Accessing and Using Historical Newspaper Data

Yann Ryan

2023-08-17

Table of contents

Welcome!

This book is a guide to accessing and analysing newspaper data, mostly using the programming language R. I hope it is of use to people interested in working with newspaper data but a bit lost on how to get started.

It uses freely-available newspaper data from collections held by the British Library and digitised by the Living with Machines and Heritage Made Digital projects, and aims to focus on, through examples, the kinds of issues and questions one might have as a researcher or student working with newspapers as sources.

Through the book, you'll learn about the key sources of digitised newspapers in the UK (Chapter 2), where to find and download newspaper data (Chapter 8), generate plain text files to work with (Chapter 9), and do everything from simple word statistics (Chapter 10) to building your own machine learning model in Chapter 13.

The book could also form the basis for a course, probably at Masters level. Each chapter ends with a few recommended readings, mostly academic papers, all focused on computational analysis of text, where possible specifically look at work done with or on newspapers. If you'd like to use this as a teacher, you are free to reuse any parts of the publication in anyway you like, to the extent to which I am entitled to grant that licence.

Acknowledgements

I'm hugely grateful to the Living with Machines project for supporting me with a Digital Residency, which gave me the time to write up a new version of this book, and port it to the Quarto format.

I'm also grateful to the British Library for their advice and information.

Contact Me

If you spot any mistakes, would like to give me feedback, or have found any of this book useful, I'd love to hear from you. Feel free to get in touch at y.c.ryan@hum.leidenuniv.nl. You can also post an issue on the book Github repository.

1 Introduction

1.1 Why Newspaper Data?

More and more newspaper data is becoming available for researchers. Most news digitisation projects now carry out Optical Character Recognition and segmentation, meaning that the digitised images are processed so that the text is machine readable, and then divided into articles. It's far from perfect, but it does generate a large amount of data: both the digitised images, and the underlying text and information about the structure.

All in all, it represents a very extensive source of historical text data, one which is ripe for analysis. Newspapers are particularly compelling as evidence because they can be a window into cultures and discourses which are not necessarily represented in other historical sources, such as printed books. The regularity and periodicity of newspapers mean they are a key source for studying events, trends, and patterns in history. To name a few projects, researchers have used newspaper data to understand Victorian jokes, understand the effects of industrialisation, track the meetings of radical groups, trace global information networks, and look at the history of pandemic reporting.

1.2 What is this book?

To a new researcher, working with newspaper data can be daunting. As well as the sheer size of the datasets, digitised newspapers are often confusingly scattered across many collections and repositories and stored in what might seem like—to a newcomer—complicated, even esoteric, formats.

This book aims to demystify some of these issues, and to provide a set of very practical tools and tutorials which should allow someone with little experience to work with newspaper data in a meaningful way. It will also reference many of the exemplary projects and papers which have worked with this kind of material, hopefully to serve as some kind of inspiration.

This book is aimed at researchers, teachers, and and other interested individuals who would like to access and analyse data from newspapers. In this book, the term *newspaper data analysis* is taken to mean approaches which look beyond the close reading of individual digitised newspapers, and instead look at some element of the underlying data at scale. In this book, this analysis primarily means working with the text data derived from processed newspapers

and metadata from collections of newspapers held by libraries and archives, but also associated data such as press directories. Other types of newspaper data, such as images, are important but beyond the scope of this book.

1.2.1 Goals

In short, this book hopes to help you:

- Know what British Library newspaper data is openly available, where it is, and where to look for more coming onstream.
- Understand something of the XML format which make up the Library's current crop of openly available newspapers.
- Have a process for extracting the plain text of the newspaper in a format which is easy to use.
- Have been introduced to a number of tools which are particularly useful for large-scale text mining of huge corpora: n-gram counters, topic modelling, text re-use.
- Understand how the tools can be used to answer some basic historical questions (whether they provide answers, I'll leave for the reader and historian to decide)

1.2.2 Structure

The book is divided into two parts, **sources** and **methods.**

1.2.2.1 Sources

The first section is a series of 'code-free' chapters, which aim to give an overview of available newspaper data, how to access it, and some caveats to watch out for when using it for your own research. It will give brief introductions to some tools made available by the Living with Machines project to download and work with newspapers. This section is suitable for anyone, though in some cases it will require some use of the command line.

1.2.2.2 Methods

The second section is more specific: a series of tutorials using the programming language R to process and analyse newspaper data. These tutorials include examples and fully worked-out code which takes the reader from the 'raw' newspaper data available online, through to downloading, extracting, and analysing the text within it. These tutorials are most suited for researchers who have a little bit of programming experience, and may be useful for teachers of courses in digital humanities or data science.

For this section, it'll be useful to have at least basic experience with the coding language R, and most likely, its very widely-used IDE, R-Studio. The tutorials will assume you have managed to install R and R Studio, and know how to install packages and use it for basic data wrangling. If you want to learn how to use R, R-Studio and the Tidyverse, there are lots of existing resources available. At the same time, the tutorials are entirely self-contained, and if you are careful and willing to very closely follow the steps, you should be able to make them run even without any coding experience.

1.3 Recommended Reading

Each chapter is accompanied by a couple of pieces of recommended reading. These are generally academic articles, related to the computation technique of the chapter. Wherever possible, they will specifically use newspaper data in their methods.

1.4 Contribute

The book has been written using Quarto, which turns pages of code into figures and text, exposing the parts of the code when necessary, and hiding it in other cases. It lives on a GitHub repository, here: and the underlying code can be freely downloaded and re-created or altered. If you'd like to contribute, anything from a few corrections to an entire new chapter, please feel free to get in touch via the Github issues page or just fork the repository and request a merge when you're done.

1.5 Recommended Reading

Nicholson, Bob. "The Digital Turn: Exploring the Methodological Possibilities of Digital Newspaper Archives." *Media History* 19, no. 1 (February 2013): 59–73. https://doi.org/10.1080/13688804.2012.752963.

Part I

Sources

This part of the book deals with sources - mostly explaining and accessing them.

2 Accessing Newspaper Data in the UK

This chapter outlines the major sources of newspaper data available in the UK.

