictionary/approach better matches human judgement for this dataset.

5.3 Exercises

1. Using a sentiment analysis dictionary such as LSD2015, analyse the sentiment expressed in the Wikipedia article on the Cold War from the previous chapter. Are there differences in sentiment between the paragraphs, and if so, what might the reasons for these differences be? Consider plotting the sentiment score against the document index.

2. How did the Cold War influence culture, as reflected in textual data? Create your own dictionary of words or phrases representing cultural aspects influenced by the Cold War (e.g. terms related to fear, propaganda and specific cultural phenomena). Apply this dictionary to a relevant corpus (e.g. articles about the Cold War or cultural texts from the period) and analyse the frequency of these terms. Discuss your findings based on the dictionary counts.

3. Explore the different sentiment scores provided by the `vader` package (pos, neg, neu and compound). Plot the distribution of the 'pos', 'neg' and 'neu' scores. How do these relate to the 'compound' score and the human-assigned labels?

4. Research other sentiment dictionaries available in R, for example, in pack-

ages such as `syuzhet` or `tidytext` that can work with `quanteda` outputs. Apply one of these dictionaries to the film review or tweet data, then compare the results with those obtained using LSD2015 or VADER. What similarities and differences are there in the sentiment scores?

5. Consider the limitations of dictionary analysis, especially with regard to context and negation. Find examples in the tweet data where a simple dictionary lookup might misclassify the sentiment due to negation ('not happy') or sarcasm. How might these cases be handled in a more advanced approach to sentiment analysis?

Chapter 6

Scaling Methods

While methods like dictionary analysis help identify themes or sentiments, their dictionary categories are often treated as distinct and not inherently ordered on a scale. If we want to compare texts or place them along a continuum (e.g., a left-right political spectrum, a scale of formality, or a sentiment dimension), we need methods to place documents on a scale.

In this chapter, we will look at three prominent scaling methods: *Wordscores* (Laver et al., 2003), *Wordfish* (Slapin & Proksch, 2008) and *Correspondence Analysis* (specifically Multiple Correspondence Analysis for categorical text data). The first two were initially part of the main `quanteda` package but have since moved to the `quanteda.textmodels` package. Correspondence Analysis, meanwhile, is a dimensionality reduction technique that can position documents and features in a multidimensional space and reveal relationships between them. For this, we will mainly use functions from the `FactoMineR` and `factoextra` packages, but we will also look at the `textmodel_ca` from `quanteda.textmodels`.

6.1 Wordscores

Wordscores is a supervised scaling method that requires a set of *reference texts* for which the position on the scale of interest is already known. These positions are then used to estimate scores for individual words, which are aggregated to estimate the position of new texts (for which the position is unknown) on the same scale. The basic idea is that words that frequently appear in texts with known extreme positions on a scale are likely to indicate that position. For instance, words that frequently appear in texts known to be on the far left of the scale might receive low scores, while words that frequently appear in texts on the far right might receive high scores. The score of a new text is calculated by taking the weighted average of the scores of the words it contains, where the

weights are usually the word frequencies.

Here, we aim to position the 2005 manifestos of the five main UK political parties (Labour, Liberal Democrats, Conservatives, Scottish National Party and Plaid Cymru) on a general left-right scale. The 2001 manifestos of the same parties are used as reference texts. We will use external ratings, such as those from the 2002 Chapel Hill Expert Survey (Bakker et al., 2012) or other expert judgements, to determine their positions on the scale.

First, we load the necessary libraries and the corpus of UK party manifestos provided in `quanteda.corpora`. We then subset the corpus to include only the 2001 and 2005 manifestos of the selected parties. We then create a document feature matrix (DFM) following standard preprocessing steps.

```
library(quanteda)
library(quanteda.corpora)
library(dplyr)  # For data manipulation
library(stringr)  # For string manipulation
library(ggplot2)  # For plotting

# Load the UK manifestos corpus
data(data_corpus_ukmanifestos)

corpus_manifestos <- corpus_subset(data_corpus_ukmanifestos, Year %in% c(2001, 2005))
corpus_manifestos <- corpus_subset(corpus_manifestos, Party %in% c("Lab", "LD", "Con",
    "SNP", "PCy"))

# Tokenise the corpus with standard preprocessing
data_manifestos_tokens <- tokens(corpus_manifestos, what = "word", remove_punct = TRUE
    remove_symbols = TRUE, remove_numbers = TRUE, remove_url = TRUE, remove_separators
    split_hyphens = FALSE, include_docvars = TRUE, padding = FALSE)

data_manifestos_tokens <- tokens_tolower(data_manifestos_tokens)
data_manifestos_tokens <- tokens_select(data_manifestos_tokens, stopwords("english"),
    selection = "remove")

data_manifestos_dfm <- dfm(data_manifestos_tokens)

# Print the document names to verify the order
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"
```

The order of the documents in the DFM is essential for assigning reference scores. We can check this by looking at the document names. Next, we set

the known scores (from the 2002 Chapel Hill Expert Survey) for the reference texts (the 2001 manifestos). We set the score to 'NA' for the new texts (the 2005 manifestos). These scores should match the order of the documents in the DFM. Finally, we run the `textmodel_wordscores()` function, providing it with the DFM and the vector of reference scores:

```
library(quanteda.textmodels)

scores <- c(7.7, 5.2, 3.8, 3.2, 3, NA, NA, NA, NA, NA)

# Run the Wordscores model
ws <- textmodel_wordscores(data_manifestos_dfm, scores)

# Display the summary of the Wordscores model, including word 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    7.7  7179   0  92 0.8646      0
## UK_natl_2001_en_Lab    5.2 16386   0 166 1.9735      0
## UK_natl_2001_en_LD     3.8 12338   0 101 1.4860      0
## UK_natl_2001_en_PCy    3.2  3508   0  72 0.4225      0
## UK_natl_2001_en_SNP    3.0  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
##        5.828        6.525        7.359        7.143        4.475        3.979
##        lives      raising       family       living       safely      earning
##        6.034        4.423        5.513        4.717        5.738        6.034
##      staying      healthy      growing        older      knowing        world
##        6.938        4.289        4.741        6.276        7.700        4.365
##       leader     stronger      society         town      country    civilised
##        4.520        4.904        4.338        7.499        4.397        4.272
##        proud    democracy   conclusion      present    ambitious    programme
##        6.064        5.255        6.938        3.589        4.468        4.230
```

The `summary()` output shows the estimated score for each word based on its

distribution across the reference texts. These are the "word scores." A text's score is a weighted average of these word scores. Wordscores also calculates a raw score for each text. However, these raw scores are often clustered around the mean. To make the virgin text scores comparable to the reference scale, a *rescaling* step is applied. `predict()` with a `textmodel_wordscores` object performs this rescaling. There are two standard rescaling methods:

1. **LBG (Laver–Benoit–Garry)** (Laver et al., 2003): Rescales the raw scores linearly to match the range of the reference scores.
2. **MV (Martin–Vanberg)** (Martin & Vanberg, 2008): Rescales the raw scores to match the median and variance of the reference scores. This method calculates standard errors and confidence intervals for the estimated text scores.

Here, we apply both rescaling methods using the `predict()` function. We also request standard errors and confidence intervals for the MV rescaling:

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

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

```
print(pred_lbg)
```

```
## UK_natl_2001_en_Con UK_natl_2001_en_Lab  UK_natl_2001_en_LD UK_natl_2001_en_PCy
##            8.773880            5.455744            3.952488            1.923355
## UK_natl_2001_en_SNP UK_natl_2005_en_Con UK_natl_2005_en_Lab  UK_natl_2005_en_LD
##            2.170182            5.651194            5.130845            5.036429
## UK_natl_2005_en_PCy UK_natl_2005_en_SNP
##            3.750297            4.285161
```

```
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.
```

```
print(pred_mv)
```

```
## $fit
##                            fit      lwr      upr
## UK_natl_2001_en_Con 7.700000 7.614308 7.785692
## UK_natl_2001_en_Lab 5.338408 5.302650 5.374166
## UK_natl_2001_en_LD  4.268507 4.227170 4.309845
## UK_natl_2001_en_PCy 2.824328 2.749561 2.899095
## UK_natl_2001_en_SNP 3.000000 2.933107 3.066893
## UK_natl_2005_en_Con 5.477515 5.379667 5.575363
## UK_natl_2005_en_Lab 5.107171 5.058585 5.155756
## UK_natl_2005_en_LD  5.039972 4.977774 5.102171
```

```
## UK_natl_2005_en_PCy 4.124604 4.035171 4.214036
## UK_natl_2005_en_SNP 4.505278 4.317356 4.693200
##
## $se.fit
## UK_natl_2001_en_Con UK_natl_2001_en_Lab  UK_natl_2001_en_LD UK_natl_2001_en_PCy
##          0.04372127          0.01824402          0.02109101          0.03814726
## UK_natl_2001_en_SNP UK_natl_2005_en_Con UK_natl_2005_en_Lab  UK_natl_2005_en_LD
##          0.03412945          0.04992337          0.02478895          0.03173450
## UK_natl_2005_en_PCy UK_natl_2005_en_SNP
##          0.04562959          0.09588018
```

The `predict()` function returns scaled scores for reference and virgin texts. The scores for the reference texts can be used to evaluate how well the Wordscores model recovers the original known positions. The warning "n features in data not used in prediction" indicates that some words in the virgin texts were not present in the reference texts (or vice versa) and, therefore, could not be used in the scoring process. This common problem highlights the importance of vocabulary overlap between reference and novel texts (and is one of the disadvantages of using Wordscores (Bruinsma & Gemenis, 2019)).

Now, we can compare the original reference scores with those predicted by Wordscores for the reference documents to assess the model's performance in recovering the known scores. The Concordance Correlation Coefficient (CCC) developed by Lin (1989) is an appropriate metric for this, as it measures the agreement between two variables by assessing how far the data points deviate from the 45-degree line (which would indicate perfect agreement). We calculate the CCC using the LBG scores for the 2001 manifestos and their original reference scores. We use the `CCC()` function from the `DescTools` package:

```
library(DescTools)
```

```
##
## Caricamento pacchetto: 'DescTools'
```

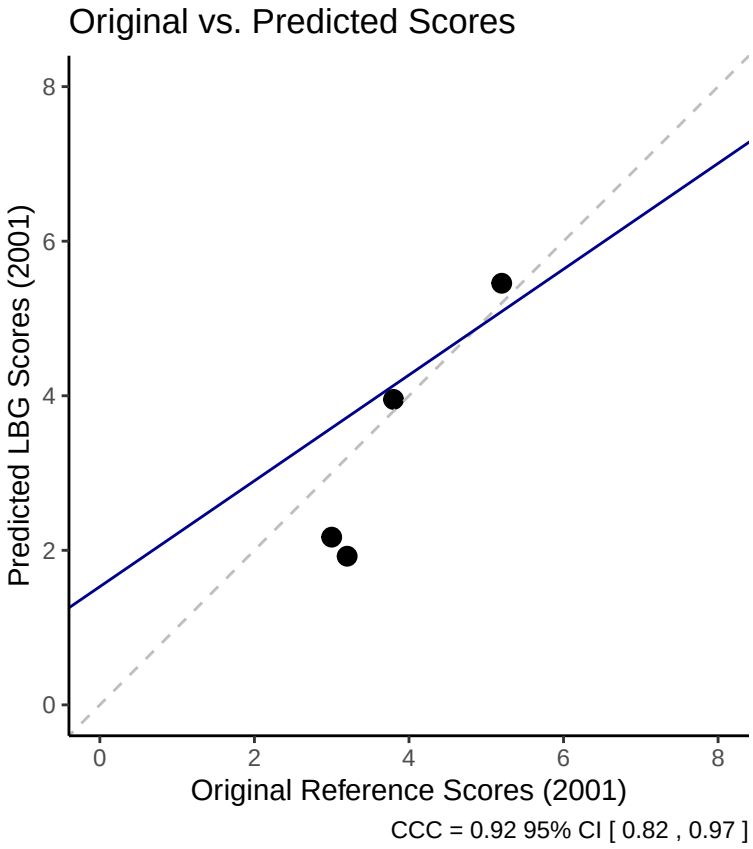
```
## Il seguente oggetto è mascherato da 'package:car':
##
##     Recode
```

```
# Create a data frame with original reference scores, and LBG predicted scores
# for the 2001 manifestos Select the first 5 rows corresponding to the 2001
# manifestos based on the dfm order

comparison_data <- data.frame(original_scores = scores[1:5], predicted_lbg = pred_lbg[1:5])

ccc_result <- CCC(comparison_data$original_scores, comparison_data$predicted_lbg,
    ci = "z-transform", conf.level = 0.95, na.rm = TRUE)
```

The CCC value indicates the level of agreement. A value closer to 1 indicates greater agreement. The confidence interval provides a range for the true CCC

value. We can examine the level of agreement in more detail by plotting the original scores against the predicted scores for the reference texts. Including the 45-degree line (perfect concordance) and a linear regression line (reduced major axis) helps visualise the relationship and identify potential outliers.

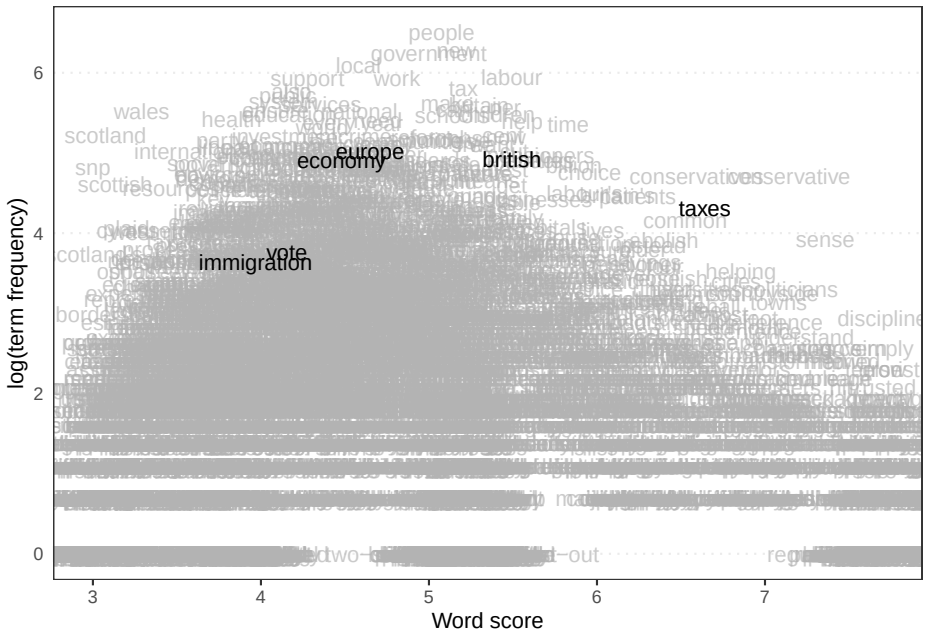
```
library(ggplot2)

# Calculate the linear model (Reduced Major Axis) for visualisation

lm_line <- lm(comparison_data$original_scores ~ comparison_data$predicted_lbg)

ggplot(comparison_data, aes(x = original_scores, y = predicted_lbg)) +
  geom_point(size = 3) + # Use slightly larger points
  scale_x_continuous(name = "Original Reference Scores (2001)", limits = c(0,8)) +
  scale_y_continuous(name = "Predicted LBG Scores (2001)", limits = c(0,8)) +
  ggtitle("Original vs. Predicted Scores") +
  coord_equal(ratio = 1) + # Ensure equal scaling for x and y axes
  geom_abline(intercept = 0, slope = 1, linetype = "dashed", color = "grey") + # 45-deg
  geom_abline(intercept = coef(lm_line)[1], slope = coef(lm_line)[2], linetype = "solid
  theme_classic() +
  labs(caption = paste("CCC =", round(ccc_result$rho.c[1], 2),
                       "95% CI [", round(ccc_result$rho.c[2], 2), ",",
                       round(ccc_result$rho.c[3], 2), "]")) # Add CCC to caption
```



This graph enables us to inspect the level of agreement visually. Points close to the dashed 45-degree line suggest accurate predictions. Deviations from this line show areas where the model struggles to reproduce the original values. Identifying points far from the line (outliers) can help us to diagnose problems, such as a particular reference text whose language does not align well with its assigned score compared to the other texts.

As well as positioning the texts, we can look at the word scores themselves. Plotting word scores can show which words are most strongly associated with different positions on the scale. The `textplot_scale1d()` function from `quanteda.textplots` is useful. We specify `margin = "features"` to plot the word scores and can highlight specific words of interest.

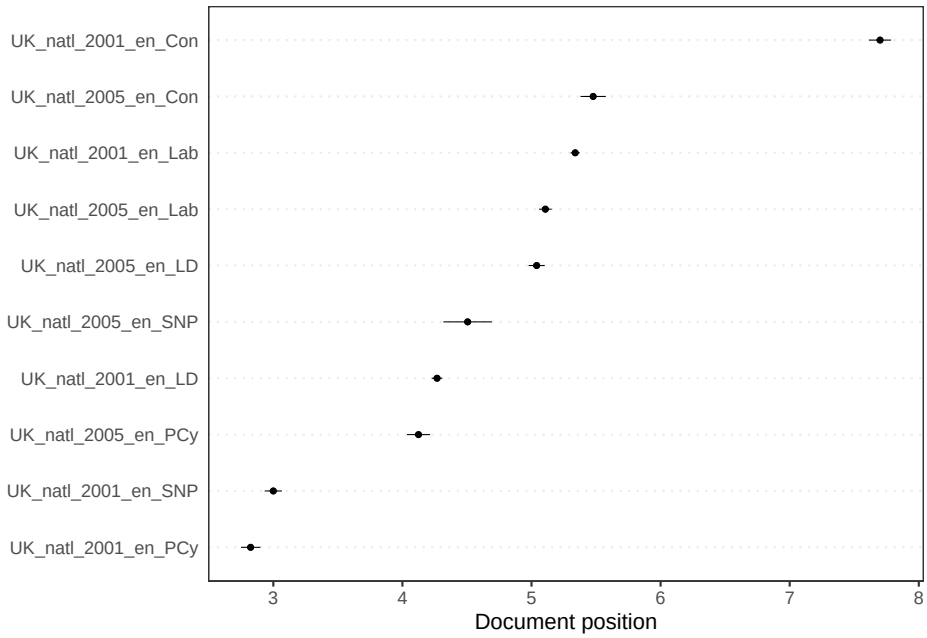
```
library(quanteda.textplots)

# Plot the distribution of word scores
textplot_scale1d(ws, margin = "features", highlighted = c("british", "vote", "europe",
    "taxes", "economy", "immigration")  # Highlight relevant words on the plot
)
```



The position of words on the scale indicates their estimated ideological leaning based on the reference texts. Words like 'taxes' or 'economy' might be expected to fall towards one end (e.g. left). In contrast, words related to national identity or immigration might be closer to the other (e.g. right), depending on the corpus and the reference scores.