One important thing to note is that the chapter deals with text only, that is, newspaper data in the form of text derived from the images of the printed pages. The images themselves are important and increasingly the object of research in their own right, but this book doesn't focus on them. In some of the cases below, but not all, the images can be accessed alongside the derived text. The newspapers described below are generally made available in a format known as METS/ALTO, which will be explained in more detail in Chapter 5.

The guide is aimed as a brief, practical overview. For more detailed information, I highly recommend checking out the Atlas of Digitised Newspapers project, which currently has very detailed information on the digitised newspapers and data for ten collections worldwide, including descriptions of metadata, standards, and API access.

2.1 What is Available? In a Nutshell:

Most of the available newspaper data in the UK is based on the collections of the British Library (though other libraries do hold significant collections). This is a huge collection, but getting access to it is not straightforward. In a nutshell:

- If your research is more methods-based or doesn't necessarily need full coverage, you could use the freely-available, public domain Heritage Made Digital and Living with Machines titles. These are used in the second section of the book as the basis for the coding tutorials.
- If you would like to do analysis on something slightly more representative, you could contact Gale and ask them to supply you with the JISC Historical Newspaper Collection. Hopefully, these titles will be made freely available in the coming years.
- To work with the largest dataset of newspapers, you would need an agreement with a commercial company, Find My Past.

2.2 British Library Newspapers - The Physical Collection

The British Library holds about 60 million issues, or 450 million pages of newspapers. They now span over 400 years, but the coverage prior to the nineteenth century is very partial (King 2007). Prior to 1800, the collection has serious gaps, and many issues of many titles have simply not survived. This means that only a fraction of what was published has been preserved or collected, and only a fraction of that which has been collected has been digitised.

It's actually surprisingly difficult to know exactly what has been digitised, but a rough count is possible, using the 'newspaper year' as a unit. This is all the issues for one title, for one year. Its not perfect, because a newspaper-year of a weekly looks the same as a newspaper-year for a daily, but it's an easy unit to count. There are currently about 40,000 newspaper-years on the British Newspaper Archive. The entire collection of British and Irish Newspapers is probably about 350,000 newspaper-years.

It's all very well being able to access the *content*, but for the purposes of the kind of things we'd like to do, access to the *data* is needed. The following are the main British Library digitised newspaper sources. The following is organised into two time periods: before and after 1800.

2.3 Pre-1800

Newspapers before 1800 are treated different to later ones and even constitute a different collection in the British Library. First of all, the collection of pre-1800 newspapers is much smaller and much less complete. This is partially because the British Library did not collect newspapers systematically until well into the nineteenth century. Therefore, even though they are out of copyright, there are no large-scale, centralised collection of early newspapers and newsbooks (their precursors).

Close to the entirety of the surviving collection of newspapers (called newsbooks from the seventeenth century, by and large) has been digitised and made available online, through resources such as EEBO, JISC Historical Texts, and Gale Primary Sources. Many universities and libraries will provide access to these resources, allowing you to browse, search and read through the texts.

Getting access to the underlying text data is less straightforward, however. Where there is OCR data of these titles, the age and quality of these texts mean it is not particularly suited to data analysis. Secondly, the images and texts themselves are mostly behind paywalls and there is no easy provision for bulk downloading of either. Two sources of newspapers for those interested in pre-1800 are worth mentioning: 1) the Burney Collection and 2) the Thomason Newsbook Collection.

2.3.1 The Thomason Newsbook Collection

The Thomason Tracts are a collection of pamphlets, periodicals and broadsides held at the British Library. Part of this are approximately 7,200 serials (or periodicals). These were digitised from microfilm scans in the 1990s, and made available online through the resource Early English Books Online (EEBO). A separate project the Text Creation Partnership (EEBO-TCP), created double-keyed transcriptions of many EEBO texts, but, sadly, not including serials (though it does include news pamphlets). These transcribed texts, in TEI-encoded format, are freely available online.

Some smaller corpora of Thomason newsbooks have been created. These include the Lancaster Newsbook Corpus, a collection of several hundred manually-transcribed serials from the 1650s, and 'George Thomason's Newsbooks', which is a corpus of all newsbooks printed in the year 1649, as well as the entire run of a single title, *Mercurius Politicus*. Neither of these may be suited to the kind of large-scale analysis proposed in this book, but they may anyway be of interest.

2.3.2 Burney Collection

The Burney Collection contains about one million pages, from the very earliest newspapers in the seventeenth century to the beginning of the nineteenth, collected by Rev. Charles Burney in the eighteenth century, and purchased by the Library in 1818(Prescott 2018). It's actually a mixture of both Burney's own collection and material inserted afterwards. It was microfilmed in its entirety in the 1970s and imaged in the 90s but because of technological restrictions it wasn't until 2007 when, with Gale, the British Library released the Burney Collection as a digital resource.

It's not generally available as a data download, but the raw OCR would be of limited use anyway. Older OCR for early modern print is not very good, and it was certainly worse ten years ago when the collection was processed. The accuracy of the OCR has been measured, and Prescott (2018) found that the ocr for the Burney newspapers offered character accuracy of 75.6% and word accuracy of 65%.

However, this is still a useful collection for browsing and keyword searching, which can usually be accessed through an institutional subscription to Gale, packaged as the Seventeenth and Eighteenth Century Burney Newspapers Collection.

2.4 After 1800

The British Library newspapers after 1800 is a much larger collection, and from the middle of the century, is quite comprehensive. Some more on the history of the collection is can be found in Chapter 4. A number of resources make digital surrogates of these newspapers available.

2.4.1 JISC Newspaper digitisation projects

Most of the academic projects in the UK which have used newspaper data, from the Political Meetings Mapper, to the Victorian Meme Machine have used the *British Library's 19th Century Newspapers* collection, published by the **Joint Information Systems Committee (JISC).**

2.4.1.1 What is available?

JISC is mostly a collection of large regional titles from across the nineteenth century, which were picked for a variety of reasons, outlined below. There are titles from many major UK towns and cities. Figure Figure 2.1 charts the number of titles per decade: the coverage of the nineteenth century is more weighted towards the end, but this also reflects the increase in the newspaper collection as a whole.

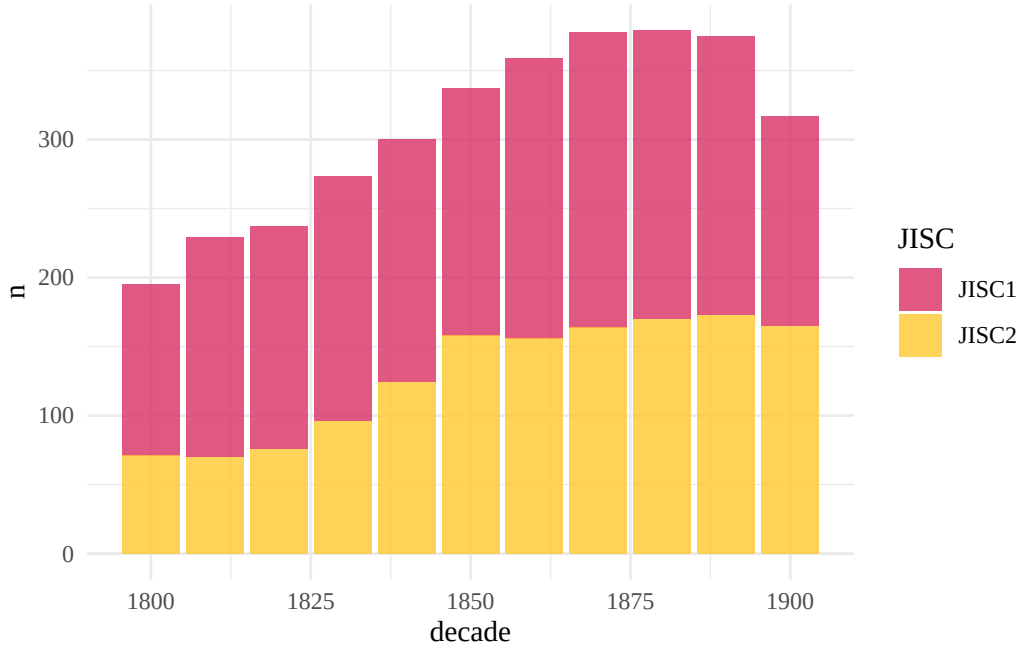


Figure 2.1: Very approximate chart of JISC titles, assuming that we had complete runs for all. Counted by year rather than number of pages digitised.

2.4.1.2 History

The JISC newspaper digitisation program began in 2004, when The British Library received two million pounds from the Joint Information Systems Committee (JISC) to complete a newspaper digitisation project. A plan was made to digitise up to two million pages, across

49 titles.(King 2007) A second phase of the project digitised a further 22 titles.(Shaw 2007, 2005)

2.4.1.3 Coverage

The titles cover England, Scotland, Wales and Ireland, and it should be noted that the latter is underrepresented although it was obviously an integral part of the United Kingdom at the time of the publication of these newspapers - something that's often overlooked in projects using the JISC data. They cover about 40 cities @ref(fig:jisc-points), and are spread across 24 counties within Great Britain @ref(fig:jisc-map), plus Dublin and Belfast. To quote the then curator of the newspapers, Ed King:

> The forty-eight titles chosen represent a very large cross-section of 19th century press and publishing history. Three principles guided the work of the selection panel: firstly, that newspapers from all over the UK would be represented in the database; in practice, this meant selecting a significant regional or city title, from a large number of potential candidate titles. Secondly, the whole of the nineteenth century would be covered; and thirdly, that, once a newspaper title was selected, all of the issues available at the British Library would be digitised. To maximise content, only the last timed edition was digitised. No variant editions were included. Thirdly, once a newspaper was selected, all of its run of issue would be digitised.(King 2008)

Further information on the selection process comes from Jane Shaw, who wrote in 2007 that:

> The academic panel made their selection using the following eligibility criteria:
>
> - To ensure that complete runs of newspapers are scanned
> - To have the most complete date range, 1800-1900, covered by the titles selected
> - To have the greatest UK-wide coverage as possible To include the specialist area of Chartism (many of which are short runs)
> - To consider the coverage of the title: e.g., the London area; a large urban area (e.g., Birmingham); a larger regional/rural area To consider the numbers printed - a large circulation
> - The paper was successful in its time via its sales
> - To consider the different editions for dailies and weeklies and their importance for article inclusion or exclusion To consider special content, e.g., the newspaper espoused a certain political viewpoint (radical/conservative)
> - The paper was influential via its editorials. (Shaw 2007)

The result was a heavily curated collection, which has been scrutinised by historians (Fyfe 2016 for example).

Recently, the Living with Machines project has done the most thorough assessment of the specific biases of the collection. Doing what has been termed an 'environmental scan' of the newspaper collection, and linking it to press directories containing information on geographic coverage, political leanings, and price, that team has shown that it had specific biases in some areas (Beelen et al. 2022).

This is all covered in lots of detail elsewhere, including some really interesting critiques of the access and so forth. Smits (2016) and Mussell (2014) both include some discussion and critique of the British Library Newspaper Collection.

Regardless of its representativeness, it only contains a tiny fraction of the newspaper collection, and by being relevant and restricted to 'important' titles, it does of course miss other voices. For example, much of the Library's collection consists of short runs, and much of it has not been microfilmed, which means it won't have been selected for digitisation. This means that 2019 digitisation selection policies are indirectly *greatly* influenced by microfilm selection policies of earlier decades. Subsequent digitisation projects are trying to rectify these imbalances.

The map below is interactive and shows the locations of the JISC 1 and 2 collections. Clicking on a point will bring up a list of the newspapers digitised in that place; clicking on these links will bring you to the British Library's catalogue page for that title.

2.4.1.4 Access

Currently researchers access this either through Gale, or through the British Library as an external researcher. Many researchers have requested access to the collection through Gale, which they will apparently do in exchange for a 'cost recovery' fee. These files were digitised using a specific format, and the text from them can be extracted using a tool developed by Living with Machines, which I'll explore in a future chapter.