Finally, we can examine the confidence intervals around the estimated text scores, which is particularly important for the virgin texts, as it measures the uncertainty in their estimated positions. The `textplot_scale1d()` function can also plot document scores with confidence intervals when `margin = "documents"` is specified, but this requires the standard errors to be available, which is the case for the MV rescaling method (`pred_mv`):

```
textplot_scale1d(pred_mv, margin = "documents"  # Specify that we want to plot documen
)
```



The length of the error bars in this graph indicates the width of the confidence interval. Longer error bars indicate greater uncertainty in the estimated position of the document in question. This uncertainty may be due to various factors, such as the number of dictionary words in the document, variability in the scores of these words and the quantity and quality of the reference texts. We can manually create this document scaling plot using `ggplot2` and the results from `pred_mv`. This requires manipulating the data to extract the scores and confidence intervals and document information into a data frame, for which we can use the `dplyr` package:

```
library(dplyr)
library(stringr)
library(tibble)

# Extract predicted fit and standard error

scores_fit <- as.data.frame(pred_mv$fit) %>%
  rename(fit = 1) # Rename the column to 'fit'

scores_se <- as.data.frame(pred_mv$se.fit) %>%
  rename(se = 1) # Rename the column to 'se'

# Combine fit and SE data frames and add document metadata

data_textplot <- scores_fit %>%
  bind_cols(scores_se) %>% # Combine the 'fit' and 'se' data frames side by side
```

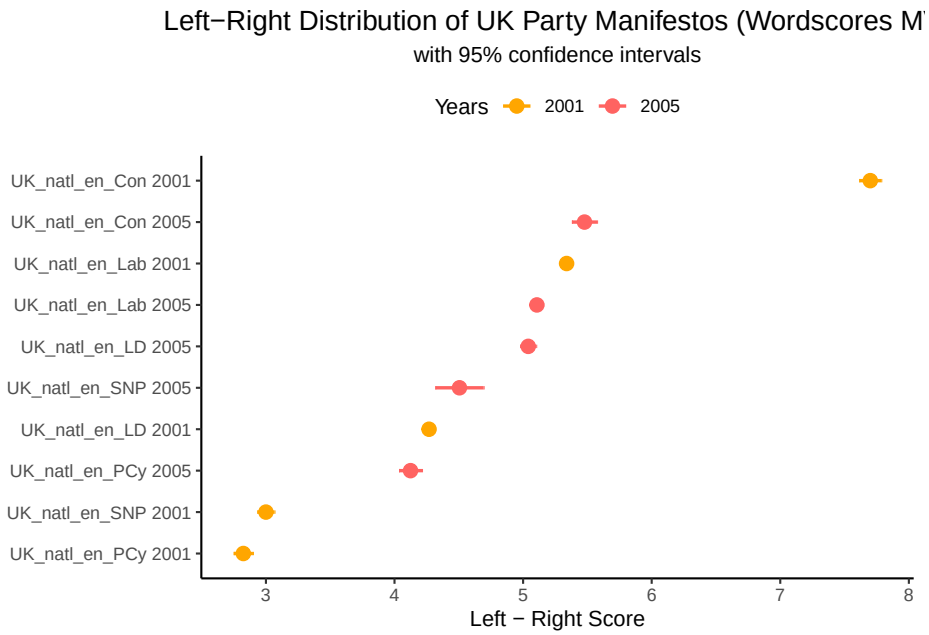
```
  rownames_to_column("doc_id") %>% # Convert the row names (which are the document IDs)
  mutate(
    lwr = fit - 1.96 * se,
    # Calculate the lower bound of the 95% confidence interval (assuming normal distri
    upr = fit + 1.96 * se,
    # Calculate the upper bound of the 95% confidence interval
    year = factor(str_extract(doc_id, "\\d{4}")),
    # Extract the year (four digits) from the doc_id and convert to a factor
    party = str_replace(doc_id, "\\d{4}_", ""),
    # Remove the year and underscore from the doc_id to get the party name
    label = factor(paste(party, year), levels = paste(party, year)[order(fit)]) # Crea
  ) %>%
  select(doc_id, fit, lwr, upr, se, year, party, label) # Select and reorder the colum

head(data_textplot)
```

```
##                doc_id      fit      lwr      upr         se year          party
## 1 UK_natl_2001_en_Con 7.700000 7.614306 7.785694 0.04372127 2001 UK_natl_en_Con
## 2 UK_natl_2001_en_Lab 5.338408 5.302650 5.374166 0.01824402 2001 UK_natl_en_Lab
## 3  UK_natl_2001_en_LD 4.268507 4.227169 4.309846 0.02109101 2001  UK_natl_en_LD
## 4 UK_natl_2001_en_PCy 2.824328 2.749560 2.899097 0.03814726 2001 UK_natl_en_PCy
## 5 UK_natl_2001_en_SNP 3.000000 2.933106 3.066894 0.03412945 2001 UK_natl_en_SNP
## 6 UK_natl_2005_en_Con 5.477515 5.379665 5.575364 0.04992337 2005 UK_natl_en_Con
##                 label
## 1 UK_natl_en_Con 2001
## 2 UK_natl_en_Lab 2001
## 3  UK_natl_en_LD 2001
## 4 UK_natl_en_PCy 2001
## 5 UK_natl_en_SNP 2001
## 6 UK_natl_en_Con 2005
```

Now that we have the data in a convenient format, we can plot the estimated document positions with their confidence intervals using `ggplot2`. Ordering the documents by their estimated score on the y-axis is often helpful for visualisation.

```
library(ggplot2)

ggplot(data_textplot, aes(x = fit, y = label, colour = year)) +
  geom_point(size = 3) +
  scale_x_continuous(name = "Left - Right Score") +
  geom_errorbarh(aes(xmax = upr, xmin = lwr),
                 height = 0,
                 linewidth = 0.8) +
  theme_classic() +
  scale_colour_manual(
```

```
    values = c("2001" = "#ffa600", "2005" = "#ff6361"),
    # Define custom colours for the years 2001 and 2005
    name = "Years"
  ) + # Set the title for the colour legend
  ggtitle("Left-Right Distribution of UK Party Manifestos (Wordscores MV)",
          subtitle = "with 95% confidence intervals") +
  theme(
    plot.title = element_text(size = 14, hjust = 0.5),
    plot.subtitle = element_text(hjust = 0.5),
    legend.position = "top",
    axis.title.y = element_blank()
  )
```



This graph shows the estimated left-right position of each manifesto in 2001 and 2005, as well as the uncertainty of the 2005 estimates. It shows the parties' relative positions and how their estimated values may have changed between the two election years. The width of the confidence intervals indicates which estimates are less specific, potentially because of limited text length or vocabulary overlap with the reference texts.

6.2 Wordfish

Unlike Wordscores, Wordfish is an unsupervised scaling method. It does not require pre-scored reference texts. Instead, it models word frequencies in documents based on a Poisson distribution while simultaneously estimating docu-

ment positions and word-specific parameters. The model assumes that a word's frequency in a document is related to the document's position on a single latent dimension, the word's overall tendency to occur in the document, and its specific association with the latent dimension.

The output of Wordfish includes * θ (theta): The estimated position of each document on the latent dimension. * α (alpha): The estimated intercept for each word, representing its overall frequency. * β (beta): The estimated weight for each word, representing its association with the latent dimension. Words with large positive beta values are more likely to appear in documents with high theta values, and vice versa for large negative beta values. * ψ (psi): A document-specific fixed effect that captures variations in document length.

Wordfish estimates a single latent dimension. The direction of this dimension is arbitrary (e.g. left to right or right to left). We can orient the dimension by specifying two *anchor texts* or by selecting a direction using the `dir` argument in `textmodel_wordfish()`.

Let's apply Wordfish to a corpus of US presidential inaugural speeches. We will use speeches after 1900 and preprocess the texts similarly to the Wordscores example. We could choose speeches from presidents typically considered at opposite ends of a relevant dimension (such as a left-right scale) to orient the scale. For example, we could use the 1965 Johnson and the 1985 Reagan speeches to define the direction, arbitrarily assigning one speech to one end of the scale and the other to the other.

```
library(quanteda)
data(data_corpus_inaugural)
set.seed(42)

corpus_inaugural <- corpus_subset(data_corpus_inaugural, Year > 1900)

# Tokenise and preprocess the corpus
data_inaugural_tokens <- tokens(
 corpus_inaugural,
 what = "word",
 remove_punct = TRUE, # Remove punctuation
 remove_symbols = TRUE, # Remove symbols
 remove_numbers = TRUE, # Remove numbers
 remove_url = TRUE, # Remove URLs
 remove_separators = TRUE, # Remove separators
 split_hyphens = FALSE, # Do not split hyphenated words
 include_docvars = TRUE # Include document variables (metadata)
)

data_inaugural_tokens <- tokens_tolower(data_inaugural_tokens)
data_inaugural_tokens <- tokens_select(data_inaugural_tokens, stopwords("english"), sel
```

```
data_inaugural_dfm <- dfm(data_inaugural_tokens)

# Print document names to identify indices for direction. We needed the order of documents to spe
data_inaugural_dfm@Dimnames$docs
```

```
##  [1] "1901-McKinley"   "1905-Roosevelt"  "1909-Taft"       "1913-Wilson"
##  [5] "1917-Wilson"     "1921-Harding"    "1925-Coolidge"   "1929-Hoover"
##  [9] "1933-Roosevelt"  "1937-Roosevelt"  "1941-Roosevelt"  "1945-Roosevelt"
## [13] "1949-Truman"     "1953-Eisenhower" "1957-Eisenhower" "1961-Kennedy"
## [17] "1965-Johnson"    "1969-Nixon"      "1973-Nixon"      "1977-Carter"
## [21] "1981-Reagan"     "1985-Reagan"     "1989-Bush"       "1993-Clinton"
## [25] "1997-Clinton"    "2001-Bush"       "2005-Bush"       "2009-Obama"
## [29] "2013-Obama"      "2017-Trump"      "2025-Trump"      "2021-Biden.txt"
```

```
# Identify the indices of the anchor documents (1965 Johnson and 1985 Reagan)
johnson_index <- which(docnames(data_inaugural_dfm) == "1965-Johnson")
reagan_index <- which(docnames(data_inaugural_dfm) == "1985-Reagan")

wordfish <- textmodel_wordfish(data_inaugural_dfm, dir = c(johnson_index, reagan_index))
summary(wordfish)
```

```
##
## Call:
## textmodel_wordfish.dfm(x = data_inaugural_dfm, dir = c(johnson_index,
##     reagan_index))
##
## Estimated Document Positions:
##                    theta      se
## 1901-McKinley     1.5231 0.03445
## 1905-Roosevelt    0.5721 0.07408
## 1909-Taft         2.1237 0.01546
## 1913-Wilson       0.9976 0.04989
## 1917-Wilson       0.8953 0.05631
## 1921-Harding      1.3240 0.03016
## 1925-Coolidge     1.4477 0.02699
## 1929-Hoover       1.5517 0.02676
## 1933-Roosevelt    1.1239 0.04450
## 1937-Roosevelt    0.6780 0.05084
## 1941-Roosevelt    0.1642 0.06566
## 1945-Roosevelt   -0.1975 0.10062
## 1949-Truman       0.8955 0.04309
## 1953-Eisenhower   0.3971 0.04596
## 1957-Eisenhower   0.2409 0.05696
## 1961-Kennedy     -0.4300 0.05878
## 1965-Johnson     -0.7405 0.05306
## 1969-Nixon       -0.8995 0.03988
```

```
## 1973-Nixon      -0.4312 0.05319
## 1977-Carter     -0.2595 0.06574
## 1981-Reagan     -0.6129 0.04347
## 1985-Reagan     -0.6120 0.04098
## 1989-Bush       -0.8492 0.03989
## 1993-Clinton    -0.9815 0.04284
## 1997-Clinton    -0.8017 0.04026
## 2001-Bush       -0.7856 0.04875
## 2005-Bush       -0.4165 0.04827
## 2009-Obama      -0.9093 0.03711
## 2013-Obama      -0.9255 0.03917
## 2017-Trump      -1.1277 0.04045
## 2025-Trump      -1.4772 0.02322
## 2021-Biden.txt  -1.4772 0.02322
##
## Estimated Feature Scores:
##      fellow-citizens assembled    4th    march    great anxiety  regard
## beta           2.139   -0.0889  1.345  0.02153 -0.07759  0.6057  0.9199
## psi           -4.906   -2.8417 -4.789 -1.77253  1.51900 -2.8294 -2.3046
##      currency  credit    none  exists     now treasury receipts inadequate
## beta    2.167  0.9628  0.5128 -0.5351 -0.4631    1.528    2.346     0.2772
## psi    -3.972 -2.2206 -2.2387 -2.4414  1.2856   -4.357   -5.658    -2.2953
##         meet current obligations government sufficient   public   needs surplus
## beta -0.4061   1.534      0.4229     0.2488     0.6309  0.42348 -0.1739   1.345
## psi   0.1697  -3.672     -0.9451     1.6480    -1.4662 -0.04104 -0.8127  -4.789
##      instead  deficit    felt constrained convene congress extraordinary
## beta -1.0077 -0.02742  0.4259       1.345   1.345   0.7940        0.7832
## psi  -0.8927 -1.47020 -2.3942      -4.789  -4.789  -0.3057       -3.0024
```

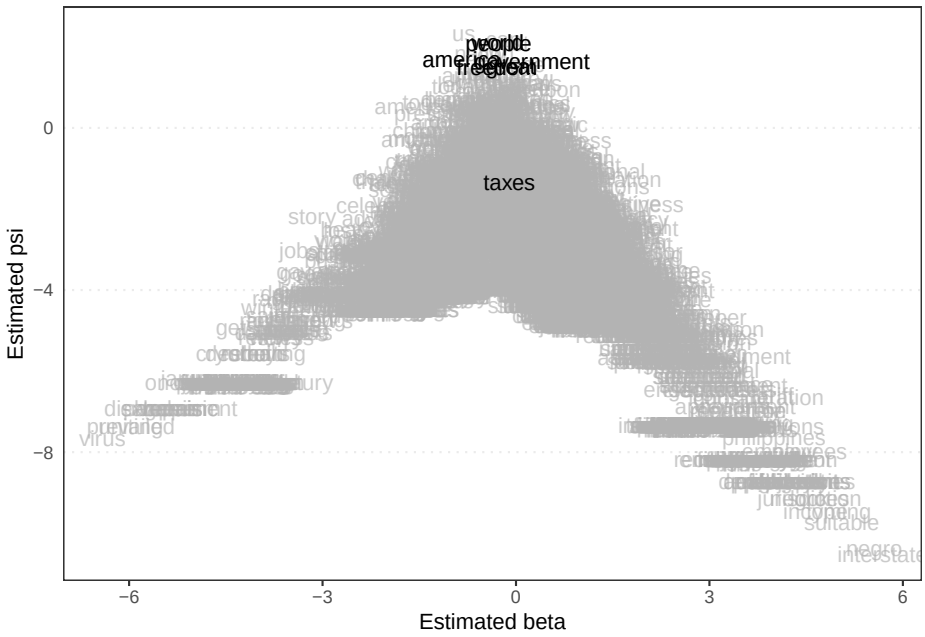
The `summary()` output for a Wordfish model provides information about the model fit and the estimated parameters (θ, α, β, ψ). The θ values are the estimated positions of the documents on the latent dimension. Like Wordscores, we can use the `predict()` function to obtain the estimated document positions with confidence intervals. The estimated position (`theta`) is called the `fit` in the output.

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

Using the `textplot_scale1d()` function, similar to Wordscores, we can visualise the estimated document positions and the word parameters. Plotting the word parameters (`margin = "features"`) shows which words are associated with which end of the latent dimension.

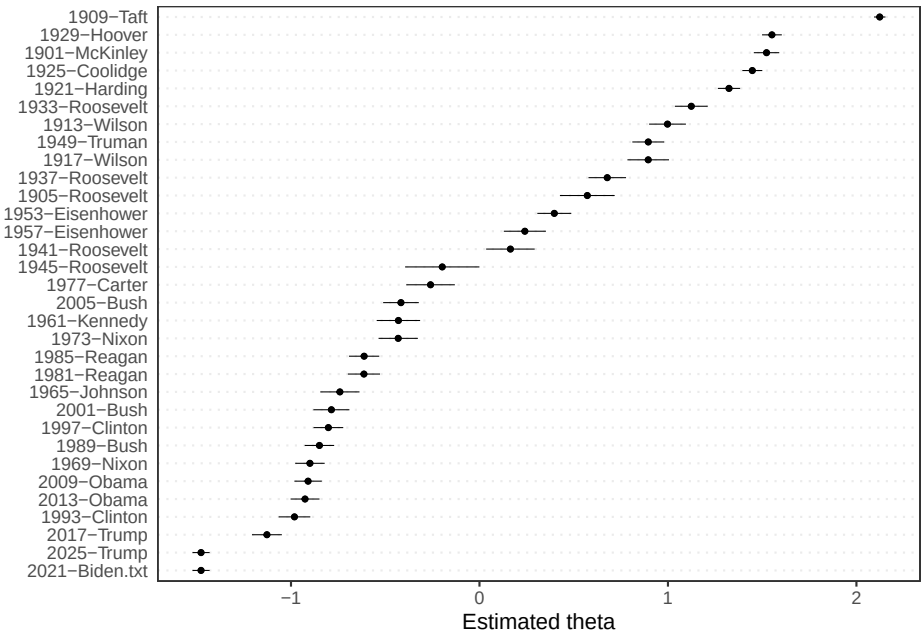
```
library(quanteda.textplots)
textplot_scale1d(wordfish,
                 margin = "features", # Plot features (words)
                 highlighted = c("america", "great", "freedom", "government", "taxes",
```

```
)
```



A word's position on this scale corresponds to its β value. Words at one end of the scale are more likely to appear in documents with high θ values, and words at the other extreme are more likely to appear in documents with low θ values. How we interpret the scale depends on the words we find at the extremes and any anchor texts used for orientation. Plotting document positions (`margin = "documents"`) visualises the estimated values of θ.

```
# Plot the distribution of document positions (theta values) with confidence
# intervals. Theta values are the estimated document scores on the latent
# dimension.
textplot_scale1d(wordfish, margin = "documents"  # Plot documents
)
```



This graph shows the estimated position of each inaugural address on the latent dimension, ordered by year. The confidence intervals indicate the uncertainty of these estimates. Interpreting this dimension requires careful consideration of the anchor texts used and the words that load highly on the dimension (from the word plot). For example, suppose we anchor with a president who is typically considered 'liberal' at one end and 'conservative' at the other, and the word plot shows terms related to social programmes at one end and terms related to individual freedom at the other. In that case, we might interpret this as a left-right political dimension. However, Wordfish can uncover any dominant latent dimension in the text, which may not always conform to preconceived notions such as a simple left-right scale.

Wordfish is a valuable tool for discovering latent dimensions in text data without relying on external scores. Its unsupervised nature can be both a strength (no need for reference data) and a weakness (latent dimension interpretation is not always straightforward and requires careful analysis of word loadings).

6.3 Correspondence Analysis

Correspondence analysis (CA) is a dimensionality reduction technique used to analyse the relationship between categorical variables. In text analysis, it is often applied to document feature matrices (DFM) to explore the associations between documents and words. CA can be considered categorical data (or count data treated as frequencies or proportions) equivalent to Principal Component Analysis (PCA). It simultaneously positions documents and features (words) in

a low-dimensional space, where proximity between points indicates association.

- **Simple Correspondence Analysis (SCA)** is applied to a two-way contingency table (like a dfm).
- **Multiple Correspondence Analysis (MCA)** is an extension used to analyse relationships between more than two categorical variables and is often applied to surveys or questionnaires. While a DFM is inherently a two-way table, MCA can be used by treating the presence or absence of each word in a document as a categorical variable. However, this approach is less common than applying SCA directly to the dfm or using specialised text analysis functions that perform a form of CA.

Here, we look at an example using a dataset where the text has been coded into specific categories, making it suitable for MCA. The example uses data from an article on the stylistic variation in Donald Trump's Twitter data (which we already looked at earlier) between 2009 and 2018 (Clarke & Grieve, 2019). In this study, the authors downloaded 21,739 tweets and grouped them into 63 categories over 5 dimensions based on their content. First, we load the packages we need for the Correspondence Analysis:

Although there is a CA function in `quanteda`, here, we first look at `FactoMineR` and `factoextra`, which allow us several interesting possibilities. `FactoMineR` is renowned for its extensive range of exploratory data analysis functions, including CA and MCA, and it integrates seamlessly with `factoextra` for visualisation. First, we load the necessary packages and the dataset:

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

Then, we import the data, which should contain categorised information about the tweets. As before, you can load the data directly from the URL:

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

This dataset contains several variables, including categorisations of the tweets and possibly other metadata. For the MCA, we must select the columns representing the categorical variables we want to analyse (the 63 categories mentioned in the original text). We will also sample a subset of the tweets for faster processing.

```
tweets <- tweets[sample(nrow(tweets), 500), ]
tweets_mat <- tweets[, 2:65]  # Select columns 2 through 65, which contain the categorical variab
```

We can then run the Multiple Correspondence Analysis using the `MCA()` function from the `FactoMineR` package. When we provide the data frame containing the

categorical variables, we need to specify the number of dimensions (`ncp`) to retain. We can determine this by examining the eigenvalues (similar to a scree plot in PCA) or base ourselves on theoretical considerations. Here, we use 5, as this was the number of dimensions found by Clarke & Grieve (2019). In addition, we can include supplementary quantitative variables (`quanti.sup`) or supplementary individuals (`ind.sup`) that do not contribute to the MCA but are projected onto the resulting dimensions. Here, we include 'Tweet.Length' as a supplementary quantitative variable:

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

First, let's look at the association of the supplementary quantitative variable (Tweet Length) with the five dimensions. The MCA output's `quanti.sup` element shows the correlation between the supplementary variable and each dimension.

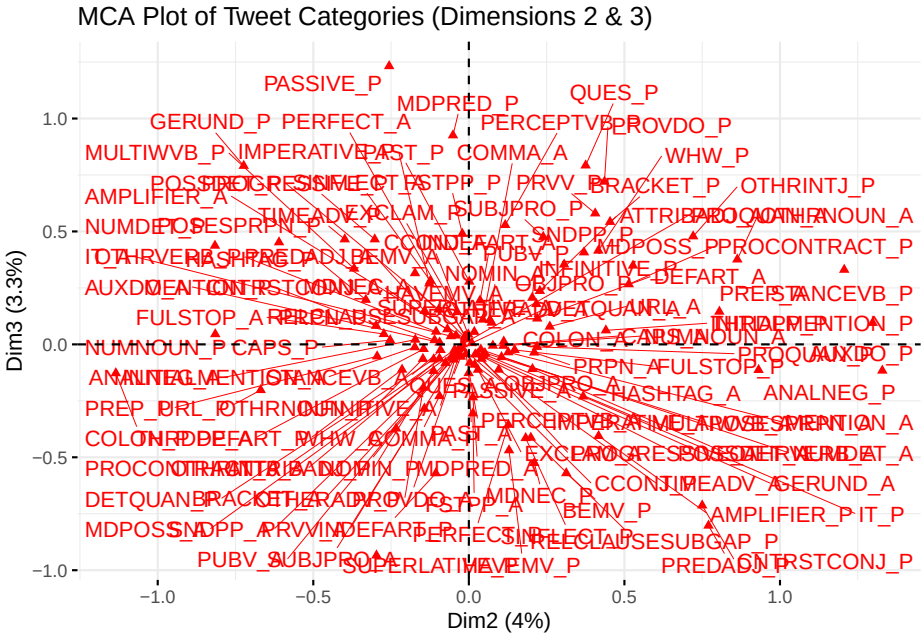
```
mca_tweets$quanti.sup  # Display the correlations between the supplementary quantitati
```

```
## $coord
##               Dim 1      Dim 2       Dim 3      Dim 4      Dim 5
## WORDCOUNT 0.8570116 -0.1467442 -0.05012065 0.01714113 -0.1163162
```

As we can see, the 'Tweet.Length' variable strongly correlates with dimension 1. This means that the first dimension primarily captures the variation in the length of tweets rather than different stylistic categories. When interpreting the correspondence between categories and dimensions, it's often advisable to focus on the dimensions that explain the variance in the categorical variables independently of the ancillary variables, such as length. We will, therefore, concentrate our visualisation on dimensions 2 and 3, as we did in the original analysis.