2.4.2 British Newspaper Archive

Most of the British Library's digitised newspaper collection is available on the British Newspaper Archive (BNA). The BNA is a commercial product run by a family history company called FindMyPast. FindMyPast is now responsible for digitising large amounts of the Library's newspapers, mostly through microfilm. As such, they have a very different focus to the JISC digitisation projects. The BNA is constantly growing, and it already dwarfs the JISC projects by number of pages: the BNA currently hosts nearly 70 million pages, against the 3 million or so of the two JISC projects. A recent project, thanks to an agreement between the supplier and the British Library, means that one million pages per year are being made freely available to read through the platform, if you register for a free account.

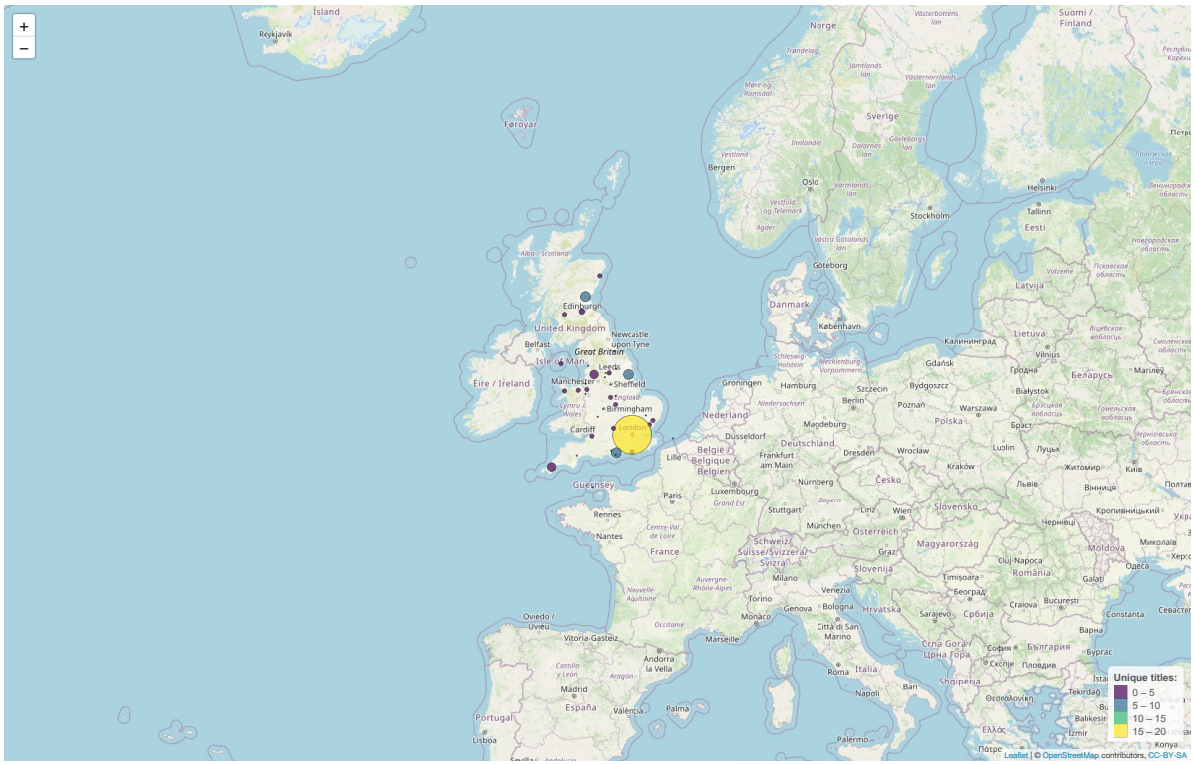


Figure 2.2: Interactive map of the JISC 1 & 2 digitised collections

2.4.2.1 Coverage

The BNA collection is very large, and is particularly focused on local newspapers. At time of writing, the BNA website (linked above) contains over 68 million pages, with more added on a weekly basis.

The titles it contains span the entire nineteenth century, peaking from 1850 - 1900. Figure Figure 2.3 charts the volume of material on the British Newspaper Archive, per year. It is derived from the metadata rather than the actual volume of data. The 'unit' is the *newspaper year*, meaning each year of a newspaper's run is counted as a single data point. It doesn't account for weekly versus daily titles, or the number of pages, but it gives an idea of the temporal shape of the coverage.

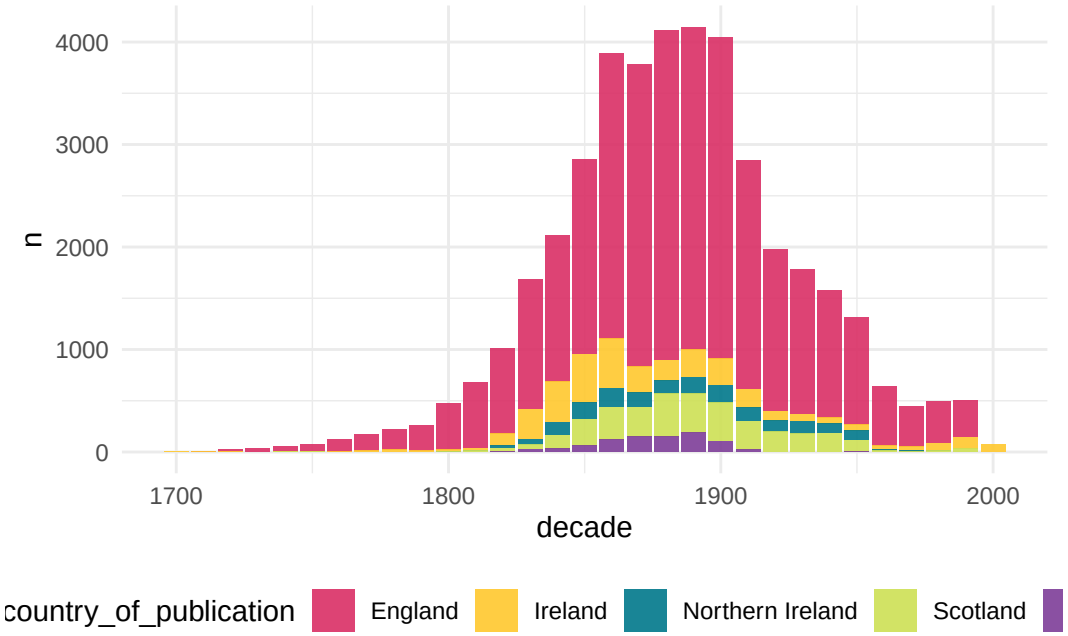


Figure 2.3: Newspaper-years on the British Newspaper Archive. Note that this includes the JISC content too

Geographically, the BNA collection is also widely spread. The most important distinction between it and JISC is that the BNA covers a much wider range of smaller, local titles. Figure Figure 2.4 maps all the titles digitised and available on the British Newspaper Archive. In this case, clicking a link will bring you to the page for that title on the resource. You'll need a subscription (or a free account if they happen to be free-to-view titles) to view it in its entirety.