We can visualise the position of the categories on the chosen dimensions using `fviz_mca_var()` from the `factoextra` package. This plot shows the categories in MCA space. Closely related categories are often related, and their position relative to the dimensions indicates their contribution to those dimensions.

```
fviz_mca_var(mca_tweets,
             repel = TRUE, # Avoid overlapping text labels
             geom = c("point", "text"), # Display both points and text labels
             axes = c(2, 3), # Specify that we want to plot Dimensions 2 and 3
             ggtheme = theme_minimal(),
             title = "MCA Plot of Tweet Categories (Dimensions 2 & 3)")
```



This plot illustrates the relationships between the tweet categories based on Dimensions 2 and 3. Clarke & Grieve (2019) interpreted Dimension 2 as a 'Conversational Style' and Dimension 3 as a 'Campaigning Style'. Thus, categories at the extremes of Dimension 2 indicate conversational style, while those at the extremes of Dimension 3 are associated with campaigning style. Categories near the centre are less distinctive along these dimensions.

To see which categories contribute most to these dimensions or have the most extreme positions, we can examine their coordinates on the dimensions. The `get_mca_var()` function extracts detailed information about the variables (categories), including their coordinates, contributions to the dimensions, and correlations.

```
var_info <- get_mca_var(mca_tweets)  # Get detailed information about the categories (variables)
coordinates <- as.data.frame(var_info$coord)  # Extract the coordinates of the categories on each
head(coordinates)
```

```
##                    Dim 1       Dim 2       Dim 3       Dim 4       Dim 5
## AMPLIFIER_A -0.09788751 -0.06902274  0.06712121  0.09543741 -0.03446247
## AMPLIFIER_P  0.59146114  0.41705287 -0.40556335 -0.57665701  0.20823099
## ANALNEG_A   -0.15748770 -0.17213448  0.02091087 -0.09791397 -0.04902178
## ANALNEG_P    0.85204886  0.93129168 -0.11313318  0.52973968  0.26522042
## ATTRIBADJ_A -0.47929022  0.22824686  0.23663061  0.13986682  0.40121761
## ATTRIBADJ_P  0.23821011 -0.11344006 -0.11760683 -0.06951464 -0.19940755
```

```
# Optionally, order by a specific dimension to see the extremes. This can help
# interpret the dimensions by identifying the categories most strongly
```

```
# associated with each end.

coordinates_ordered_dim2 <- coordinates %>%
    arrange(`Dim 2`)

head(coordinates_ordered_dim2)  # Display categories with the most negative coordinate
```

```
##                  Dim 1      Dim 2       Dim 3       Dim 4      Dim 5
## COLON_P     -0.2592962 -1.1346550 -0.12765743  0.65087583  0.6419737
## NUMDET_P     0.3617208 -0.8159953  0.43697966 -0.03973954  0.1617759
## NUMNOUN_P    0.2506834 -0.8150235  0.04534697 -0.38332211  0.3973838
## GERUND_P     0.2383111 -0.7235272  0.79047745 -0.04533401 -0.3780261
## URL_P       -0.5334139 -0.6689921 -0.20223454  0.29823581  0.6217575
## POSESPRPN_P  0.1372452 -0.6100926  0.45200582  0.16700550  0.6420377
```

```
tail(coordinates_ordered_dim2)  # Display categories with the most positive coordinate
```

```
##                       Dim 1     Dim 2       Dim 3       Dim 4       Dim 5
## PREP_A           -0.8230712 0.8052457  0.14477086 -0.03477667  0.03766616
## OTHRNOUN_A       -1.4568686 0.8629022  0.37599003 -0.41308773  0.09841326
## ANALNEG_P         0.8520489 0.9312917 -0.11313318  0.52973968  0.26522042
## STANCEVB_P        0.5514010 1.2065100  0.33043865  0.49040727  0.77609391
## INITIALMENTION_P -1.3740889 1.2977164  0.09504675 -0.65336128 -0.36025425
## AUXDO_P           0.7401893 1.3289167 -0.11606038  1.11590177  0.30943522
```

```
coordinates_ordered_dim3 <- coordinates %>%
    arrange(`Dim 3`)

head(coordinates_ordered_dim3)
```

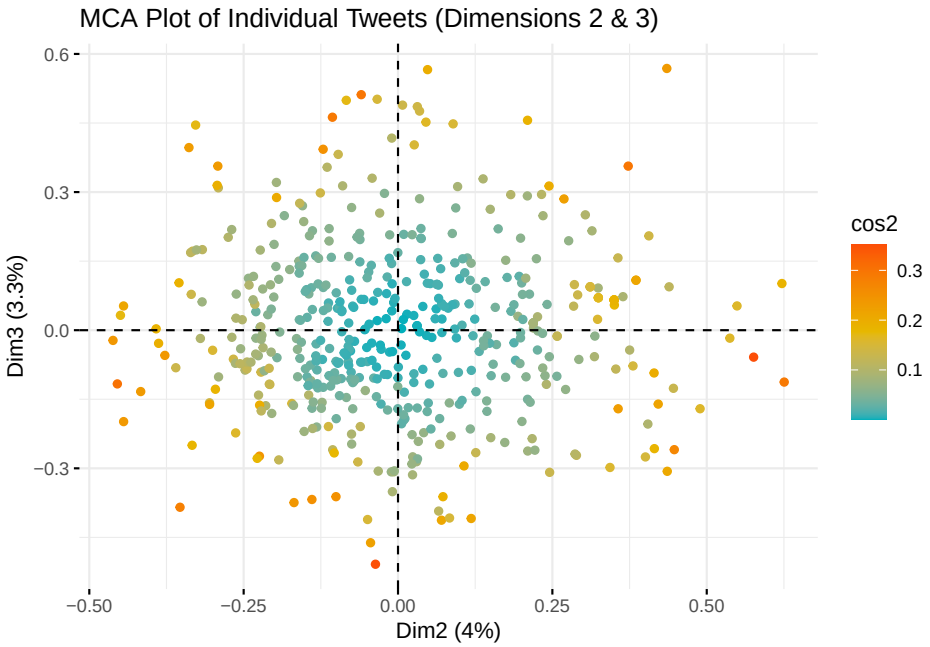
```
##                   Dim 1      Dim 2      Dim 3       Dim 4       Dim 5
## SUPERLATIVE_P 0.3844324 -0.2969177 -0.9344720  0.37897964 -1.12086446
## PREDADJ_P     0.5907107  0.7702513 -0.8023569 -0.66156655  0.32914762
## CNTRSTCONJ_P  0.4785936  0.7500073 -0.7131886 -0.58944621  0.45511677
## INDEFART_P    0.4894847 -0.1060032 -0.5706842 -0.05053374  0.20323712
## BEMV_P        0.5529525  0.3132293 -0.5702540 -0.34951629  0.06513688
## MDNEC_P       0.5222506  0.2103261 -0.5253531  0.61481327  0.08689422
```

```
tail(coordinates_ordered_dim3)
```

```
##                Dim 1       Dim 2     Dim 3       Dim 4       Dim 5
## PRVV_P    0.6250762  0.40612075 0.5799258  0.79378561 -0.07698886
## PROVDO_P  1.0370597  0.43591147 0.7204183 -0.03205591 -0.66174907
## GERUND_P  0.2383111 -0.72352720 0.7904774 -0.04533401 -0.37802611
## QUES_P    0.3404573  0.37539603 0.7922378  1.57205964  0.26088244
## MDPRED_P  0.4722946 -0.05099973 0.9256090 -0.68949403 -0.08883854
## PASSIVE_P 0.6729391 -0.25613607 1.2319758 -1.15578164 -0.16784366
```

Examining the categories with the highest absolute coordinates on Dimensions 2 and 3 provides insight into the characteristics of these stylistic dimensions. For instance, if categories such as 'Use of colloquialisms' or 'Personal anecdotes' have high positive coordinates on Dimension 2, this lends weight to interpreting this dimension as 'Conversational Style'. Similarly, if categories such as 'Policy mentions' or 'Calls to action' have high positive coordinates on Dimension 3, this would support the interpretation of this dimension as 'Campaigning Style'. MCA also enables individuals (or, in this case, tweets) to be plotted in the same space using the `fviz_mca_ind()` function, which can reveal clusters of tweets with similar stylistic features:

```
# Plot the position of individual tweets on Dimensions 2 and 3.
fviz_mca_ind(mca_tweets,
             geom = "point", # Show points for each tweet
             axes = c(2, 3), # Plot Dimensions 2 and 3
             col.ind = "cos2", gradient.cols = c("#00AFBB", "#E7B800", "#FC4E07"), # Color indivi
             repel = FALSE,
             ggtheme = theme_minimal(),
             title = "MCA Plot of Individual Tweets (Dimensions 2 & 3)")
```



Interpreting MCA results is often an iterative and exploratory process that involves examining the position of categories and individuals and potentially using additional variables or external information to understand the meaning of the dimensions. The designers of `FactoMineR` have developed a Shiny app called `Factoshiny` that provides an interactive interface for exploring MCA results that can be very helpful in this process.

```
library(Factoshiny)

# Launch the interactive Shiny app for MCA results. This provides a graphical
# user interface to explore the MCA output interactively

res.shiny <- MCAshiny(mca_tweets)  # Use the MCA output object as input.
```

Ensure you quit the Shiny application by clicking the "Quit the App" button to return to your R session. For more information on Correspondence Analysis and the `FactoMineR` package, see the original article by Lê et al. (2008) or the package website.

While `FactoMineR` is excellent for MCA with pre-coded categorical data, the `quanteda.textmodels` package also provides a function, `textmodel_ca()`, designed explicitly for Simple Correspondence Analysis (SCA) on a document-feature matrix (dfm). This is particularly useful when you want to perform CA directly on raw term frequencies in your corpus without prior categorisation. Based on their co-occurrence patterns, SCA on a dfm positions both documents and terms in a low-dimensional space.

Let's apply `textmodel_ca()` to the UK party manifestos dfm we created earlier in the Wordscores section. We will fit an SCA model and then explore the resulting document and feature placements. First, ensure you have loaded the `quanteda.textmodels` package. Then, we can apply the `textmodel_ca()` function to our `data_manifestos_dfm`, for which we must specify the number of dimensions (`nd`) to compute:

```
# Ensure quanteda.textmodels is loaded
library(quanteda.textmodels)

# We will compute the first 2 dimensions
ca_model <- textmodel_ca(data_manifestos_dfm, nd = 2)
ca_model$sv
```

```
## [1] 0.5276873 0.4724757
```

The output for `textmodel_ca` shows the eigenvalues for the first two dimensions, indicating the amount of variance explained by each dimension. The `ca_model` object also provides the coordinates of both features (terms) and documents on the computed dimensions. Features or documents with similar coordinates are located close together in the CA space, suggesting a strong association based on their co-occurrence patterns. We can access the coordinates of the documents and features directly from the model object for plotting. The document coordinates are stored in `ca_model$rowcoord`, and the feature coordinates are in `ca_model$colcoord`:

```
doc_coords <- as.data.frame(ca_model$rowcoord)
doc_coords$document <- rownames(doc_coords)
```

```
head(doc_coords)
```

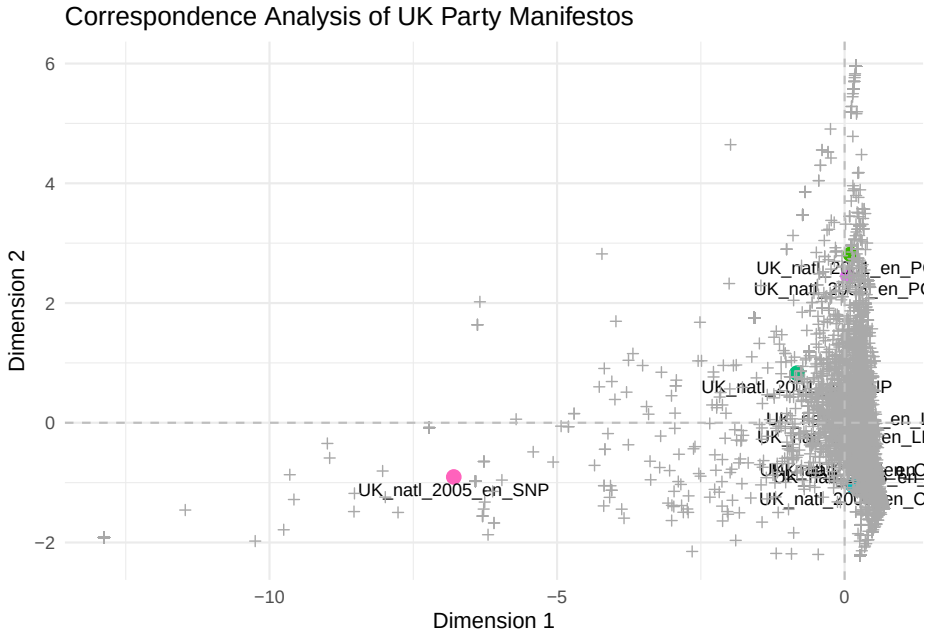
```
##                            Dim1       Dim2            document
## UK_natl_2001_en_Con   0.1524448 -0.5672257 UK_natl_2001_en_Con
## UK_natl_2001_en_Lab   0.3302092 -0.5700563 UK_natl_2001_en_Lab
## UK_natl_2001_en_LD    0.1771004  0.2938892  UK_natl_2001_en_LD
## UK_natl_2001_en_PCy   0.1031924  2.8146966 UK_natl_2001_en_PCy
## UK_natl_2001_en_SNP -0.8278683  0.8287998 UK_natl_2001_en_SNP
## UK_natl_2005_en_Con   0.1432759 -1.0469080 UK_natl_2005_en_Con
```

```
# Extract feature coordinates

feature_coords <- as.data.frame(ca_model$colcoord)
feature_coords$feature <- rownames(feature_coords)
head(feature_coords)
```

```
##                     Dim1       Dim2      feature
## time           0.1959597 -0.3493652         time
## common         0.2421251 -0.6085238       common
## sense          0.3245334 -1.0271480        sense
## conservative   0.2831843 -1.3381374 conservative
## manifesto      0.2763475 -0.7723784    manifesto
## introduction -0.4311302  1.4987321 introduction
```

Now that we have extracted the coordinates, we can visualise the documents and features in the CA space using `ggplot2`. Plotting both documents and features on the same plot enables us to inspect their relationships directly. Terms located near a document are likely to appear frequently compared to other documents, and documents located near each other tend to use similar vocabulary.

```
library(ggplot2)

ggplot() +
  geom_point(data = doc_coords, aes(x = Dim1, y = Dim2, color = document), size = 3) +
  geom_text(data = doc_coords, aes(x = Dim1, y = Dim2, label = document), vjust = 1.5, size = 3)
  geom_point(data = feature_coords, aes(x = Dim1, y = Dim2), shape = 3, color = "darkgrey") +
geom_vline(xintercept = 0, linetype = "dashed", color = "grey") + # Add vertical line at x=0
  geom_hline(yintercept = 0, linetype = "dashed", color = "grey") + # Add horizontal line at y=0
  ggtitle("Correspondence Analysis of UK Party Manifestos") +
scale_x_continuous(name = "Dimension 1") +
scale_y_continuous(name = "Dimension 2") +
  theme_minimal() +
  theme(legend.position = "none")
```



This plot illustrates the relative positions of UK party manifestos and terms extracted from the first two dimensions by SCA. Documents positioned closer together use similar vocabulary, and terms positioned closer to documents are more characteristic of them. The interpretation of the dimensions (e.g. as a left-right scale or another thematic contrast) depends on examining which documents and terms fall at the extremes of each dimension. For instance, if manifestos from left-leaning parties and terms related to social welfare appear at one end of Dimension 1 and manifestos from right-leaning parties and terms related to the economy or security appear at the other, then Dimension 1 probably represents a left-right political spectrum.

6.4 Exercises

1. Apply the Wordscores method to the inaugural speeches corpus (data_corpus_inaugural). Select a set of reference speeches and assign them hypothetical scores based on a dimension of interest (e.g. populism, as defined by external knowledge). Estimate the scores for the remaining speeches and visualise the results. Discuss the estimated positions and the uncertainty around them.

2. Apply the Wordfish method to the corpus of UK party manifestos used in the Wordscores example. Select two manifestos to define the direction of the scale. Interpret the resulting dimension based on the words that load highly on it and the manifestos' positions. Compare the Wordfish results with the Wordscores results.

3. Explore another dataset with categorical text annotations or create a categorised dataset from a text corpus (for example, by coding themes in a small set of documents). Perform Multiple Correspondence Analysis on this data. Interpret the main dimensions based on the categories that contribute most to them. Visualise the categories and/or individuals in the MCA space.

4. Apply Simple Correspondence Analysis to a document-feature matrix of your choice using the `textmodel_ca()` function. Interpret the first two dimensions based on the words and documents located at the extremes. Visualise the results and discuss the relationships revealed by the plot.

5. Use the `textmodel_affinity()` function to compute the affinity matrix for a corpus. Explore the matrix to identify documents with a high affinity to specific documents of interest. Apply a clustering method (e.g. hierarchical clustering) to the affinity matrix and interpret the resulting clusters.

6. Research other scaling methods for text analysis, such as ideal point models or specialised techniques for specific text data types. How do they differ from Wordscores, Wordfish and Correspondence Analysis in terms of their assumptions and applications?

Chapter 7

Supervised Methods

Supervised learning methods use this pre-labelled data to recognise patterns. Think of it like a student learning from examples with answers provided by a teacher. Labelled data, also known as the training set, consists of a collection of texts, each of which has a known category or value assigned to it. Examples include movie reviews labelled as 'positive' or 'negative', news articles categorised by topic (e.g. 'sports', 'politics', 'finance'), and survey responses assigned a satisfaction score. A supervised learning algorithm examines the features of these labelled texts — typically, which words appear and how often — and tries to determine the relationship between the words and the labels. For instance, it may recognise that reviews containing words such as 'amazing', 'love' and 'excellent' are often categorised as 'positive'. Once the algorithm has learned these patterns from the training set, it can predict the labels of new, unlabelled texts — usually called the test set.

Supervised methods are effective when manual labelling of a subset of data is feasible, but automatic classification of larger volumes of text is required. Supervised methods differ from dictionary methods, which rely on fixed lists of words, and unsupervised methods, such as topic modelling or clustering, which discover patterns without needing pre-assigned labels.

Several algorithms for supervised text classification are available within the R ecosystem, particularly those integrated with `quanteda`. We will cover two widely used, relatively simple yet effective algorithms: Support Vector Machines (SVM) and Naive Bayes (NB). SVM is a powerful discriminative classifier that identifies an optimal boundary, or hyperplane, to distinguish between different categories in a high-dimensional feature space. Naive Bayes is a probabilistic classifier based on Bayes' theorem, which simplifies the assumption that features are independent of each other.

We will also explore methods for visualising model performance and implementing cross-validation using the `caret` package, which provides a unified interface

for many machine-learning algorithms and validation techniques.

7.1 Support Vector Machines (SVM)

SVMs are powerful supervised learning models that are frequently employed for classification tasks. In text classification, for example, they operate by mapping features of documents, such as word counts, from a document-feature matrix (DFM) into a high-dimensional space. In this space, SVMs identify the optimal hyperplane that most effectively separates documents into different categories.

Imagine you have a piece of paper with red and blue dots scattered on it, and you want to draw a straight line that best separates them. An SVM tries to do something similar but in many more dimensions. Each document can be considered a point in a very high-dimensional space. Each dimension corresponds to a unique word in your vocabulary, and the position of the document along that dimension depends on how frequently that word appears in the document or on other measures, such as TF-IDF. An SVM aims to find the optimal hyperplane that best separates documents into different categories, such as "positive review" versus "negative review". 'Best' means the hyperplane with the largest possible margin — the broadest possible gap — between the closest documents of the different classes. Documents closest to this boundary, which defines the margin, are called 'support vectors'. SVMs are known for their effectiveness, especially when dealing with many features, which is common in text data since each word can constitute a feature.