The map is not complete - titles from Ireland are not shown at the moment because of an issue with the metadata, but will be added in a later update. However the map gives a good sense of the geographic coverage, which is particularly clustered in parts of the Midlands and North,

as well as London. Many smaller villages and towns are represented. Coverage in Wales is still patchy, but most big population centres are represented.

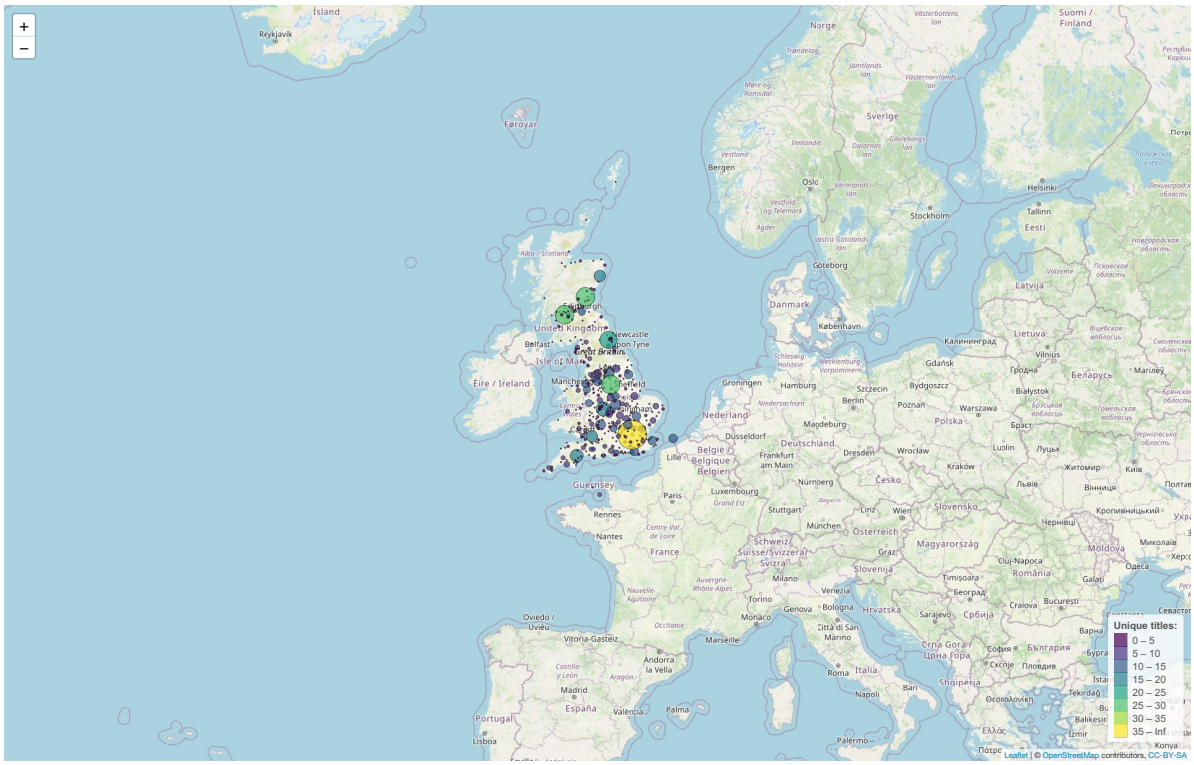


Figure 2.4: Newspaper-years on the British Newspaper Archive. Note that this includes the JISC content too

2.4.2.2 Access

The BNA's primary focus is on providing search access to individual researchers. Sadly, there is little provision made for data access, or even institutional subscriptions for researchers. There are some exceptions: the Bristol N-Gram and named entity datasets project used the BNA data, processing the top phrases and words from about 16 million pages. The collection has tripled in size since then: it's likely that were it to be run again the results would be different.

The Living with Machines project negotiated a full data transfer directly from Find My Past (the owners of the British Newspaper Archive) for the first time. The story of this is written in some detail in the book *Collaborative Historical Research in the Age of Big Data* (Ahnert et al. 2023). In a nutshell, because of copyright issues, the project had to deal directly with Find My Past, and there were a set of detailed restrictions on what could be done with the data, such as having to keep it in secure storage, and delete it after two years.

That is all to say that working with 'all' the UK's available digitised newspaper data is probably not feasible for most individual researchers or even small or medium-sized research projects, at the present time. While hopefully this will change in the future, what it means for now is that most data-driven historical work carried out on newspapers has used the JISC data, rather than the much larger BNA collection, because of relative ease-of-access.

2.4.3 British Library Open Research Repository (bl.iro.bl.uk)

However, there is some positive news in accessing UK newspaper data, thanks to a new resource which has become available in the last couple of years: the Shared Research Repository, a resource maintained by the British Library. Newspaper digitisation projects carried out internally by the British Library do not have the same copyright or commercial restrictions as JISC and the BNA, meaning in theory the data from them can be opely shared with no license restrictions. Two recent projects, Living with Machines and Heritage Made Digital, have carried out digitisation in this way, and so the data can be much more easily and openly shared. It is still a tiny collection in comparison to the entire BNA, but it is a step in the right direction.

The BL open repository is a centralised repository for all sorts of research outputs and data, and the Library has been using it as a storage space for newspaper data from various newspaper digitisation projects. The text data (not the images) have been made freely available, with no license restrictions, meaning they can be used for any purpose whatsoever.

2.4.3.1 Coverage

As of publication, a total of 57 titles, or 437 'newspaper years' are currently available.

These titles currently come from two sources:

- The Living with Machines project
- Heritage Made Digital

Because the information on the motivations behind these projects is somewhat new, more transparent, and easier to access, I'll explain them in a bit more detail.