In this example, we will use the `caret` package to train an SVM model for binary polarity classification, use `quanteda` for text preprocessing and the binary polarity label ("neg" or "pos") provided by a sample of the Large Movie Review Dataset (`data_corpus_LMRD`) as the target variable:

```
set.seed(42)

library(quanteda)
library(quanteda.classifiers)
library(caret)  # For model training, evaluation, and cross-validation
library(ggplot2)
library(pROC)  # For ROC analysis

corpus_reviews <- corpus_sample(data_corpus_LMRD, 2000)  # Sample 2000 reviews
```

We need to split our data into separate training and test sets for supervised learning: we will use the training set to build the model and the test set to evaluate its performance on unseen data later on. We will use the `polarity` variable as our target, ensuring it is a factor as this is the required format for classification tasks in `caret`. Note that we will filter for documents with non-missing polarity labels before splitting. `caret` requires factor levels to be valid R variable names (like "neg" and "pos"):

```r
polarity_labels <- factor(corpus_reviews$polarity)

# Identify documents with valid polarity labels

valid_docs_index <- which(!is.na(polarity_labels))
corpus_reviews_valid <- corpus_reviews[valid_docs_index]
polarity_valid <- polarity_labels[valid_docs_index]
polarity_valid <- factor(polarity_valid, levels = c("neg", "pos"))

# Check the distribution of polarity_valid BEFORE splitting
print(table(polarity_valid))
```

```
## polarity_valid
##  neg  pos
## 1023  977
```

```r
# Create stratified split indices to ensure both classes are included in
# train/test sets
neg_indices <- which(polarity_valid == "neg")
pos_indices <- which(polarity_valid == "pos")

# Determine the number of instances for train/test per class (70/30 split)
set.seed(42)
train_size_neg <- floor(0.7 * length(neg_indices))
train_size_pos <- floor(0.7 * length(pos_indices))

# Sample indices for training set from each class
train_indices_neg <- sample(neg_indices, size = train_size_neg, replace = FALSE)
train_indices_pos <- sample(pos_indices, size = train_size_pos, replace = FALSE)

# Combine training indices
train_index <- c(train_indices_neg, train_indices_pos)

# The remaining indices are for the test set
all_valid_indices <- seq_along(polarity_valid)
test_index <- all_valid_indices[!all_valid_indices %in% train_index]

# Split the corpus subset and polarity labels into training and testing sets
# using the determined indices
corpus_reviews_train <- corpus_reviews_valid[train_index]
corpus_reviews_test <- corpus_reviews_valid[test_index]

polarity_train <- polarity_valid[train_index]
polarity_test <- polarity_valid[test_index]

# Check the distribution of the split
```

```
print("Training set class distribution:")
```

```
## [1] "Training set class distribution:"
```

```
print(table(polarity_train))
```

```
## polarity_train
## neg pos
## 716 683
```

```
print("Testing set class distribution:")
```

```
## [1] "Testing set class distribution:"
```

```
print(table(polarity_test))
```

```
## polarity_test
## neg pos
## 307 294
```

Next, we preprocess the training and test corpus subsets to generate the DFMs. Here, it is crucial that the test DFM has the same features and order as the training DFM. The `dfm_match()` function ensures this. We will then apply common text cleaning steps during tokenisation and convert the DFMs to matrices to ensure compatibility with `caret`.

```
# Tokenise and preprocess the training corpus

tokens_train <- tokens(corpus_reviews_train, what = "word", remove_punct = TRUE,
    remove_symbols = TRUE, remove_numbers = TRUE, remove_url = TRUE, remove_separators
    tokens_tolower() %>%
    tokens_select(stopwords("english"), selection = "remove")

# Tokenise and preprocess the test corpus

tokens_test <- tokens(corpus_reviews_test, what = "word", remove_punct = TRUE, remove_s
    remove_numbers = TRUE, remove_url = TRUE, remove_separators = TRUE) %>%
    tokens_tolower() %>%
    tokens_select(stopwords("english"), selection = "remove")

# Create DFMs from the processed tokens

dfm_train <- dfm(tokens_train)
dfm_test <- dfm(tokens_test)

# Ensure the test DFM has the same features as the training DFM

dfm_test_matched <- dfm_match(dfm_test, features = featnames(dfm_train))
```

```
# Convert DFMs to matrices for caret

matrix_train <- as.matrix(dfm_train)
matrix_test <- as.matrix(dfm_test_matched)

# Remove any zero-variance columns from the training matrix, as caret::train
# can have issues Ensure the test matrix also has these columns removed

nzv_train <- nearZeroVar(matrix_train)
if (length(nzv_train) > 0) {
    matrix_train <- matrix_train[, -nzv_train]
    matrix_test <- matrix_test[, -nzv_train]
}
```

Now we train the SVM model using `caret::train()`. `caret::train()` provides a consistent interface for training various models. We specify `method = "svmLinear"` for a linear kernel SVM, which is often a good starting point for text data.

Rather than using a single train/test split, we can use k-fold cross-validation to obtain a more reliable estimate of the model's performance. The `caret::train()` function makes this easy using the `trControl` argument. To demonstrate this, we will define a 10-fold cross-validation set-up. For binary classification, the `twoClassSummary` function calculates metrics such as accuracy, kappa, sensitivity, specificity, and ROC AUC.

What is k-fold cross-validation? Put simply, it involves splitting the training data into equal-sized parts, known as 'folds'. For instance, if k equals 10, the training data would be divided into ten folds, and the model would be trained ten times. During each iteration, one fold is held out as a validation set and the model is trained using the remaining 9 folds. The trained model is then tested on the held-out validation set, and the performance metrics are recorded. After ten iterations, the performance metrics from each fold are averaged to provide a more reliable estimate of how the model will perform on new data.

The 'caret' package makes cross-validation easy. We use the `trainControl()` function to specify the cross-validation settings and the `train()` function to train the model. We select a linear SVM using the argument `method = "svmLinear"`. Linear SVMs are often very effective and computationally less intensive than non-linear ones for text. Setting `metric = "ROC"` instructs caret to optimise the model based on the area under the ROC curve (AUC), a typical and effective metric for binary classification. Setting `classProbs = TRUE` means that we require the model to output class probabilities for ROC analysis and calibration plots, such as the probability that a review is 'positive'. `summaryFunction = twoClassSummary` is used for binary classification problems and calculates useful metrics such as sensitivity, specificity, and ROC

AUC during cross-validation.

```
train_control_cv_clf <- trainControl(
  method = "cv",           # Use cross-validation
  number = 10,             # Number of folds
  savePredictions = "final", # Save predictions for the final model
  classProbs = TRUE,     # Compute class probabilities (needed for ROC/Calibration)
  summaryFunction = twoClassSummary # Use metrics suitable for binary classification
)

# Train the SVM model using caret::train with cross-validation
# We use svmLinear as the method
# metric = "ROC" tells caret to optimise based on AUC
# Ensure the factor levels for y (polarity_train) are valid R names ("neg", "pos")

svm_model_caret_cv <- train(
  x = matrix_train,       # Training feature matrix
  y = polarity_train,      # Training response vector (factor: "neg", "pos")
  method = "svmLinear", # Use linear kernel SVM
  trControl = train_control_cv_clf, # Apply cross-validation control for classificatio
  metric = "ROC"          # Optimize based on ROC AUC
)

print(svm_model_caret_cv)
```

```
## Support Vector Machines with Linear Kernel
##
## 1399 samples
##  245 predictor
##    2 classes: 'neg', 'pos'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 1260, 1258, 1258, 1260, 1260, 1259, ...
## Resampling results:
##
##   ROC        Sens       Spec
##   0.7917106  0.7347418  0.7174979
##
## Tuning parameter 'C' was held constant at a value of 1
```

The output from `print(svm_model_caret_cv)` now includes the average classification performance metrics, like Accuracy, Kappa, and ROC AUC, across the 10 folds. This provides a more accurate prediction of how the model will perform on unseen data than a single split would. We then use the trained `svm_model_caret_cv` to predict the polarity labels and probabilities for the documents in the test set.

```r
# Predict the labels for the test set
svm_predict_labels_caret_cv <- predict(svm_model_caret_cv, newdata = matrix_test)

# Predict probabilities for the test set (needed for ROC and Calibration)
svm_predict_probs_caret_cv <- predict(svm_model_caret_cv, newdata = matrix_test,
    type = "prob")

# Display the head of the predicted labels and probabilities
head(svm_predict_labels_caret_cv)
```

```
## [1] pos neg pos neg neg pos
## Levels: neg pos
```

```r
head(svm_predict_probs_caret_cv)
```

```
##         neg       pos
## 1 0.4393709 0.5606291
## 2 0.5076856 0.4923144
## 3 0.4927485 0.5072515
## 4 0.5997361 0.4002639
## 5 0.6494433 0.3505567
## 6 0.4764429 0.5235571
```

Furthermore, we can examine the classification performance using `caret::confusionMatrix()` to compare the predicted and actual labels for the test set. This Confusion Matrix is a fundamental tool for evaluating classification models. It is a table that summarises performance by showing:

- **True Positives (TP)**: instances that were correctly predicted as positive
- **True Negatives (TN)**: instances that were correctly predicted as negative
- **False Positives (FP)**: instances that were incorrectly predicted as positive
- **False Negatives (FN)**: instances that were incorrectly predicted as negative

From these counts, various metrics are derived:

- **Accuracy: (TP + TN) / Total**, representing overall correctness. However, it can be misleading if the classes are imbalanced;
- **Sensitivity: TP/(TP+FN)**, indicating how well the model identifies actual positives;
- **Specificity: TN/(TN+FP)**, showing how well the model identifies actual negatives;
- **Precision: TP/(TP+FP)**, showing the proportion of predicted positives that were actually positive;
- **F1-score: 2 * (Precision * Recall) / (Precision + Recall)**, the harmonic mean of precision and recall, is useful when both are important.

```r
# Ensure the reference (polarity_test) has the same valid levels as the predicted data

confusion_matrix_caret_cv <- confusionMatrix(
  data = svm_predict_labels_caret_cv, # Predicted labels (factor)
  reference = polarity_test            # Actual labels (reference factor)
)

# Print the confusion matrix and performance statistics
print(confusion_matrix_caret_cv)
```

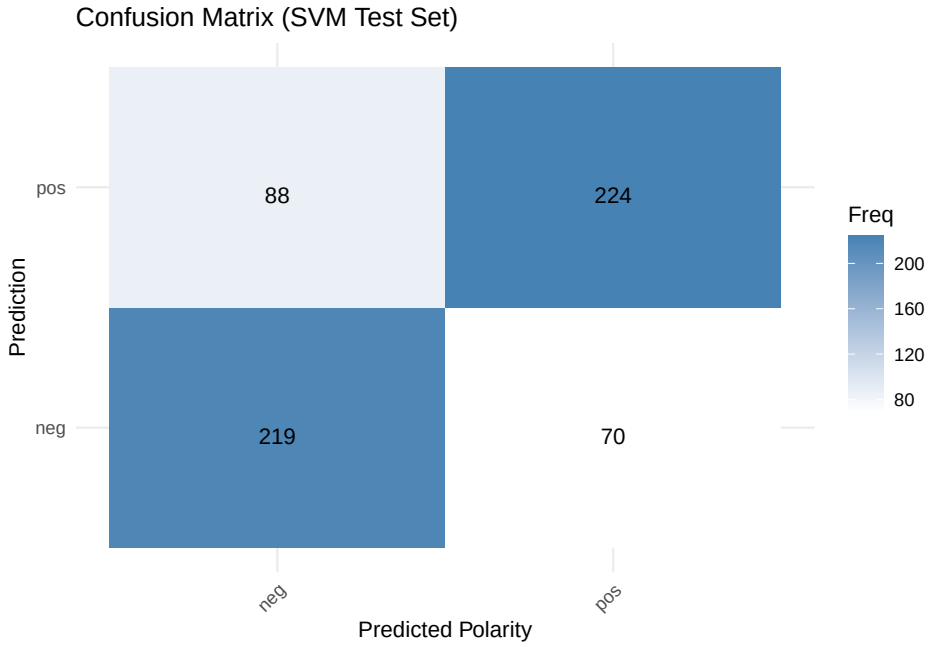
```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction neg pos
##        neg 219  70
##        pos  88 224
##
##                Accuracy : 0.7371
##                  95% CI : (0.7, 0.7719)
##     No Information Rate : 0.5108
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.4746
##
##  Mcnemar's Test P-Value : 0.1762
##
##             Sensitivity : 0.7134
##             Specificity : 0.7619
##          Pos Pred Value : 0.7578
##          Neg Pred Value : 0.7179
##              Prevalence : 0.5108
##          Detection Rate : 0.3644
##    Detection Prevalence : 0.4809
##       Balanced Accuracy : 0.7376
##
##        'Positive' Class : neg
##
```

The confusion matrix for the test set — not used during the cross-validation training process — provides a final evaluation of how well the model performs on data that has never been seen before. It shows the number of true positives, false positives and false negatives, from which various metrics can be derived. Finally, we can visualize all this:

```r
cm_table <- as.data.frame(confusion_matrix_caret_cv$table)
```

```
ggplot(data = cm_table,
       aes(x = Reference, y = Prediction, fill = Freq)) +
  geom_tile() +
  geom_text(aes(label = Freq), vjust = 1, color = "black") + # Add text labels for counts
  scale_fill_gradient(low = "white", high = "steelblue") + # Colour scale
  scale_x_discrete(name = "Actual Polarity") +
  scale_x_discrete(name = "Predicted Polarity") +
  ggtitle("Confusion Matrix (SVM Test Set)") +
  theme_minimal() + # Use a minimal theme
  theme(axis.text.x = element_text(angle = 45, hjust = 1)) # Rotate x-axis labels if needed
```

```
## Scale for x is already present.
## Adding another scale for x, which will replace the existing scale.
```



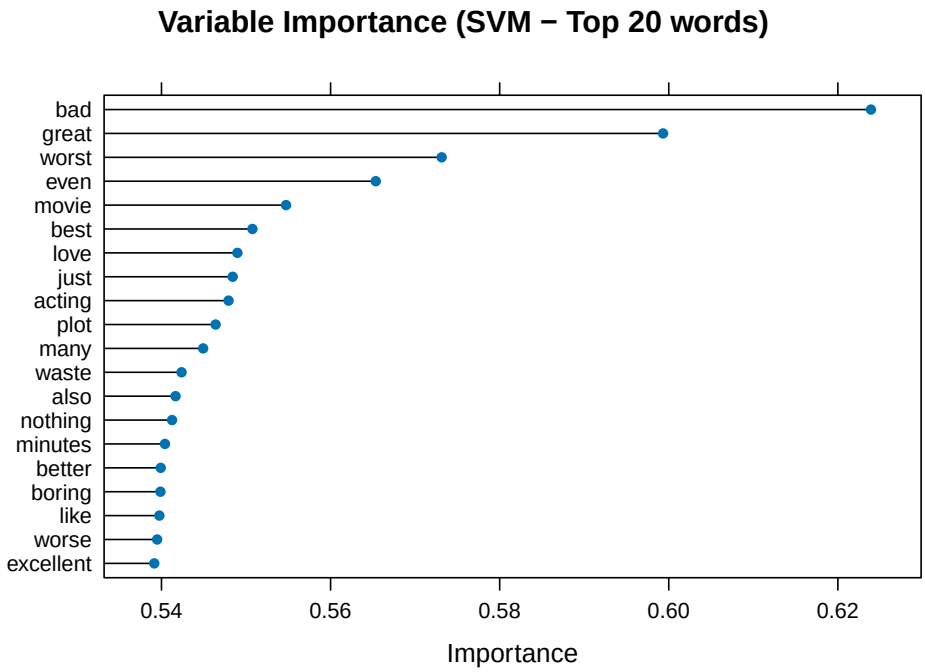
In classification using linear SVMs, variable importance is usually related to the size of the model coefficients, showing which features contribute most to class separation. The `caret::varImp()` function can extract this information:

```
var_importance_svm <- varImp(svm_model_caret_cv, scale = FALSE)

# Print the variable importance
print(var_importance_svm)
```

```
## ROC curve variable importance
##
##   only 20 most important variables shown (out of 245)
```

```
##
##           Importance
## bad           0.6239
## great         0.5993
## worst         0.5731
## even          0.5653
## movie         0.5547
## best          0.5508
## love          0.5490
## just          0.5484
## acting        0.5479
## plot          0.5464
## many          0.5449
## waste         0.5424
## also          0.5417
## nothing       0.5413
## minutes       0.5404
## better        0.5399
## boring        0.5399
## like          0.5398
## worse         0.5395
## excellent     0.5391
```

```
# Plot the top 20 most important variables
plot(var_importance_svm, top = 20, main = "Variable Importance (SVM - Top 20 words)")
```

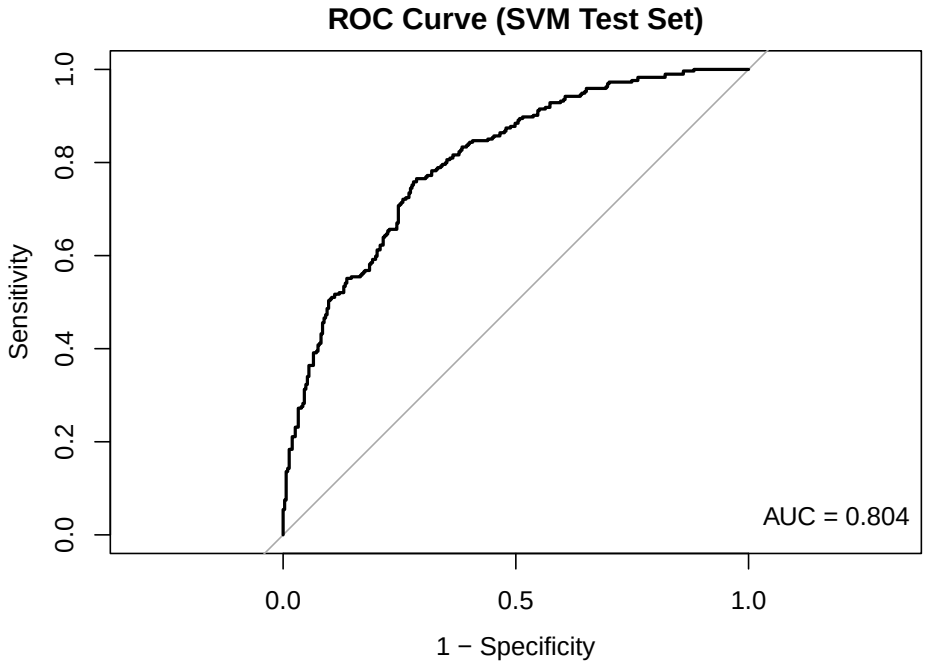


This graph illustrates the words that greatly influenced the SVM model's polarity classification decisions. Words with high importance may have large positive or negative coefficients, which push documents towards the 'pos' or 'neg' class boundary.

Next, we plot the Receiver Operating Characteristic (ROC) curve using the predicted probabilities from the test set and the actual test labels and calculate the Area Under the Curve (AUC). The ROC curve illustrates the trade-off between sensitivity and specificity as the classification threshold varies.

The ROC curve is another common way to evaluate binary classifiers. It plots the true positive rate (sensitivity) against the false positive rate (1 - specificity) at various classification threshold settings. A model that performs no better than random guessing would have a ROC curve close to the diagonal line (from bottom-left to top-right). A good model will have an ROC curve that bows towards the top-left corner. The area under the curve (AUC) summarises the ROC curve into a single number. An AUC of 1 indicates a perfect classifier; an AUC of 0.5 indicates performance no better than random chance; and an AUC of less than 0.5 suggests performance worse than random chance, indicating something is likely wrong, such as the flipped labels.

```
# The roc() function needs the actual responses (factor) and the predicted
# probabilities for the positive class (numeric)
roc_obj_svm <- roc(response = polarity_test, predictor = svm_predict_probs_caret_cv[,
    "pos"], levels = levels(polarity_test))
plot(roc_obj_svm, main = "ROC Curve (SVM Test Set)", legacy.axes = TRUE)

auc_value_svm <- auc(roc_obj_svm)  # Add AUC value to the plot
legend("bottomright", legend = paste("AUC =", round(auc_value_svm, 3)), bty = "n")
```

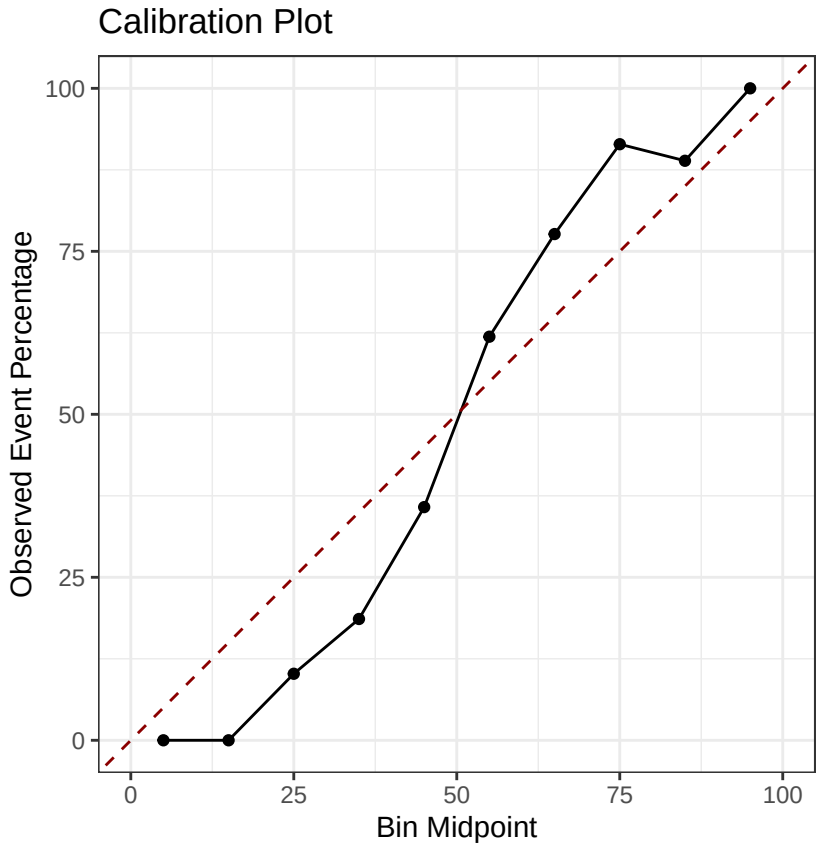


The ROC curve and the area under the curve (AUC) provide insight into the model's ability to distinguish between positive and negative classes. An AUC close to 1 indicates excellent discrimination, whereas an AUC close to 0.5 suggests that the model performs no better than by chance.

Finally, we can examine the calibration plot. This shows how well the predicted probabilities match the observed frequencies in the test set. In a well-calibrated model, the points should lie close to the diagonal line. This indicates that a predicted probability of 0.8, for example, corresponds to the positive class occurring in approximately 80% of instances with that predicted probability. A calibration plot helps us to assess whether the model's predicted probabilities are reliable. For example, suppose the model predicts a probability of 0.8 for a set of reviews being 'positive'. In that case, we hope about 80% of those reviews are positive—the plot groups predictions by probability score. For example, it groups all reviews where P(positive) is between 0.7 and 0.8 and plots the proportion of positive reviews observed in each group against the predicted probability. A perfectly calibrated model would have points along the diagonal line (y = x).

```
# Create a data frame containing the observed outcomes and the predicted probabilities

calibration_data <- data.frame(obs = polarity_test, prob_pos = svm_predict_probs_caret_

# Compute calibration information
```

```r
calibration_obj_svm <- calibration(
  obs ~ prob_pos,
  # Formula: observed variable ~ predicted probability variable
  data = calibration_data,
  # The data frame containing these variables
  class = "pos",
  # Specify the positive class name from the 'obs' column
  cuts = 10,
  # Number of bins (quantiles) for grouping probabilities
  method = "quantile" # Method for creating bins
)

calibration_data_for_plot <- calibration_obj_svm$data

ggplot(data = calibration_data_for_plot, aes(x = midpoint, y = Percent)) +
  geom_line() + # Plot the calibration line connecting the points
  geom_point() + # Plot the points for each bin
  geom_abline(
    intercept = 0,
    slope = 1,
    linetype = "dashed",
    color = "darkred"
  ) +
  scale_x_continuous(name = "Bin Midpoint", limits = c(0, 100)) +
  scale_y_continuous(name = "Observed Event Percentage", limits = c(0, 100)) +
  ggtitle("Calibration Plot") +
  theme_bw() +
  coord_equal()
```



The closer the black line representing the model's calibration is to the red dashed line representing perfect calibration, the more reliable the model's probabilities are. Deviations from this indicate either overconfidence, where the line is above the diagonal for low probabilities and below it for high probabilities or underconfidence. Poor calibration means that, while the model may make the correct classification, the stated confidence level (i.e. the probability) may be unreliable.

7.2 Logistic Regression

Logistic Regression is one of the most well-known types of supervised models and is well-suited for classification because `textmodel_lr` is already built-in, it is practical to use in R. In addition, this implementation includes L2 regularisation by default to prevent overfitting in high-dimensional text feature spaces. For consistency, we will reuse the data loading, sampling, splitting, and preprocessing steps from the SVM example.

First, ensure the necessary libraries are loaded:

```r
set.seed(42)  # Set seed for reproducibility

library(quanteda)
library(quanteda.textmodels)  # For textmodel_lr
library(caret)  # For evaluation metrics
library(pROC)  # For ROC analysis (optional, but good for binary classification)
library(ggplot2)  # For plotting (optional)

# Load the movie review corpus and sample a subset (matching the SVM example)
corpus_reviews_lr <- corpus_sample(data_corpus_LMRD, 2000)  # Sample 2000 reviews
```

We will use the same data splitting logic as in the SVM example, ensuring a stratified split based on the `polarity` variable to maintain the proportion of positive and negative reviews in both the training and test sets.

```r
# Extract the polarity label as the target variable and convert it to a factor
# Assuming the polarity variable is binary ('neg', 'pos') in your corpus object
polarity_labels_lr <- factor(corpus_reviews_lr$polarity)

# Identify documents with valid polarity labels (not NA)
valid_docs_index_lr <- which(!is.na(polarity_labels_lr))

# Subset the corpus and polarity labels to only include documents with valid
# polarity
corpus_reviews_valid_lr <- corpus_reviews_lr[valid_docs_index_lr]
polarity_valid_lr <- polarity_labels_lr[valid_docs_index_lr]

# Ensure polarity_valid_lr is a factor with levels 'neg', 'pos' in that
# specific order
polarity_valid_lr <- factor(polarity_valid_lr, levels = c("neg", "pos"))

# Check if both levels ('neg', 'pos') are present
if (!all(c("neg", "pos") %in% levels(polarity_valid_lr)) || any(table(polarity_valid_lr) ==
    0)) {
    stop("The sampled corpus subset does not contain both 'neg' and 'pos' classes after filtering
}

# Manually create stratified split indices (reusing the logic from the SVM
# example) Get indices for each class
neg_indices_lr <- which(polarity_valid_lr == "neg")
pos_indices_lr <- which(polarity_valid_lr == "pos")

# Determine the number of instances for train/test per class (70/30 split)
set.seed(42)  # for reproducibility
train_size_neg_lr <- floor(0.7 * length(neg_indices_lr))
train_size_pos_lr <- floor(0.7 * length(pos_indices_lr))
```

```
# Sample indices for training set from each class
train_indices_neg_lr <- sample(neg_indices_lr, size = train_size_neg_lr, replace = FALS
train_indices_pos_lr <- sample(pos_indices_lr, size = train_size_pos_lr, replace = FALS

# Combine training indices
train_index_lr <- c(train_indices_neg_lr, train_indices_pos_lr)

# The remaining indices are for the test set
all_valid_indices_lr <- seq_along(polarity_valid_lr)
test_index_lr <- all_valid_indices_lr[!all_valid_indices_lr %in% train_index_lr]

# Split the corpus subset and polarity labels into training and testing sets
corpus_reviews_train_lr <- corpus_reviews_valid_lr[train_index_lr]
corpus_reviews_test_lr <- corpus_reviews_valid_lr[test_index_lr]

polarity_train_lr <- polarity_valid_lr[train_index_lr]
polarity_test_lr <- polarity_valid_lr[test_index_lr]

# Check the distribution of the split
print("Training set class distribution (LR example):")
```

```
## [1] "Training set class distribution (LR example):"
```

```
print(table(polarity_train_lr))
```

```
## polarity_train_lr
## neg pos
## 716 683
```

```
print("Testing set class distribution (LR example):")
```

```
## [1] "Testing set class distribution (LR example):"
```

```
print(table(polarity_test_lr))
```

```
## polarity_test_lr
## neg pos
## 307 294
```

Next, we preprocess the training and test corpus subsets to create DFMs, applying similar cleaning steps as before and matching the test DFM features to the training DFM.

```
# Tokenise and preprocess the training corpus
tokens_train_lr <- tokens(corpus_reviews_train_lr, what = "word", remove_punct = TRUE,
    remove_symbols = TRUE, remove_numbers = TRUE, remove_url = TRUE, remove_separators
    tokens_tolower() %>%
    tokens_select(stopwords("english"), selection = "remove")
```

```
# Tokenise and preprocess the test corpus
tokens_test_lr <- tokens(corpus_reviews_test_lr, what = "word", remove_punct = TRUE,
    remove_symbols = TRUE, remove_numbers = TRUE, remove_url = TRUE, remove_separators = TRUE) %>
    tokens_tolower() %>%
    tokens_select(stopwords("english"), selection = "remove")

# Create dfms
dfm_train_lr <- dfm(tokens_train_lr)
dfm_test_lr <- dfm(tokens_test_lr)

# Ensure the test dfm has the same features as the training dfm
dfm_test_matched_lr <- dfm_match(dfm_test_lr, features = featnames(dfm_train_lr))

# Display DFM dimensions
cat("Dimensions of Training DFM (LR example):", dim(dfm_train_lr), "\n")
```

```
## Dimensions of Training DFM (LR example): 1399 23543
```

```
cat("Dimensions of Matched Test DFM (LR example):", dim(dfm_test_matched_lr), "\n")
```

```
## Dimensions of Matched Test DFM (LR example): 601 23543
```

Now, we train the Regularized Logistic Regression model using `textmodel_lr`.

```
# Train the textmodel_lr model for binary classification The training labels
# are the polarity labels from the training corpus
library(quanteda.textmodels)

model_lr_lmrd <- textmodel_lr(dfm_train_lr, polarity_train_lr)

# Print the model summary
summary(model_lr_lmrd)
```

```
##
## Call:
## textmodel_lr.dfm(x = dfm_train_lr, y = polarity_train_lr)
##
## Lambda Min:
## [1] 0.009405
##
## Lambda 1se:
## [1] 0.01498
##
## Estimated Feature Scores:
##     (Intercept) just thought finish whole year without giving single movie bomb
## pos      0.1042    0       0      0     0    0       0      0      0     0    0
##     rating friend brought notorious turd house last night feared  worst knowing
```

```
## pos      0      0       0         0    0     0    0     0      0 -1.281       0
##     reputation god-awful anticipated mexican-made mess dubbed english produced
## pos          0         0           0            0    0      0       0        0
```

We then use the trained model to predict the polarity labels for the documents in the test set.

```
# Predict the classes for the matched test set
predictions_lr_lmrd <- predict(model_lr_lmrd, newdata = dfm_test_matched_lr)

# Display the first few predictions
head(predictions_lr_lmrd)
```

```
##  test/neg/6932_2.txt train/neg/6580_4.txt  test/pos/1099_7.txt
##                  neg                  pos                  pos
## train/pos/9167_7.txt  test/neg/8064_1.txt  test/neg/4057_4.txt
##                  neg                  neg                  pos
## Levels: neg pos
```

Finally, we evaluate the model's performance on the test set using a confusion matrix. Since this is a binary classification task, we can also compute metrics like ROC AUC, which is similar to the SVM evaluation.

```
# Get the actual classes from the test corpus
actual_classes_lmrd <- polarity_test_lr

# Ensure actual and predicted classes are factors with the same levels for
# comparison The levels should already be consistent ('neg', 'pos') from the
# splitting step
confusion_matrix_lr_lmrd <- confusionMatrix(predictions_lr_lmrd, actual_classes_lmrd)

# Print the confusion matrix and performance statistics
print(confusion_matrix_lr_lmrd)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction neg pos
##        neg 232  46
##        pos  75 248
##
##                Accuracy : 0.7987
##                  95% CI : (0.7643, 0.83)
##     No Information Rate : 0.5108
##     P-Value [Acc > NIR] : < 2e-16
##
##                   Kappa : 0.598
##
```

```
##  Mcnemar's Test P-Value : 0.01091
##
##             Sensitivity : 0.7557
##             Specificity : 0.8435
##          Pos Pred Value : 0.8345
##          Neg Pred Value : 0.7678
##              Prevalence : 0.5108
##          Detection Rate : 0.3860
##    Detection Prevalence : 0.4626
##       Balanced Accuracy : 0.7996
##
##        'Positive' Class : neg
##
```

7.3 Naive Bayes (NB)

As with SVM, Naive Bayes is a simple yet efficient text classification model. It belongs to a family of probabilistic classifiers based on applying Bayes' Theorem with a "naive" that every word in a document is conditionally independent of every other word given the class. The classifier calculates this value for each class and assigns the document to the class with the highest resulting probability. Despite its simplifying assumption, Naive Bayes performs remarkably well for many text classification tasks, such as spam detection and document categorization.

For this example, we will use data from the Manifesto Project (also known as the Comparative Manifesto Project (CMP)). To use this data, ensure you have signed up, downloaded the API key, loaded the package, and set the key.

```
library(manifestoR)  # Used to download Manifesto Project data
library(quanteda)  # For text analysis and DFM creation
library(quanteda.textmodels)  # For the Naive Bayes model
library(ggplot2)  # For plotting
library(DescTools)  # For Krippendorf's Alpha
library(caret)  # For confusion matrix and classification metrics
library(dplyr)  # For data manipulation

# Set your Manifesto Project API key Replace 'manifesto_apikey.txt' with the
# path to your API key file
mp_setapikey("manifesto_apikey.txt")
```

As the entire MP corpus is relatively large, here we will only focus on part of it: the manifestos for the 2015 United Kingdom general election. To achieve this, we have created a data frame listing the party and year for the 'mp_corpus' command. Please note that The Manifesto Project uses unique codes for each party, which can be found in its codebook.

```
# Party codes: CON(51620), LABOUR(51320), LIBDEM(51421), SNP(51901), PLAID
# CYMRU(51902), GREEN(51110), UKIP(51951)
manifestos_info <- data.frame(party = c(51320, 51620, 51110, 51421, 51901, 51902,
    51951), date = rep(201505, 7))

# Download the specified manifestos as a manifestoR corpus
manifesto_corpus <- mp_corpus(manifestos_info)
```

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

We will create a data frame containing the quasi-sentences (the unit of analysis), their assigned `cmp_code`, and the `party` they belong to. This process extracts the core data needed for our analysis.

```
# Programmatically create a list of data frames, one for each document
corpus_list <- lapply(manifesto_corpus, function(doc) {
    data.frame(texts = content(doc), cmp_code = codes(doc), party = doc$meta$party,
        stringsAsFactors = FALSE)
})

# Combine the list of data frames into a single data frame
manifesto_data <- do.call(rbind, corpus_list)
```

Next, we clean and transform the data. We convert the `party` codes to factor labels (e.g., "CON", "LABOUR") for clarity, ensure text and code columns are in the correct format, and remove entries with `NA` codes, which typically correspond to un-coded document headers and titles.

```
# Transform and clean the data frame
manifesto_data$party <- factor(manifesto_data$party, levels = c(51110, 51320, 51421,
    51620, 51901, 51902, 51951), labels = c("GREEN", "LABOUR", "LIBDEM", "CON", "PC",
    "SNP", "UKIP"))
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))

# Remove NA values (sentences without a code)
manifesto_data <- na.omit(manifesto_data)

# Display the structure of the cleaned data
str(manifesto_data)
```

```
## 'data.frame':    9763 obs. of  3 variables:
##  $ texts   : chr  "Our manifesto begins with the Budget Responsibility Lock we offe
##  $ cmp_code: num  414 414 414 414 414 414 414 414 414 414 ...
##  $ party   : chr  "LABOUR" "LABOUR" "LABOUR" "LABOUR" ...
```

```
##  - attr(*, "na.action")= 'omit' Named int [1:1014] 1 2 3 4 5 6 9 19 20 21 ...
##   ..- attr(*, "names")= chr [1:1014] "51320_201505.1" "51320_201505.2" "51320_201505.3" "51320
```

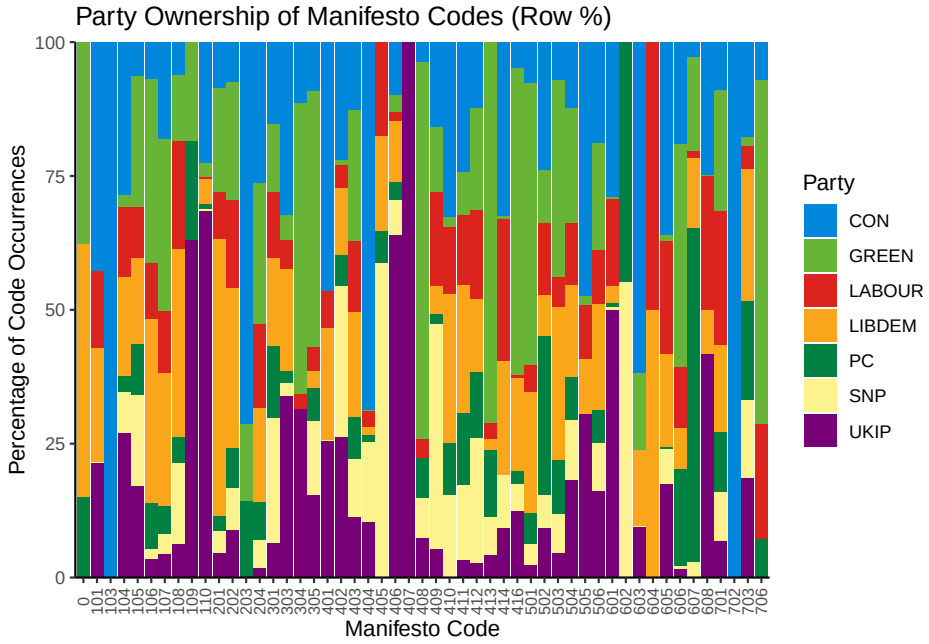
We can calculate row percentages to understand how much a party "owns" a specific code. This shows the proportion of a code's total appearances attributed to each party.

```
# Calculate row percentages of code occurrences by party
prop_row <- as.data.frame(prop.table(table(manifesto_data$cmp_code, manifesto_data$party),
    margin = 1) * 100)
names(prop_row) <- c("Code", "Party", "Percentage")

# Display head of row percentages
head(prop_row)
```

```
##   Code Party Percentage
## 1    0   CON   0.000000
## 2  101   CON  42.857143
## 3  103   CON 100.000000
## 4  104   CON  28.632479
## 5  105   CON   6.382979
## 6  106   CON   6.896552
```

Visualizing the `prop_row` object as a stacked bar chart clarifies how parties utilize different codes.

```
# Define party colours for consistent plotting
party_colors <- c(CON = "#0087DC", GREEN = "#67B437", LABOUR = "#DC241F", LIBDEM = "#FAA61A",
    PC = "#008142", SNP = "#FDF38E", UKIP = "#780077")

# Plot the row percentages
ggplot(data = prop_row, aes(x = factor(Code), y = Percentage, fill = Party)) + scale_fill_manual(
    geom_bar(stat = "identity", position = "stack") + scale_y_continuous(expand = c(0,
    0)) + theme_classic() + theme(axis.text.x = element_text(angle = 90, vjust = 0.5,
    size = 8)) + labs(x = "Manifesto Code", y = "Percentage of Code Occurrences",
    title = "Party Ownership of Manifesto Codes (Row %)")
```



This chart shows that some parties dominate specific categories. For example, we can see that UKIP is the only one to use code 406 (Protectionism: Positive). Note that these are percentages; a high percentage might reflect dominance but could be based on a very small number of sentences. Other categories are more evenly distributed across parties. We can also analyze the thematic composition of each party's manifesto by calculating column percentages:

```
# Calculate column percentages (percentage of party manifesto per code)
prop_col <- as.data.frame(prop.table(table(manifesto_data$cmp_code, manifesto_data$part
    margin = 2) * 100)
names(prop_col) <- c("Code", "Party", "Percentage")

# Display head of column percentages
head(prop_col)
```

```
##   Code Party Percentage
## 1    0   CON  0.0000000
## 2  101   CON  0.3778338
## 3  103   CON  0.1259446
## 4  104   CON  4.2191436
## 5  105   CON  0.3778338
## 6  106   CON  0.2518892
```

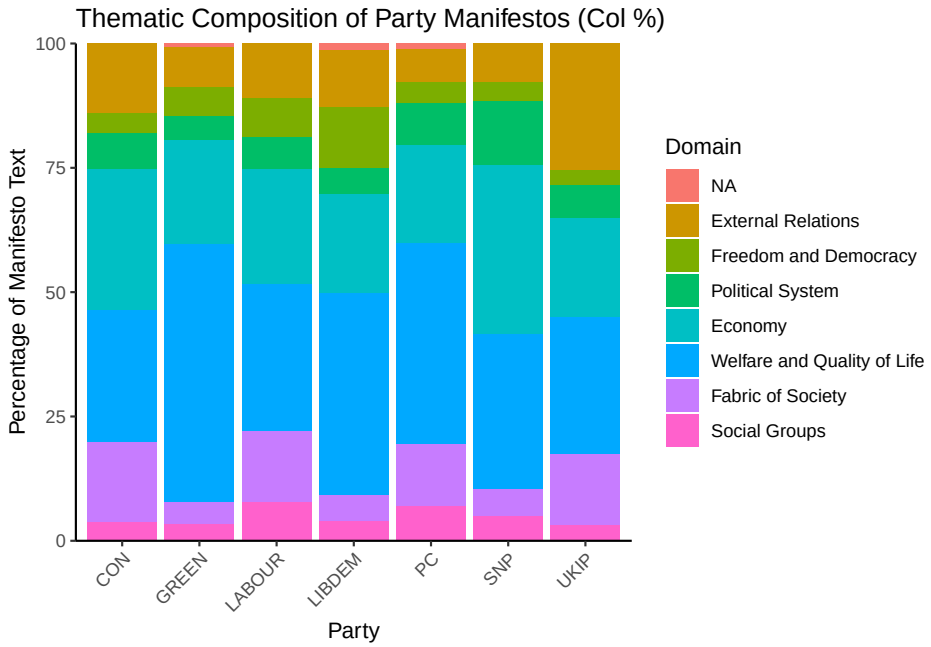
With 57 possible codes, grouping them into the 7 thematic domains defined by the Manifesto Project is more practical. We create a new `Domain` variable and assign each code to its corresponding domain.

```
# Assign codes to their respective domains
prop_col$Code <- as.numeric(as.character(prop_col$Code))
prop_col$Domain <- cut(prop_col$Code, breaks = c(-1, 0, 110, 204, 305, 416, 507,
    608, 706), labels = c("NA", "External Relations", "Freedom and Democracy", "Political System"
    "Economy", "Welfare and Quality of Life", "Fabric of Society", "Social Groups"))

# Display head with Domain information
head(prop_col)
```

```
##   Code Party Percentage             Domain
## 1    0   CON  0.0000000                 NA
## 2  101   CON  0.3778338 External Relations
## 3  103   CON  0.1259446 External Relations
## 4  104   CON  4.2191436 External Relations
## 5  105   CON  0.3778338 External Relations
## 6  106   CON  0.2518892 External Relations
```

We can now plot the distribution of these domains within each party's manifesto.

```
# Plot the column percentages aggregated by domain
ggplot(data = prop_col, aes(x = Party, y = Percentage, fill = Domain)) + geom_bar(stat = "identit
    position = "stack") + scale_y_continuous(expand = c(0, 0)) + theme_classic() +
    labs(x = "Party", y = "Percentage of Manifesto Text", title = "Thematic Composition of Party
    theme(axis.text.x = element_text(angle = 45, hjust = 1))
```



The plot shows that "Welfare and Quality of Life" and "Economy" are dominant

themes across most parties. We can also see party-specific focuses, such as UKIP's emphasis on "External Relations".