2.4.3.2 Living with Machines

The Living with Machines project, a collaboration between the Alan Turing Institute and the British Library (as well as other partners), has digitised and made available a new selection of newspapers. According to that project, the aim of the digitisation was research-led rather than curatorial, meaning they chose their titles according to flexible research needs. According to the project, the key aims for the digitisation were to respond to specific research questions,

and to rebalance the bias in the existing newspaper corpus (Tolfo et al. 2022). The eventual selections were a mix of research interests and 'practical factors'.

To help with choosing the most useful and optimal set of newspapers to digitise with limited resources, the team developed a number of tools, including a user interface called *Press Picker*, as well as digitising a set of press directories, which hold information on the newspapers which could then be used in making choices.

The project's historical research interest focused on industrialisation and its impact, and as a result many of the titles digitised are specifically related to that subject, for example by the topic or area of coverage of the newspaper.

2.4.3.3 Heritage Made Digital

Heritage Made Digital was a project within the British Library to digitise up to 1.3 million pages of 19th century newspapers. It has a specific curatorial focus: it picked titles which are completely out of copyright, which means that they all *finished* publication before 1879. It also had preservation aims: because of this, it chose titles which were not on microfilm, and were also in poor or unfit condition. Newspaper volumes in unfit condition cannot be called up by readers: this meant that if a volume is not microfilmed and is in this state, it can't be read by *anyone*.

The other curatorial goal was to focus on 'national' titles. In practice this meant choosing titles printed in London, but without a specific London focus. The JISC digitisation projects focused on regional titles, then local, and all the 'big' nationals like the *Times* or the *Manchester Guardian* have been digitised by their current owners. This means that a group of historically important titles may have fallen through the cracks, and this projects is digitising some of those.[1]

As can be seen in Figure Figure 2.5, the years digitised so far cover most of the century, with the Living with Machines titles focused on the second half, and Heritage Made Digital slightly earlier.

Taken together, the collection spans much of the UK, with an emphasis on London and industrial areas, because of the focus of the projects.

This map in Figure Figure 2.6 displays them. A few are currently missing from the map because the relevant ID numbers to link them to the title list and ultimately, geographic coordinates, were not found. The links in the pop-up take you to the download page for that title on the Open Research Repository.

[1]The term national is debatable, but it's been used to try and distinguish from titles which clearly had a focus on one region. Even this is difficult: regionals would have often had a national focus, and were in any case reprinting many national stories. But their audience would have been primarily in a limited geographical area, unlike many London-based titles, which were printed and sent out across the country, first by train, then the stories themselves by telegraph.

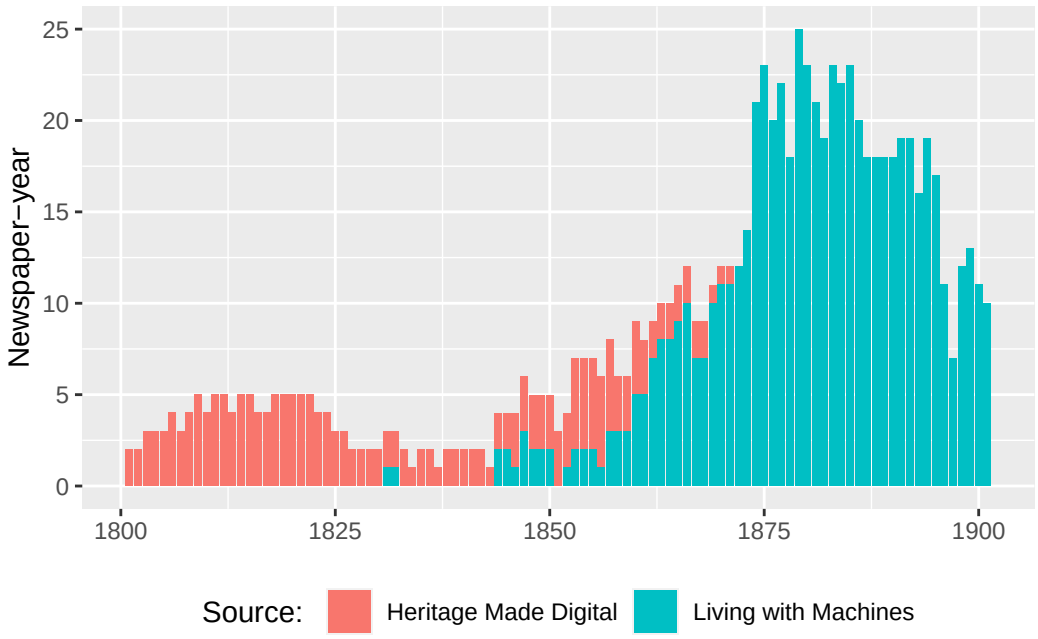


Figure 2.5: Time-series graph of the newspapers digitised by Living with Machines and Heritage Made Digital, and made available on the Open Research Repository.

2.4.3.4 Access

The good news is that as these newspapers are deemed out of copyright, the text data can be made freely downloadable. Currently the first batch of newspapers, in METS/ALTO format, are available on the British Library's Open Repository. They have a CC-0 licence, which means they can be used for any purpose whatsoever. The code examples in the following chapters will use this data, and chapter six will show you how to download the titles using a bulk downloader, instead of going through them one-by-one.

The titles are primarily available as individual downloads through the web interface of the repository. Downloading them is as simple as clicking a link to a .zip file. The easiest way to find all of them is to use the collections organisational system of the repository. The collection British Library News Datasets contains a number of newspaper datasets, including metadata and newspapers. The newspapers themselves are contained with the sub-collection Newspapers. If you click on this second link, you'll see a list of datasets. Each dataset corresponds to a single title (actually, often a group of titles if they changed name or merged). Clicking on a title will display a page containing links to .zip files, one for each year of the title. Note that there may be multiple pages.

Downloading the data one year of a title at a time can be a time-consuming process. In (Chapter 8), I'll demonstrate some methods for downloading titles in bulk.

Figure 2.6: Newspaper-years on the British Newspaper Archive. Note that this includes the JISC content too.

2.4.4 Other data sources

As a final note, there are also a number of really useful metadata files available through the British Library research repository. Firstly, two title-level lists of all British and Irish Newspapers held by the library: one for Britain and Ireland here, and an updated, world-wide one, here. A later chapter Chapter 7 describes these in some more detail.

Most recently, the Living with Machines project has published two digitised press directories, available here and here, plus a dataset of extracted structured data. All of these are have a Public Domain licence.

2.5 Recommended Reading

Beelen, Kaspar, Jon Lawrence, Daniel C S Wilson, and David Beavan. "Bias and Representativeness in Digitized Newspaper Collections: Introducing the Environmental Scan." *Digital Scholarship in the Humanities* 38, no. 1 (April 3, 2023): 1–22. https://doi.org/10.1093/llc/fqac037.

Fyfe, Paul. "An Archaeology of Victorian Newspapers." *Victorian Periodicals Review* 49, no. 4 (2016): 546–77. https://www.jstor.org/stable/26166577.

3 Accessing Newspaper Data Internationally

Many countries have digitised and published parts of their national newspaper collections. In most cases, the newspapers are made available through interfaces designed for search and browsing. Access to the underlying data, for the kinds of methods used in this book, varies greatly across national collections. Some, such as Australia and the U.S. have digitised and made freely available large collections of newspapers, accessible through an API which means they can be easily downloaded or incorporated into third-party applications or resources. Others, such as the UK, are making some data available. Many others do not make any provisions beyond keyword searching or images without OCR.

This chapter is a work-in-progress, and it attempts to survey the existing data provisions made for national newspaper collections. It is not meant as a comprehensive guide to the international digitised newspaper landscape. For a more detailed description of the format, availability, and structure of some key national collections, see the Atlas of Digitised Newspapers.

In a few cases, where title lists have been made available, I have included interactive maps intended as a fun way of seeing at a glance what is included in the collection. More will be added if the correct metadata can be found.

3.1 United States

The Library of Congress in the U.S. sponsored a project called 'Chronicling America', which has created a newspaper dataset and interface which currently has about 16 million pages. All the titles are freely available through the website without a paywall. To access the data itself, the CA database has an API, with instructions here. As with the UK titles, Chronicling America newspapers use the METS/ALTO format. However it may be in a slightly different format and require adjusting the method by which you extract the text from the .xml files.

You can also download all the OCR results directly for each title on this page. Each newspaper contains a list of folders for each issue, and within that can be found a single file for the OCR results (ocr.xml) and a single file for the plain text (ocr.txt).

Chronicling America publish a simple tab-separated-values list of the titles currently in the database here: https://chroniclingamerica.loc.gov/newspapers.txt

We can easily use this information to produce a State-level map of the newspaper titles:

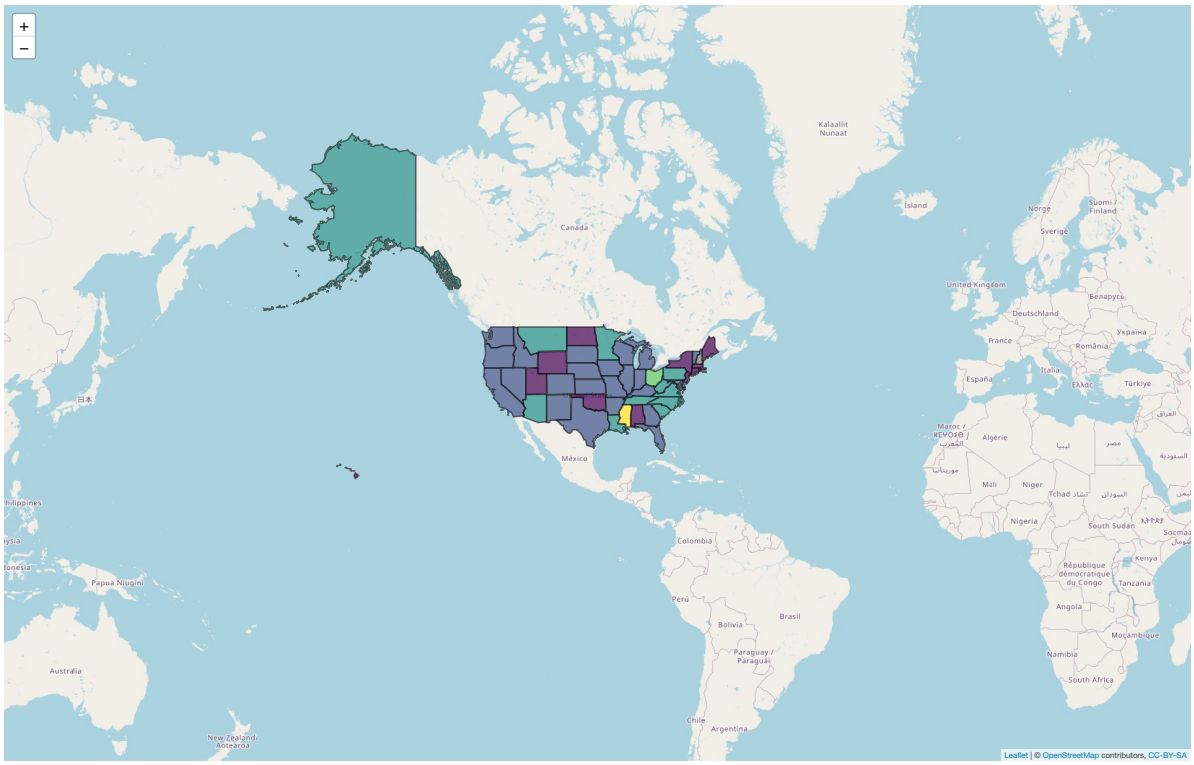
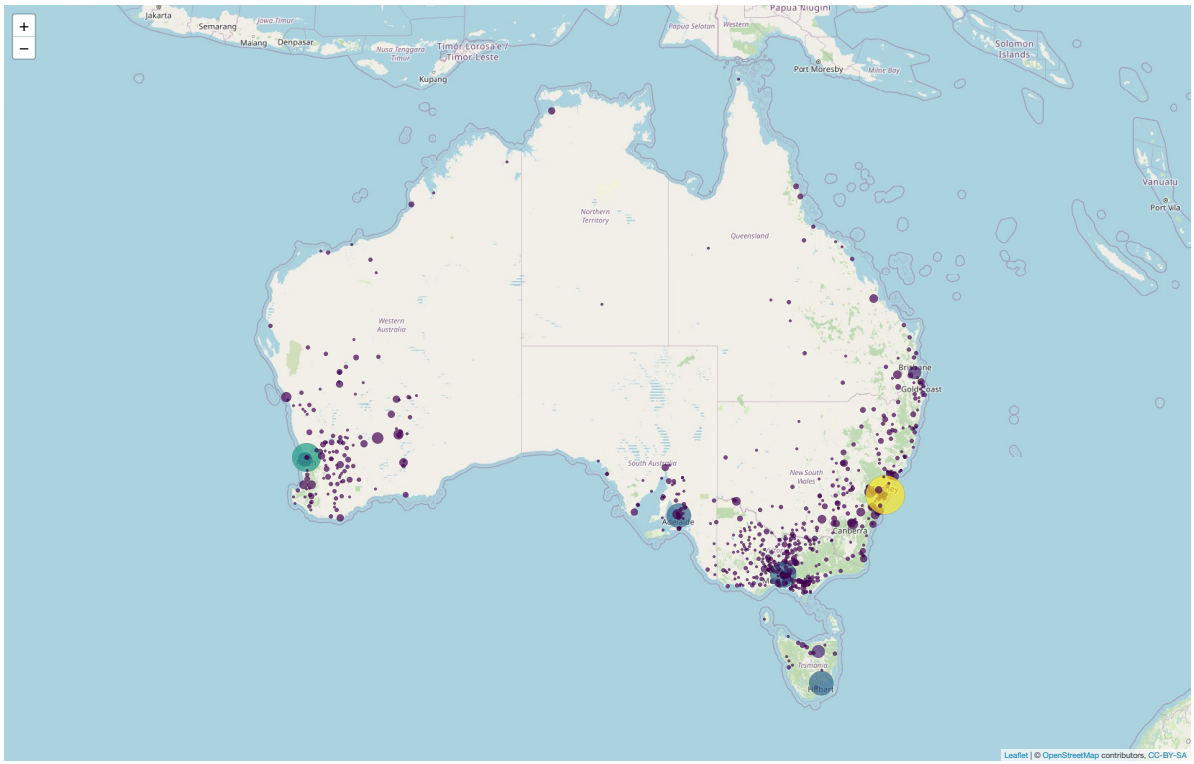