Now, let us turn to the Naive Bayes. First, she place our data into a corpus object:

```
manifesto_corpus <- corpus(manifesto_data, text_field = "texts")
summary(manifesto_corpus, 10)
```

```
## Corpus consisting of 9763 documents, showing 10 documents:
##
##              Text Types Tokens Sentences cmp_code  party
##   51320_201505.7    13     14         1      414 LABOUR
##   51320_201505.8    29     35         1      414 LABOUR
##  51320_201505.10     9      9         1      414 LABOUR
##  51320_201505.11     7      7         1      414 LABOUR
##  51320_201505.12    17     17         1      414 LABOUR
##  51320_201505.13    21     22         1      414 LABOUR
##  51320_201505.14    10     10         1      414 LABOUR
##  51320_201505.15    18     20         1      414 LABOUR
##  51320_201505.16    26     30         1      414 LABOUR
##  51320_201505.17    20     22         1      414 LABOUR
```

To train and evaluate our model, we must split the data into training and test sets. We'll use 80% for training and 20% for testing. `set.seed` ensures that the random sampling is reproducible.

```
set.seed(42)

# Generate a random sample of document indices for the training set
id_train <- sample(1:ndoc(manifesto_corpus), size = ndoc(manifesto_corpus) * 0.8,
    replace = FALSE)

# Create training and test sets by subsetting the corpus
train_corpus <- corpus_subset(manifesto_corpus, 1:ndoc(manifesto_corpus) %in% id_train)
test_corpus <- corpus_subset(manifesto_corpus, !1:ndoc(manifesto_corpus) %in% id_train)
```

Next, we create Document-Feature Matrices (DFMs) from our corpus subsets. A DFM is a table where rows represent documents, columns represent words (features), and cells contain word counts. We perform basic text cleaning during this step: removing punctuation, numbers, and symbols. We also trim the DFM to remove very infrequent terms, which helps reduce noise and improve model performance.

```
# Create DFM for the training set
train_dfm <- train_corpus %>%
    tokens(remove_punct = TRUE, remove_numbers = TRUE, remove_symbols = TRUE) %>%
    dfm() %>%
```

```
    dfm_trim(min_termfreq = 5)

# Create DFM for the test set We don't trim the test set yet, as we will match
# its features to the training set
test_dfm <- test_corpus %>%
    tokens(remove_punct = TRUE, remove_numbers = TRUE, remove_symbols = TRUE) %>%
    dfm()
```

We train the Naive Bayes model using the training DFM and the `cmp_code` variable as the target label.

```
manifesto_nb <- textmodel_nb(train_dfm, y = docvars(train_corpus, "cmp_code"))
summary(manifesto_nb)
```

The model summary provides estimated feature scores (the probability of a word given a class). We aim to evaluate the model's performance on unseen data from the test set. Keep in mind that the Naive Bayes model can only make predictions based on features it has seen during training. Therefore, we must align the feature set of the test DFM with the training DFM. The `dfm_match()` function ensures that the test DFM has the same features as the training DFM.

```
matched_test_dfm <- dfm_match(test_dfm, features = featnames(train_dfm))
```

The number of features in the matched test DFM is now identical to the training DFM. We can now predict the classes for the test set.

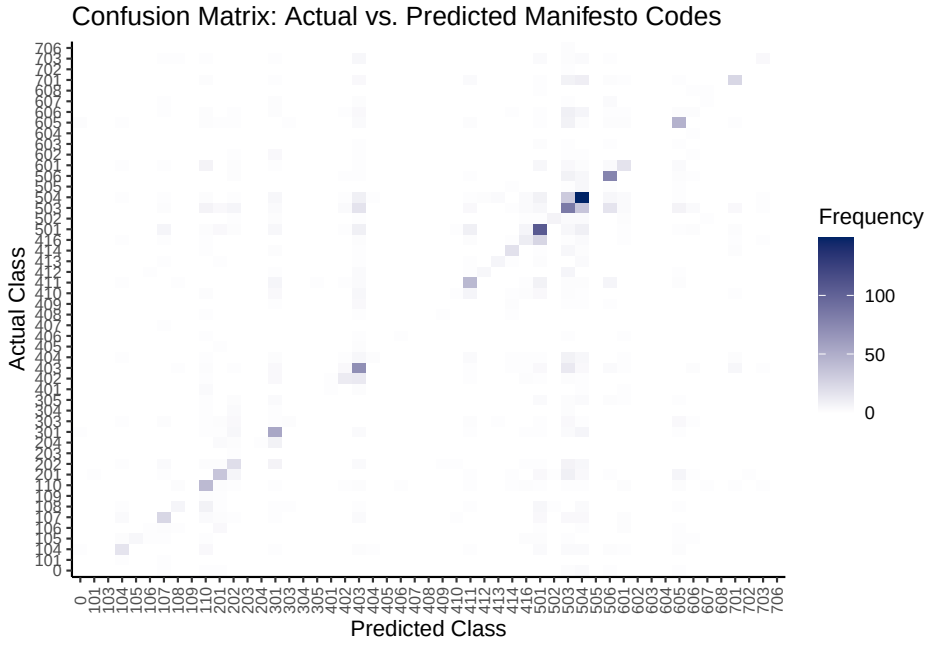
```
# Predict the classes for the matched test set
predicted_class <- predict(manifesto_nb, newdata = matched_test_dfm)

# Get the actual classes from the test corpus's document variables
actual_class <- docvars(test_corpus, "cmp_code")

# Create a confusion matrix (table of actual vs. predicted classes)
table_class <- table(Actual = actual_class, Predicted = predicted_class)

# Display a subset of the confusion matrix due to its large size
table_class[1:10, 1:10]
```

```
##        Predicted
## Actual   0 101 103 104 105 106 107 108 109 110
##    0     0   0   0   0   0   0   1   0   0   1
##    101   0   0   0   1   0   0   1   0   0   0
##    104   1   0   0  16   0   0   0   0   0   4
##    105   0   0   0   2   5   1   1   0   0   1
##    106   0   0   0   0   0   1   2   1   0   0
##    107   0   0   0   3   0   0  24   0   0   3
##    108   0   0   0   1   0   0   0   6   0   8
##    109   0   0   0   0   0   0   0   0   0   2
```

```
##     110  0    0    0    0    0    0    0    2    0   42
##     201  0    1    0    0    0    0    1    0    0    0
```

A confusion matrix table can be large and difficult to interpret. A heatmap offers a more intuitive visualization, highlighting where the model is accurate (strong diagonal) and where it makes errors (dark cells off the diagonal).

```
# Convert the table to a data frame for plotting
table_class_df <- as.data.frame(table_class)
names(table_class_df) <- c("Actual", "Predicted", "Frequency")

# Plot the confusion matrix as a heatmap
ggplot(data = table_class_df, aes(x = Predicted, y = Actual)) + geom_tile(aes(fill = Fr
    scale_fill_gradient(low = "white", high = "#002366", name = "Frequency") + labs(x =
    y = "Actual Class", title = "Confusion Matrix: Actual vs. Predicted Manifesto Codes
    theme_classic() + theme(axis.text.x = element_text(angle = 90, vjust = 0.5, size =
    axis.text.y = element_text(size = 8))
```



To further understand how well NB has done, we can calculate aggregate performance metrics like accuracy, precision, recall, and F1-score. The `caret` package's `confusionMatrix` function is ideal.

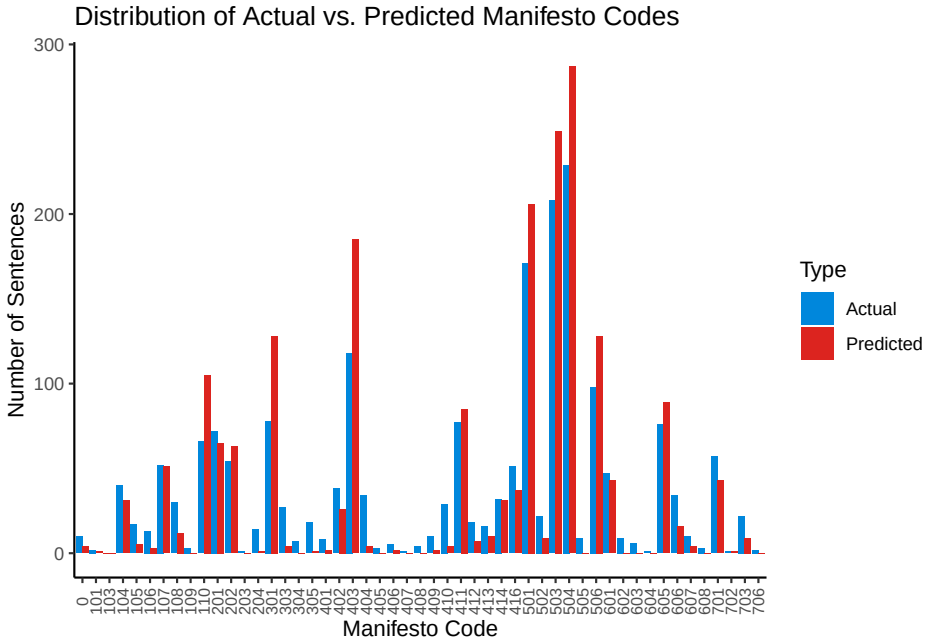
```
# Ensure levels are consistent for the confusion matrix function
all_levels <- unique(c(as.character(actual_class), as.character(predicted_class)))
actual_class_factor <- factor(actual_class, levels = all_levels)
predicted_class_factor <- factor(predicted_class, levels = all_levels)
```

```r
# Calculate various classification metrics using caret::confusionMatrix
classification_metrics <- confusionMatrix(predicted_class_factor, actual_class_factor)

# Display macro-averaged F1-score (calculated manually from per-class stats)
macro_f1 <- mean(classification_metrics$byClass[, "F1"], na.rm = TRUE)
cat("Macro-Averaged F1-Score:", macro_f1, "\n")
```

```
## Macro-Averaged F1-Score: 0.3686071
```

In addition to this, comparing the frequency distribution of predicted classes against the actual classes can reveal if the model is biased towards or against specific categories.

```r
# Create data frames for actual and predicted class counts and combine them
actual_counts_df <- as.data.frame(table(actual_class), stringsAsFactors = FALSE) %>%
    rename(Code = actual_class, Frequency = Freq) %>%
    mutate(Type = "Actual")

predicted_counts_df <- as.data.frame(table(predicted_class), stringsAsFactors = FALSE) %>%
    rename(Code = predicted_class, Frequency = Freq) %>%
    mutate(Type = "Predicted")

class_distribution_df <- bind_rows(actual_counts_df, predicted_counts_df)

# Plot the distribution
ggplot(class_distribution_df, aes(x = Code, y = Frequency, fill = Type)) + geom_bar(stat = "ident
    position = "dodge") + theme_classic() + labs(x = "Manifesto Code", y = "Number of Sentences",
    title = "Distribution of Actual vs. Predicted Manifesto Codes") + scale_fill_manual(values =
    Predicted = "#DC241F")) + theme(axis.text.x = element_text(angle = 90, vjust = 0.5,
    size = 8))
```



This plot helps visualize if the model over-predicts (red bar is taller) or under-predicts (blue bar is taller) specific codes compared to their true frequency in the test set.

Finally, we can use Krippendorff's α again to measure the agreement between the model's predictions and the human-assigned codes, accounting for chance agreement.

```
reliability_data <- as.matrix(rbind(as.character(actual_class), as.character(predicted_

# Calculate for nominal data
kripp_alpha <- KrippAlpha(reliability_data, method = "nominal")
kripp_alpha$value
```

```
## [1] 0.420661
```

Interpreting the α value requires context. Research by Mikhaylov et al. (2012) estimates the agreement *among trained human coders* for the Manifesto Project to be between 0.350 and 0.400. Seen this way, our $\alpha = 0.43$ from our simple automated model is therefore quite good!

7.4 Exercises

1. Modify the `caret` SVM example to use TF-IDF weighting for the DFM rather than raw term frequency. Does this affect the performance of the SVM model when evaluated using cross-validation?

2. Using the movie review sentiment data, train a logistic regression model with the `glmnet` method using the `caret::train()` function (ensure you have the `glmnet` package installed). Use cross-validation for training. Compare the performance of this model with that of the SVM model using a confusion matrix, an ROC-AUC plot, and a calibration plot.

3. Investigate the impact of various pre-processing steps (e.g. removing numbers, stemming versus non-stemming and using bigrams) on the performance of the Naive Bayes model with the Manifesto Project data. Evaluate performance using cross-validation and compare confusion matrices.

4. Investigate how to deal with imbalanced classes in text classification. For a dataset with imbalanced classes, research a relevant technique (e.g. the ROSE package or the sampling options in the `trainControl` function of the `caret` package) and apply it during cross-validated training. Does this improve the performance of the minority class compared to training without addressing the imbalance?

5. For either the SVM or Naive Bayes model, research how to tune hyperparameters using the `tuneGrid` argument in `caret::train()`. Implement a simple hyperparameter tuning process using cross-validation, then report on the performance of the best-tuned model.

Chapter 8

Unsupervised Methods

Unlike supervised methods, which require labelled data to train a model for classifying or predicting values for new text, unsupervised learning methods in text analysis aim to discover inherent patterns and structures within text data without relying on pre-assigned labels. These methods are useful for exploring large unannotated corpora to identify recurring themes (topics), group similar documents (clustering) and reduce the dimensionality of the data.

Supervised models excel at well-defined classification tasks where labelled data is available, but unsupervised methods allow us to uncover latent structures that we may not have anticipated or would be prohibitively expensive to label. This chapter focuses on probabilistic topic modelling, a popular suite of unsupervised methods for identifying abstract 'topics' within a body of text. Each document is treated as a mixture of these topics, with a word distribution characterising each. We will cover Latent Dirichlet Allocation (LDA), seeded LDA, which incorporates prior knowledge, the Structural Topic Model (STM), which allows the inclusion of document metadata to model topic prevalence and content, and Latent Semantic Analysis (LSA) as a dimensionality reduction technique.

The original document provided a solid introduction to these methods. I have expanded upon the explanations of each technique, particularly focusing on the rationale behind key steps and the interpretation of results. I have also significantly extended the sections on model validation for LDA, STM, and LSA, as understanding how to evaluate these models is crucial for their practical application.

8.1 Latent Dirichlet Allocation (LDA)

Latent Dirichlet Allocation (LDA) is a generative probabilistic model for collections of discrete data, such as text corpora. The core idea is that each document

contains various topics, and a word distribution characterises each topic. The model assumes that the process of writing a document includes:

1. Choosing the length of the document.
2. Choosing a mixture of topics for the document.
3. For each word in the document:
    - Choosing a topic from the mixture of topics in the document.
    - Choosing a word from the word distribution of the selected topic.

LDA aims to infer these latent topic mixtures for each document and word distributions for each topic, given the observed word frequencies in the corpus.

We will primarily use the `topicmodels` package to run LDA in R. This package works with a specific document-term matrix format, so we first need to convert our `quanteda` dfm into this format using the `convert()` function. We will use the inaugural speeches corpus (`data_inaugural_dfm`) from previous chapters as our example data.

We begin by loading the necessary libraries. `topicmodels` is the core package for LDA. `quanteda` is needed for corpus and DFM manipulation. `dplyr`, `tidytext`, and `ggplot2` are essential for subsequent data manipulation and visualisation of the model outputs in a tidy format.

```
# Install topicmodels if you haven't already: install.packages('topicmodels')
library(topicmodels)
library(quanteda)  # Ensure quanteda is loaded
library(dplyr)  # For data manipulation later
library(tidytext)  # For tidying model output later
library(ggplot2)  # For visualization later


data(data_corpus_inaugural)
data_inaugural_tokens <- tokens(data_corpus_inaugural, remove_punct = TRUE, remove_symb
    remove_numbers = TRUE, remove_url = TRUE, remove_separators = TRUE, split_hyphens =
data_inaugural_tokens <- tokens_tolower(data_inaugural_tokens)
data_inaugural_tokens <- tokens_select(data_inaugural_tokens, stopwords("english"),
    selection = "remove")
data_inaugural_dfm <- dfm(data_inaugural_tokens)
data_inaugural_dfm <- dfm_compress(data_inaugural_dfm, margin = "features")

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

Once the libraries have been loaded and the data has been prepared in the required format, we will specify the parameters for the LDA model using Gibbs sampling. These parameters control the inference process, including the number of iterations, burn-in period, thinning interval, random seeds for reproducibility, and independent chains to run.

When using Gibbs sampling, specific parameters need to be set: *burnin* (the

number of initial iterations to discard), *iter* (the total number of iterations after the burn-in period), *thin* (the thinning interval), *seed* (the random seed or seeds for multiple runs), *nstart* (the number of independent chains) and *best* (whether to keep the model with the highest log-likelihood if nstart > 1). Additionally, we must set the desired number of topics to extract. This is a common challenge in topic modelling, as no definitive method exists. It often involves a combination of statistical measures, such as likelihood or coherence scores, and qualitative evaluation of the topics to determine their meaning and interpretability.

Using Gibbs sampling, we can fit an LDA model with a chosen number of topics. For example, k could be set to 10. The explanation of the parameters and the choice of k has been expanded slightly for clarity.

```
# Set parameters for Gibbs sampling
burnin <- 2000  # Number of initial iterations to discard.
iter <- 1000  # Number of iterations to keep after burnin.
thin <- 200  # Keep every 200th iteration.
seed <- list(42, 5, 24, 158, 2500)  # Seeds for multiple runs.
nstart <- 5  # Number of independent chains to run.
best <- TRUE  # Keep the best model from the multiple runs.
k_lda <- 10  # Number of topics to find.
```

With the parameters defined, we fit the LDA model to the document-term matrix. The `LDA()` function from the `topicmodels` package performs this fitting process using the specified method and control parameters.

```
# Fit the LDA model using Gibbs sampling
inaugural_lda10 <- LDA(inaugural_dtm, k = k_lda, method = "Gibbs", control = list(burnin = burnin
    iter = iter, thin = thin, seed = seed, nstart = nstart, best = best))

# Display the top words for each topic
print("Top terms per topic:")
```

```
## [1] "Top terms per topic:"
```

```
terms(inaugural_lda10, 10)  # Show top 10 terms for each of the 10 topics. These terms are the wo
```

```
##       Topic 1    Topic 2    Topic 3    Topic 4   Topic 5        Topic 6
##  [1,] "world"    "will"     "life"     "will"    "shall"        "union"
##  [2,] "peace"    "must"     "spirit"   "us"      "states"       "constitution"
##  [3,] "nations"  "make"     "things"   "america" "now"          "can"
##  [4,] "free"     "business" "nation"   "can"     "will"         "one"
##  [5,] "must"     "american" "men"      "people"  "constitution" "states"
##  [6,] "freedom"  "made"     "task"     "nation"  "congress"     "state"
##  [7,] "can"      "trade"    "purpose"  "one"     "upon"         "free"
##  [8,] "new"      "secure"   "problems" "new"     "great"        "among"
##  [9,] "men"      "law"      "without"  "must"    "years"        "within"
## [10,] "progress" "can"      "action"   "world"   "laws"         "blessings"
```

```
##       Topic 7      Topic 8    Topic 9     Topic 10
##  [1,] "freedom"    "war"      "great"     "government"
##  [2,] "let"        "just"     "may"       "people"
##  [3,] "time"       "nations"  "power"     "will"
##  [4,] "citizens"   "united"   "well"      "upon"
##  [5,] "man"        "every"    "whole"     "country"
##  [6,] "earth"      "powers"   "states"    "public"
##  [7,] "generation" "duties"   "might"     "every"
##  [8,] "human"      "commerce" "executive" "rights"
##  [9,] "liberty"    "citizens" "time"      "national"
## [10,] "courage"    "states"   "state"     "interests"
```

The `terms()` function allows us to inspect the words with the highest probability of belonging to each topic. These top words provide initial clues for interpreting the meaning of each discovered topic. By examining the words associated with each topic, we can begin to understand the themes present in the corpus.

To further explore the topic-word distributions, known as β, and prepare them for visualization, we use the `tidy()` function from the `tidytext` package. `tidytext` facilitates working with text data and model outputs in a "tidy" format, which is compatible with `dplyr` and `ggplot2` for efficient data manipulation and visualization.

```
library(tidytext)  # Ensure tidytext is loaded
library(dplyr)  # Ensure dplyr is loaded
library(ggplot2)  # Ensure ggplot2 is loaded

# Tidy the LDA model output to get the topic-word probabilities (beta) The
# 'beta' matrix represents the probability of a word belonging to a topic.
inaugural_lda10_topics <- tidy(inaugural_lda10, matrix = "beta")

# Display the structure of the tidied beta output
print(inaugural_lda10_topics)
```

```
## # A tibble: 92,090 x 3
##    topic term                 beta
##    <int> <chr>               <dbl>
##  1     1 fellow-citizens 0.0000142
##  2     2 fellow-citizens 0.000167
##  3     3 fellow-citizens 0.0000233
##  4     4 fellow-citizens 0.00000746
##  5     5 fellow-citizens 0.00124
##  6     6 fellow-citizens 0.0000164
##  7     7 fellow-citizens 0.0000181
##  8     8 fellow-citizens 0.0000150
##  9     9 fellow-citizens 0.00426
## 10    10 fellow-citizens 0.00000728
```

```
## # i 92,080 more rows
```

The resulting `inaugural_lda10_topics` data frame contains columns for the `topic`, the `term` (word), and `beta` (the probability of that word occurring in that topic). To visualize the top words for each topic, we filter this data frame to retain only the top N words per topic based on their beta values, typically the top 10, before creating a bar chart.

We select the top terms for each topic based on their beta values, grouping the data by topic and then using `slice_max` to select the top 10 terms within each group. Finally, we arrange the results for better readability.

```
# Select the top 10 terms for each topic based on beta values
inaugural_lda10_topterms <- inaugural_lda10_topics %>%
  group_by(topic) %>%          # Group the data by topic
  slice_max(beta, n = 10) %>%
  ungroup() %>%
  arrange(topic, -beta)
```

Using the filtered data, we create a faceted bar chart to visualize the top terms for each topic and their corresponding beta probabilities. Reordering the terms within each facet based on their beta values makes the plot easier to interpret. The explanation of the plot has been slightly expanded.

```
inaugural_lda10_topterms %>%
  mutate(term = reorder_within(term, beta, topic)) %>%
  ggplot(aes(beta, term, fill = factor(topic))) +      # Plot beta on the x-axis, term on the y-a
  geom_col(show.legend = FALSE) +                      # Add bars, hide legend
  facet_wrap(~ topic, scales = "free_y") +             # Create a separate plot (facet) for each
  scale_y_reordered() +
  labs(title = "Top 10 Words per Topic (LDA Beta)",
       x = "Beta (Probability in Topic)",
       y = "Term") +
  theme_minimal() # Use a minimal theme for a clean appearance.
```

This plot provides a visual overview of the keywords that define each topic.

By examining these words, we can assign a meaningful label or interpretation to each discovered topic. This step is crucial for understanding the thematic structure of the corpus.

Another important output of LDA is the document-topic distribution, known as γ, which represents the proportion of each topic present in each document. This information allows us to identify which topics are most prominent in specific documents. We again use the `tidy()` function to extract this data in a convenient format.

```
# Tidy the LDA model output to get the document-topic probabilities (gamma) The
# 'gamma' matrix represents the proportion of each topic in each document.
inaugural_lda10_documents <- tidy(inaugural_lda10, matrix = "gamma")

# Display the structure of the tidied gamma output
print(inaugural_lda10_documents)
```

```
## # A tibble: 600 x 3
##    document        topic  gamma
##    <chr>           <int>  <dbl>
##  1 1789-Washington     1 0.0351
##  2 1793-Washington     1 0.0536
##  3 1797-Adams          1 0.0686
##  4 1801-Jefferson      1 0.0594
##  5 1805-Jefferson      1 0.0335
##  6 1809-Madison        1 0.0326
##  7 1813-Madison        1 0.0417
##  8 1817-Monroe         1 0.0159
##  9 1821-Monroe         1 0.0130
## 10 1825-Adams          1 0.0490
## # i 590 more rows
```

The `inaugural_lda10_documents` data frame contains columns for the `document`, the `topic`, and `gamma` (the proportion of that topic in that document). We can visualize the topic distribution across documents, for example, by looking at the top topics in a selection of documents or by visualizing the distribution of a specific topic across all documents. Here, we will visualize the topic distribution for a few selected documents to illustrate how topics are mixed within documents.

We select a subset of documents to visualize their topic distributions. This allows us to examine the topic proportions in individual documents and see how different topics contribute to the content of each document.

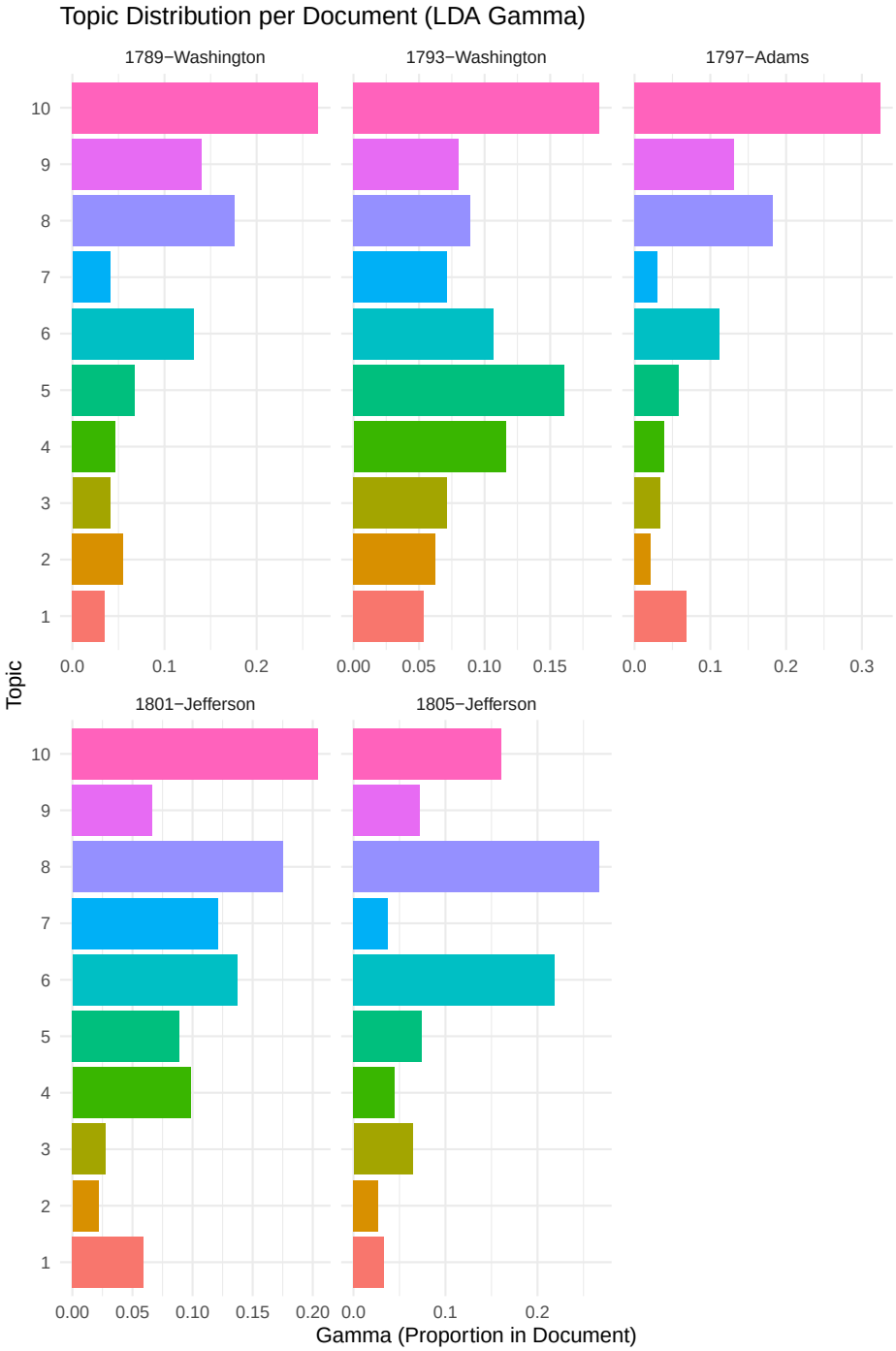
```
# Select a few documents to visualise (e.g., the first few)
selected_docs <- unique(inaugural_lda10_documents$document)[1:5] # Select the names of the first

# Filter the gamma data for selected documents and arrange
```

```
inaugural_lda10_selected_docs <- inaugural_lda10_documents %>%
  filter(document %in% selected_docs) %>% # Keep only the rows where the document name
  arrange(document, -gamma) # Arrange the data first by document name, then by gamma i

# Show the structure of the selected documents' gamma data
print(inaugural_lda10_selected_docs)
```

```
## # A tibble: 50 x 3
##    document        topic  gamma
##    <chr>           <int>  <dbl>
##  1 1789-Washington    10 0.267
##  2 1789-Washington     8 0.176
##  3 1789-Washington     9 0.140
##  4 1789-Washington     6 0.132
##  5 1789-Washington     5 0.0674
##  6 1789-Washington     2 0.0548
##  7 1789-Washington     4 0.0463
##  8 1789-Washington     3 0.0407
##  9 1789-Washington     7 0.0407
## 10 1789-Washington     1 0.0351
## # i 40 more rows
```

With the data filtered for the selected documents, we can create a faceted bar plot. This plot shows the proportion of each topic within each of the selected documents, providing a visual representation of the document-topic mixtures.

```
inaugural_lda10_selected_docs %>%
    mutate(topic = factor(topic)) %>%
    ggplot(aes(gamma, topic, fill = topic)) + geom_col(show.legend = FALSE) + facet_wra
    scales = "free_x") + labs(title = "Topic Distribution per Document (LDA Gamma)",
    x = "Gamma (Proportion in Document)", y = "Topic") + theme_minimal()
```



It might be a good idea at this point to refer back to the "Validate, Vali-

date,"Validate" by Grimmer & Stewart (2013), which we mentioned earlier. This is especially relevant here, as unsupervised methods literally "find" patterns, with the risk that — given we tend to be very good at recognizing patterns – we find non-sense patterns. Thus, we must ensure that the patterns or topics we find are meaningful and useful. The validation needed for this often involves a combination of quantitative metrics and qualitative interpretation.

On the quantitative side, we can use **topic coherence**, which assesses the semantic similarity between the high-scoring words in a topic. Topics with high coherence tend to be more human-interpretable. While the `topicmodels` package doesn't directly provide a built-in topic coherence measure, it can be computed using other packages like `ldatuning` or manual calculation based on word co-occurrence statistics in the corpus (we will return to this when discussing STM, where the package makes this significantly easier).

Another quantitative approach involves examining **perplexity**, a measure of how well the model predicts a held-out set of documents. Lower perplexity generally indicates a better model fit. As the `topicmodels` package provides the log-likelihood of the model, we can derive the perplexity from it.

```
# Get the log-likelihood of the fitted model
log_likelihood <- logLik(inaugural_lda10)
N <- sum(inaugural_dtm)  # Calculate the total words in the document-term matrix.
perplexity <- exp(-log_likelihood/N)  # Calculate perplexity using the formula.
print(paste("Perplexity of the LDA model:", perplexity))
```

```
## [1] "Perplexity of the LDA model: 1262.87536366491"
```

Most importantly, beyond quantitative metrics, **qualitative evaluation** is essential. For LDA, this involves carefully inspecting the top terms for each topic (as shown in the beta visualization) and assessing whether they form a coherent theme. We also examine the document-topic distributions (gamma visualization) to see if documents with high proportions of a given topic are indeed about the interpreted theme. Comparing topic assignments to known characteristics of documents or human judgments (if available) can also provide valuable validation. Experimenting with different numbers of topics (k) and comparing the resulting topics qualitatively and quantitatively is a standard practice in LDA model validation.

Finally, the choice of the number of topics (k) is the most problematic aspect of validation as, while quantitative measures like coherence or perplexity can guide this choice, ultimately, the interpretability and utility of the topics for the research question are paramount. To compare models with different numbers of topics, one would typically train multiple LDA models (each with a different k) and then compare their log-likelihood/perplexity or topic coherence scores, visualizing these scores against k.

8.2 Seeded Latent Dirichlet Allocation (sLDA)

An alternative to the above approach is one known as seeded-LDA. This approach uses seed words to 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:

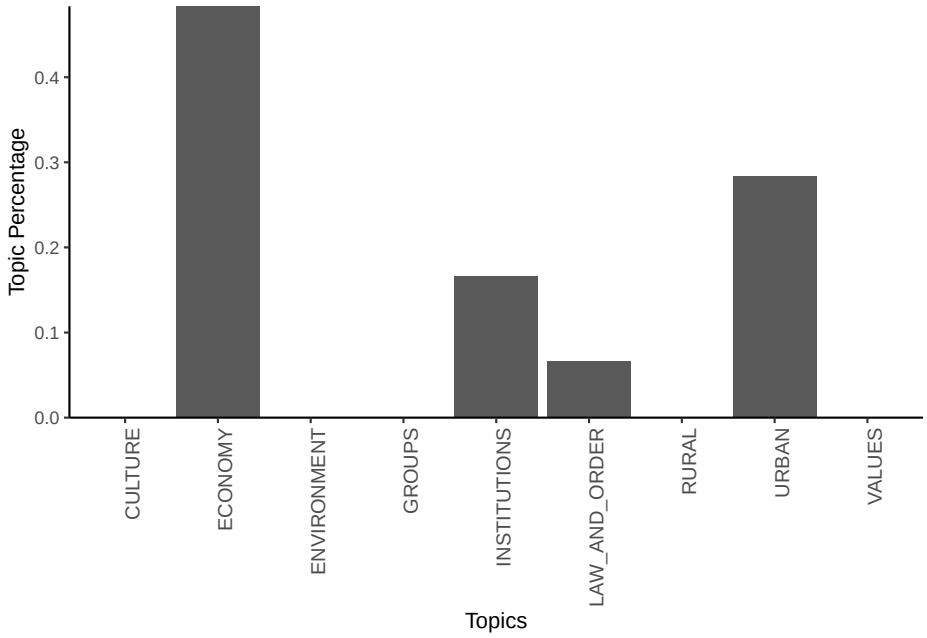
```
library(seededlda)
library(quanteda.dictionaries)

set.seed(42)
```

Next, we need to specify a selection of seed words in dictionary form. While we can construct a dictionary ourselves, we 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_LaverGarry)
```

Note that using the dictionary has ensured that we only use the categories in the dictionary. This means we can look at which topics are in each inaugural speech and which terms were most likely for each. 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   demands
## 3    C CULTURE operation
## 4    D CULTURE      idea
## 5    E CULTURE     lines
## 6    F CULTURE     price
```

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 references the people. In the same way, we can make similar observations for the other categories.

8.3 Structural Topic Model (STM)

The Structural Topic Model (STM) is another probabilistic topic modelling approach that extends traditional LDA by explicitly incorporating document

metadata (such as publication date, author, source, or other document-level characteristics) into the model. STM can model:

1. **Topic prevalence:** How the proportion of topics in a document relates to metadata. For example, how does the prevalence of "business" topics change over time or differ between authors?
2. **Topic content:** How the words associated with a topic (the β distribution) vary according to metadata. For example, are the words used to discuss "the environment" different in documents from different political parties?

One of the key advantages of STM is that it uses this metadata to estimate the topic-document (θ) and topic-word (β) distributions, potentially leading to more coherent and meaningful topics and allowing researchers to directly test hypotheses about the relationship between metadata and language use. Also, Unlike standard LDA, where the hyperparameters (α and β) are typically fixed, STM allows them to be influenced by covariates.

Figure 8.1 provides a diagram illustrating the structure of the STM.

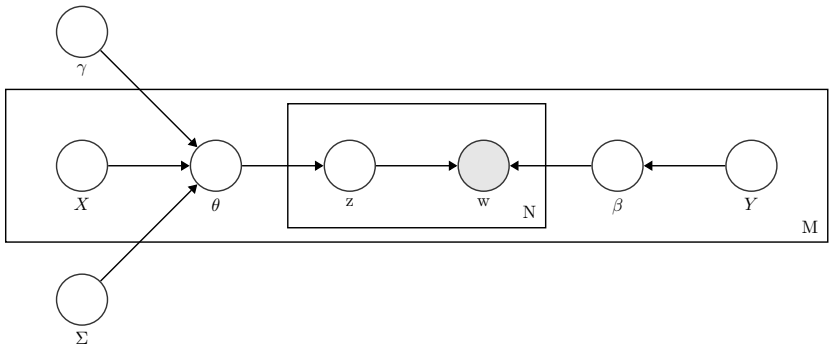


Figure 8.1: Plate diagram for a Structucal Topic Model.

Figure 8.1 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:

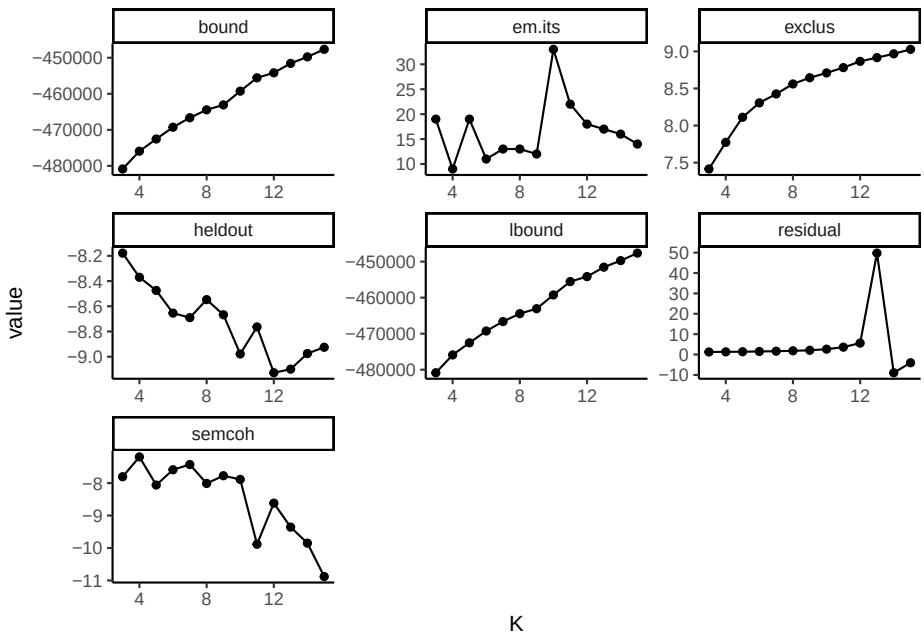
```
k <- c(3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15)
```

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

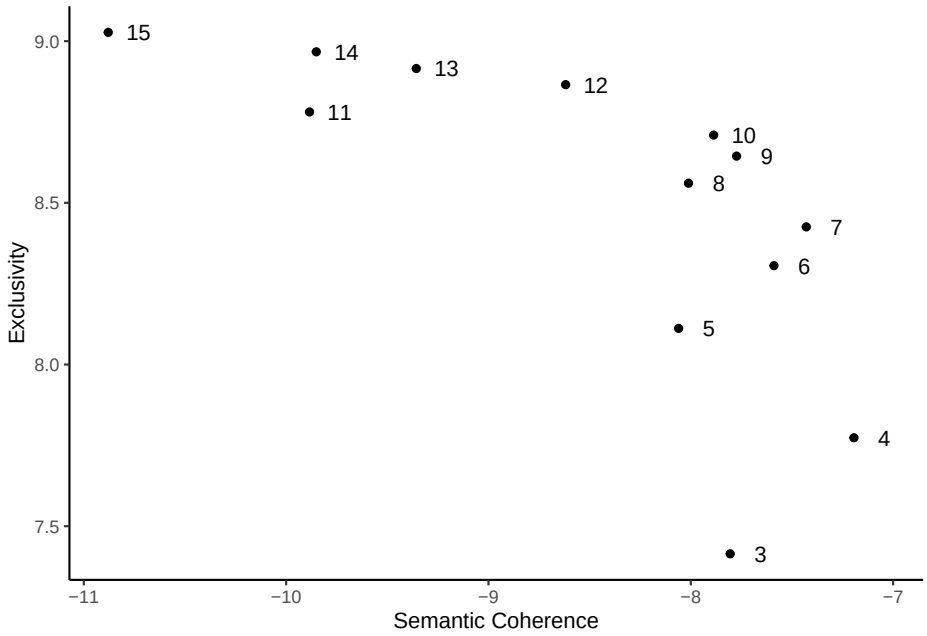
```
findingk_results <- as.data.frame(matrix(unlist(findingk$results), nrow = length(unlist
names <- names(findingk$results)
names(findingk_results) <- names
```

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 10 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 <- 10
output_stm <- stm(documents, vocabulary, K = n_topics, prevalence = ~Party + s(Year),
    data = meta, init.type = "Spectral", verbose = TRUE)
labelTopics(output_stm)
```

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-05)
```

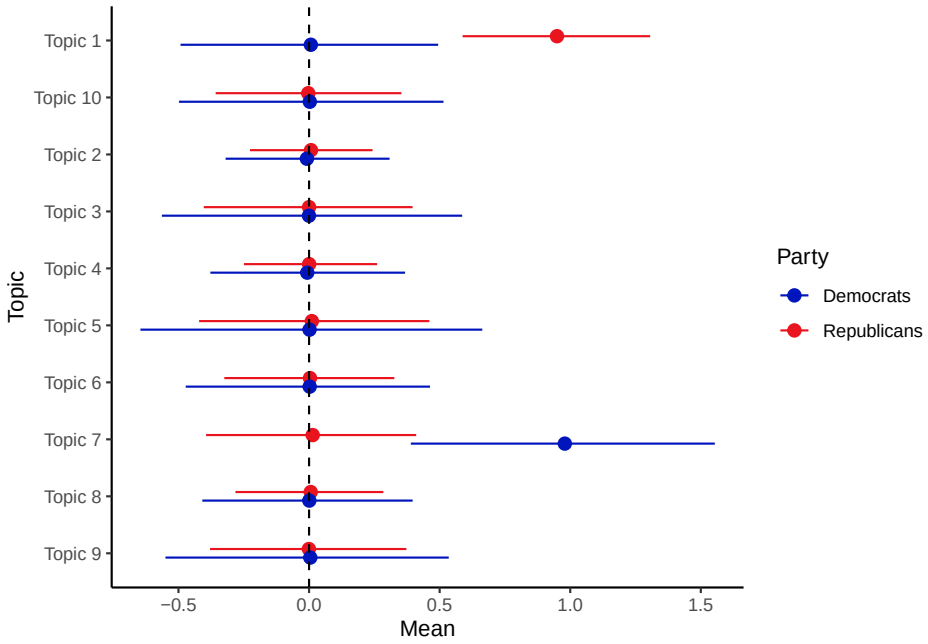
While we can visualise this with the `plot.estimateEffect` option, the visualisation is far from ideal. Thus, let us use some data-wrangling and make the plot ourselves:

```
estimate_data <- plot.estimateEffect(est_assoc_effect, "Party", method = "pointestimate",
    model = output_stm, omit.plot = TRUE)
estimate_graph_means <- estimate_data$means
estimate_graph_means <- data.frame(matrix(unlist(estimate_graph_means), nrow = length(estimate_gr
    byrow = TRUE))
estimate_graph_means <- data.frame(c(rep("Republicans", 10), rep("Democrats", 10)),
    c(estimate_graph_means$X1, estimate_graph_means$X2))

estimate_graph_cis <- estimate_data$cis
estimate_graph_cis <- data.frame(matrix(unlist(estimate_graph_cis), nrow = length(estimate_graph_
    byrow = TRUE))
estimate_graph_cis <- data.frame(c(estimate_graph_cis$X1, estimate_graph_cis$X3),
    c(estimate_graph_cis$X2, estimate_graph_cis$X4))

Topic <- c("Topic 1", "Topic 2", "Topic 3", "Topic 4", "Topic 5", "Topic 6", "Topic 7",
    "Topic 8", "Topic 9", "Topic 10", "Topic 1", "Topic 2", "Topic 3", "Topic 4",
    "Topic 5", "Topic 6", "Topic 7", "Topic 8", "Topic 9", "Topic 10")

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$Topic)))
```

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.

8.4 Latent Semantic Analysis (LSA)

Finally, we will look at Latent Semantic Analysis (LSA), a matrix factorization technique that uses Singular Value Decomposition (SVD) on a term-document matrix to uncover latent semantic structures. Unlike probabilistic models like LDA or STM, LSA is algebraic and assumes that semantically similar terms appear in similar documents.

We'll use the `textmodel_lsa()` function from the `quanteda.textmodels` package to perform LSA on the U.S. Presidential Inaugural Corpus.

Before applying LSA, we need to load the necessary R libraries and prepare the text data:

```
library(quanteda)
library(quanteda.textmodels)
library(dplyr)
library(tidyr)
library(ggplot2)

data("data_corpus_inaugural")
```

```
inaugural_tokens <- tokens(data_corpus_inaugural, remove_punct = TRUE, remove_symbols = TRUE,
    remove_numbers = TRUE) %>%
    tokens_tolower() %>%
    tokens_remove(stopwords("en"))

inaugural_dfm <- dfm(inaugural_tokens)
```

With the data preprocessed and organized into a document-feature matrix, we are ready to fit the LSA model. We use the `textmodel_lsa()` function, specifying the document-feature matrix and the desired number of latent dimensions (`nd`). Again, choosing the optimal number of dimensions often requires experimentation and evaluation, which we will look at in the validation section. Setting a random seed ensures that the results are reproducible.

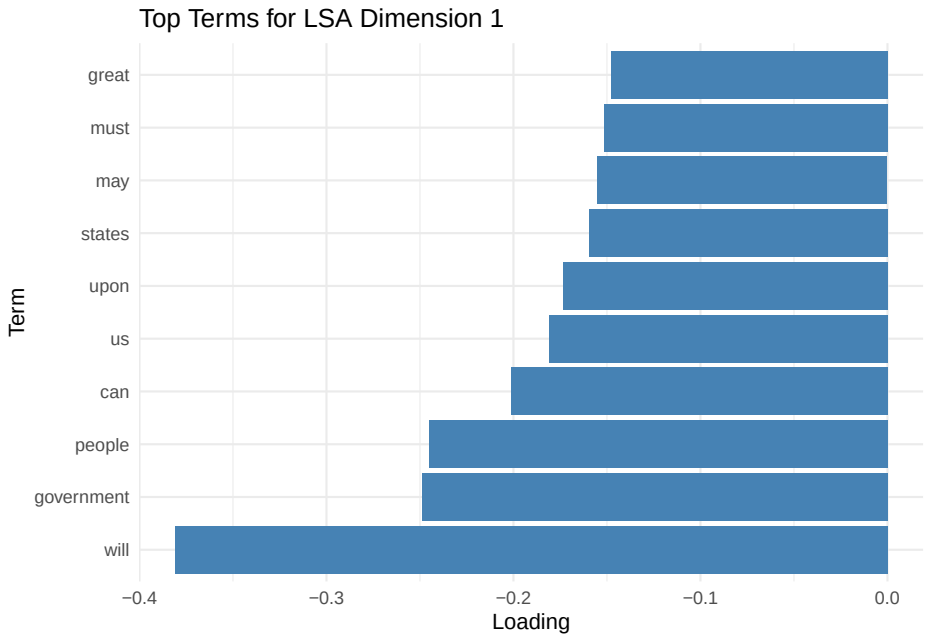
```
set.seed(42)

# Fit LSA with 10 dimensions. The number of dimensions (nd) determines the size
# of the reduced semantic space.
n_dims <- 10
lsa_model <- textmodel_lsa(inaugural_dfm, nd = n_dims)

lsa_model
```

```
##   [[ suppressing 33 column names 'fellow-citizens', 'senate', 'house' ... ]]
```

After fitting the LSA model, we can explore the term loadings. Term loadings indicate the association of each term (word) with each of the discovered latent dimensions. High positive or negative loadings suggest that a term is strongly associated with a particular dimension. By examining the terms with the highest absolute loadings for each dimension, we can begin to interpret the semantic meaning captured by that dimension. We extract the term loadings from the model, convert them into a tidy data frame, and then identify the top terms for each dimension based on the absolute value of their loadings. Visualizing the top terms for a dimension, as shown in the optional plot, can aid in this interpretation.

```
term_matrix <- as.data.frame(lsa_model$features)
colnames(term_matrix) <- paste0("Dim", 1:n_dims) # Rename columns
term_matrix$term <- rownames(term_matrix) # Add the actual terms as a column

term_long <- term_matrix %>%
  pivot_longer(cols = starts_with("Dim"), names_to = "Dimension", values_to = "Loading")

top_terms <- term_long %>%
  group_by(Dimension) %>%
  slice_max(abs(Loading), n = 10, with_ties = FALSE) %>%
  arrange(Dimension, -abs(Loading))
```

```
ggplot(filter(top_terms, Dimension == "Dim1"), aes(x = reorder(term, Loading), y = Loa
  geom_col(fill = "steelblue") + # Create a bar chart with the loading on the x-axis a
  coord_flip() + # Flip the coordinate axes to make term labels easier to read.
  labs(title = "Top Terms for LSA Dimension 1", x = "Term", y = "Loading") + # Add plo
  theme_minimal() # Use a minimal theme for a clean plot.
```



Similar to term loadings, we can explore document loadings. Document loadings represent the association of each document with each latent dimension. Documents with high positive or negative loadings on a particular dimension are estimated to be strongly related to the semantic concept captured by that dimension. By examining the documents with the highest loadings for each dimension, we can see which texts most represent the themes identified by LSA. We extract the document loadings and identify the top documents for each dimension based on their positive loadings.

```
doc_matrix <- as.data.frame(lsa_model$docs)
colnames(doc_matrix) <- paste0("Dim", 1:n_dims)
doc_matrix$document <- docnames(inaugural_dfm)

doc_long <- doc_matrix %>%
  pivot_longer(cols = starts_with("Dim"), names_to = "Dimension", values_to = "Loading"

top_docs <- doc_long %>%
  group_by(Dimension) %>% # Group by dimension to find top documents within each dimen
  slice_max(Loading, n = 5, with_ties = FALSE) %>% # Select the top 5 documents based
```

```
  arrange(Dimension, -Loading) # Arrange the results for better readability.
```

By examining the most positively and negatively associated terms and documents per dimension, we can interpret the latent semantic "concepts" discovered by LSA. Overall, LSA is useful for:

- **Similarity:** Comparing documents based on their LSA vectors.
- **Keywords:** Finding terms similar to a query term in the LSA space.
- **Summarization:** Identifying key sentences representing the document's main dimensions.
- **Reducing dimensionality:** Using the LSA dimensions as features for subsequent supervised learning tasks.

So, how do we validate an LSA? As with LDA, our validation assesses how well the reduced-dimensionality space captures the original data structure and semantic relationships. To begin with, we can look at how much variance each dimension explains. This helps determine how many dimensions to keep by looking for an "elbow" point, where additional dimensions offer diminishing returns. The explained variance for each dimension is proportional to the square of its corresponding singular value from the SVD.

```
# Extract singular values from the LSA model.
singular_values <- lsa_model$sk

# Compute the proportion of variance explained by each dimension
explained_variance <- singular_values^2 / sum(singular_values^2)

# Compute the cumulative variance.
cumulative_variance <- cumsum(explained_variance)

variance_df <- data.frame(
  Dimension = seq_along(explained_variance), # Dimension number.
  Explained = explained_variance, # Proportion of variance explained by each dimension.
  Cumulative = cumulative_variance # Cumulative proportion of variance explained.
)
```

We can easily visualize the individual and cumulative variance explained to identify a good cutoff point for dimensionality. This helps select the number of dimensions (`nd`) to use, balancing dimensionality reduction with retaining sufficient information.

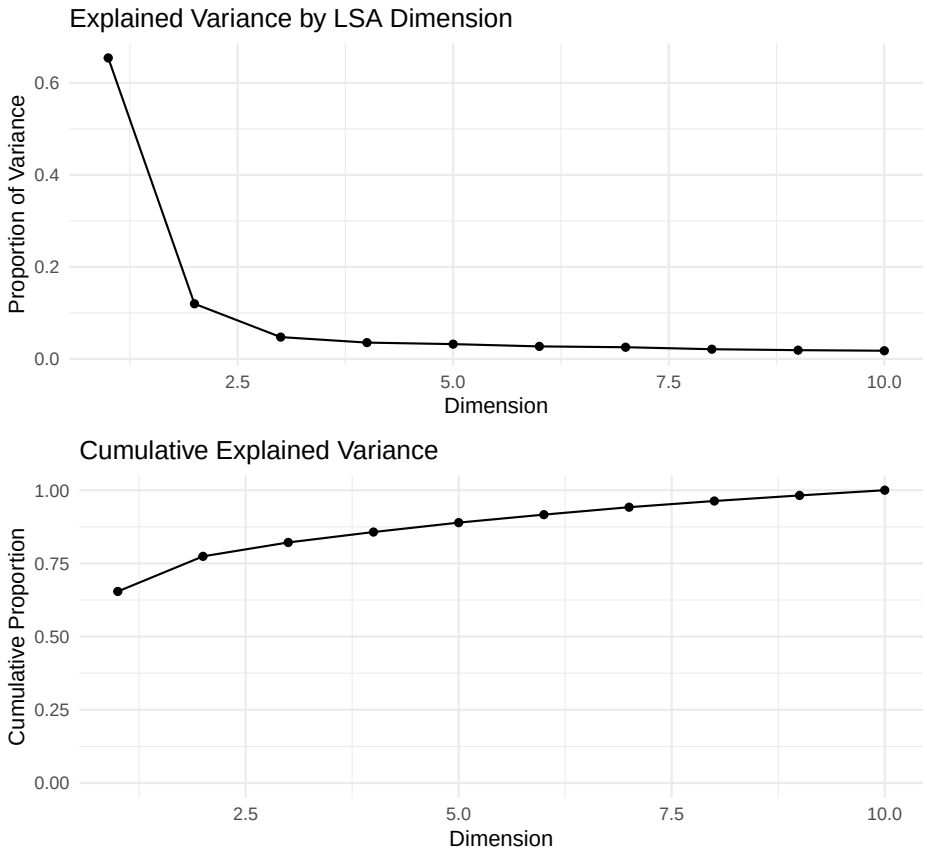
```
library(ggplot2)
library(gridExtra)
```

```
##
## Caricamento pacchetto: 'gridExtra'
```

```
## Il seguente oggetto è mascherato da 'package:dplyr':
```

```
##
##     combine
```

```r
p1 <- ggplot(variance_df, aes(x = Dimension, y = Explained)) +
  geom_line() + geom_point() + # Add lines and points.
  labs(title = "Explained Variance by LSA Dimension", # Add title and labels.
       x = "Dimension", y = "Proportion of Variance") +
  theme_minimal()

p2 <- ggplot(variance_df, aes(x = Dimension, y = Cumulative)) +
  geom_line() + geom_point() + # Add lines and points.
  labs(title = "Cumulative Explained Variance", # Add title and labels.
       x = "Dimension", y = "Cumulative Proportion") +
  ylim(0, 1) + # Set y-axis limits from 0 to 1.
  theme_minimal()

# Arrange the plots vertically.
grid.arrange(p1, p2, ncol = 1)
```



These plots help us determine the ideal number of dimensions to retain by identifying where the cumulative curve levels off—a common strategy known as the "elbow method." Choosing the dimensionality is a trade-off between reducing noise and computational complexity versus preserving semantic information.

Another way to validate LSA is to examine whether semantically similar words are close together in the latent space. We compute the cosine similarity between LSA-generated vectors for selected terms. Terms used in similar contexts should have high cosine similarity in the LSA space.

```
library(coop)  # Need to install: install.packages('coop')
```

```
##
## Caricamento pacchetto: 'coop'
```

```
## Il seguente oggetto è mascherato da 'package:quanteda':
##
##     sparsity
```

```
term_vectors <- lsa_model$features

# Choose a set of semantically related terms for validation.
selected_terms <- c("america", "united", "states", "freedom", "liberty", "war", "peace")

selected_term_vectors <- term_vectors[selected_terms, , drop = FALSE]
term_similarity <- cosine(t(selected_term_vectors))
```

We expect related terms (e.g., "freedom" and "liberty") to have high cosine similarity. This indicates that LSA captures semantic relationships effectively. Unrelated terms should have lower cosine similarity.

Similarly, we can validate whether semantically similar documents are close in the latent space. Again, we use cosine similarity on the LSA vectors of selected presidential speeches. Documents that discuss identical themes or are from similar historical periods might be expected to have a higher similarity.

```
doc_vectors <- lsa_model$docs
docnames_all <- docnames(inaugural_dfm)  # Get all document names.

# Select documents of interest by their names for validation.
selected_docs <- c("1789-Washington", "1861-Lincoln", "2001-G.W.Bush", "2009-Obama")
selected_doc_vectors <- doc_vectors[docnames_all %in% selected_docs, , drop = FALSE]

# Compute cosine similarity between selected document vectors.
doc_similarity <- cosine(t(selected_doc_vectors))
```

By interpreting this matrix, we can observe whether documents from similar eras or with similar themes cluster together—an indicator that LSA is capturing meaningful structures. For instance, one might expect speeches from presidents

in closer periods or from the same political party to exhibit higher similarity than those from vastly different eras or parties.

8.5 Exercises

1. Apply standard LDA to a different corpus (e.g. UK party manifestos or movie reviews). Experiment with varying numbers of topics (k) and evaluate the topics qualitatively based on the most frequent words and quantitatively using semantic coherence and perplexity if these are available in the package used. Select an appropriate k and interpret the main topics found.

2. For the LDA model fitted in Exercise 1, visualise the document-topic distributions for a few selected documents. Which documents have the highest proportions of the most interesting topics? Validate your interpretation by examining the content of these documents. As part of your validation process, calculate and compare the log-likelihood or perplexity for models with different numbers of topics.

3. Create a custom dictionary with seed words relevant to a research question, then apply seeded LDA to a suitable corpus. Interpret the seeded topics and examine the dominant topic assignments per document. If available, compare the resulting top terms and document assignments to those from a standard LDA model on the same corpus as a form of validation.

4. Apply STM to a corpus with rich metadata (e.g. a dataset of news articles containing the date and source or a corpus of speeches containing speaker attributes). Select a set of topics using searchK, carefully evaluating the diagnostic plots. Fit the STM model to the prevalence formula, including relevant metadata. Use estimateEffect() to analyse how the metadata affects the prevalence of specific topics, visualising the results and interpreting the significance of the effects as part of the validation process. Examine the 'labelTopics()' output for qualitative validation of the topics.

5. Apply LSA to a corpus using the textmodel_lsa function. Experiment with different numbers of dimensions (nd). Examine the term and document loadings for a selected number of dimensions, then try to interpret the latent concepts discovered by LSA. Calculate and visualise the cumulative explained variance by the dimensions to help select nd.

6. For your LSA model from Exercise 5, calculate and examine the cosine similarity between a few pairs of selected documents or terms that you expect to be similar or dissimilar based on your knowledge of the corpus. Assess whether the LSA similarity scores align with your expectations to validate the results.

7. Research and implement a method to compute topic coherence (e.g. point-

wise mutual information between top words) for an LDA or STM model, in cases where either `searchK` or `labelTopics` do not provide sufficient detail, or where another package is being used. Use this to quantitatively validate your chosen topic models and compare models with different k.

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

Bruinsma, B., & Gemenis, K. (2019). Validating Wordscores: The Promises and Pitfalls of Computational Text Scaling. *Communication Methods and Measures*, *13*(3), 212–227. https://doi.org/10.1080/19312458.2019.1594741

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

Denny, M. J., & Spirling, A. (2018). Text preprocessing for unsupervised learning: Why it matters, when it misleads, and what to do about it. *Political Analysis*, *26*(2), 168–189. https://doi.org/10.1017/pan.2017.44

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

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. (2019). *Content Analysis - An Introduction to Its Methodology* (4th ed.). Sage. https://doi.org/10.4135/9781071878781

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., & 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

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

Neuendorf, K. A. (2016). *The Content Analysis Guidebook* (2nd ed.). Sage. https://doi.org/10.4135/9781071802878

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

Wickham, H., Çetinkaya-Rundel, M., & Grolemund, G. (2023). *R for Data Science* (2nd ed.). https://r4ds.hadley.nz/

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