Figure 3.1: Interactive map of Library of Congress Newspapers

3.2 Australia

Trove is a centralised data store of cultural heritage items from the National Library of Australia and other partners. Newspapers published between 1803 and 1955 are freely available through Trove. As well as browsing and searching, Trove has an API which allows you to download newspaper text and image data, and associated metadata. See the documentation for more details. Tim Sherratt publishes a large number of guides to using Trove, including live code tutorials.



3.3 The Netherlands

Historical newspapers in the Netherlands are available through a resource and interface called Delpher, maintained by the Koninklijke Bibliotheek, the Dutch Royal (e.g. National) Library. As well as browsing/searching, users can download in bulk all newspapers published between 1618 and 1879. A full list of the available titles can be found here.

Newspaper data is available in a series of .zip files, and is published in METS/ALTO format. Page scans are not included, but can be retrieved manually from the web using a unique identifier and a URL. A full description of the data format and structure is available on the

Delpher website. There is also an API available: users need to apply directly to Delpher for access. See here.

3.4 Finland

Many of Finland's newspapers have been digitised, and are available through a standard web interface. All the OCR results (not images) of newspapers published between 1771 and 1874 are available as a single bulk download through the Language Bank of Finland. The file is 13GB and the newspaper OCR results are in METS/ALTO format.

3.5 Luxembourg

Luxembourg have digitised and made available about 800,000 pages of digitised newspapers, and made them available through the National Library's Open Data service. The data is presented in a number of different 'packs', each made with different user needs in mind. The data is in METS/ALTO format. The website contains extensive documetation on the format used, and a tool for processing the files can be found on the organisation's Github page.

Helpfully, they also make available a 'text analysis pack' where the plain text has been extracted from the METS/ALTO and made available either in a series of simplified .xml files, or as a .json file with one line per article.

3.6 Pan-European Collections

A number of European projects have worked to make newspapers from multiple countries available through a single repository and interface. Europeana Newspapers makes available, for browsing and keyword searching, about 20 million pages of newspapers from 18 partner libaries. You can view the final report, including links to tools and further information, here.

Another project worth mentioning is Impresso.. Impresso is a database and interface combining newspapers from multiple European countries. The data is not all freely available, but the interface allows for a number of text analysis tasks (such as topic modelling and text reuse) to be carried out on a large corpus, once a non-disclosure agreement has been signed. Users can also create a search query and export the resulting articles as a dataset. A new version is currently in the pipeline.

4 History of the British Library Collection

Newspaper data, meaning the text and images on which we can run computational methods such as those in this book, has a complex and multi-layered history. An excellent example of work peeling back these layers is Fyfe (2016). In this paper, the author explains in detail the process of digitisation, how it is based on previous projects (microfilm), and how this has impacted the makeup of the digitisation landscape today.

We can consider newspaper data as the result of a culmination of layers of objects and processes. At each stage, decisions are made around selection and preservation, which may have been made with long-term goals in mind, but may equally just be the result of contingency, practicality, or even random chance. The key layers might be thought of as follows:

- Newspapers as physical objects
- Newspapers as collection objects held by libraries and archives
- Newspapers as microfilm or other pre-digital copies
- Digitised images
- Derived data (text and articles)
- Interfaces

In this short chapter, I'll go through each of these in turn. There isn't space to give the whole history of newspaper archives, but hopefully this can give a sense of what has been done, and, more importantly, how this might affect your own research.

4.0.1 Newspapers as physical objects

The original source material of newspaper data, is, naturally, newspapers themselves: a term which became ubiquitous in the eighteenth century to describe large-format, regular and periodical items containing printed news. Prior to this other forms of mass-produced news included periodicals such as *newsbooks* (printed news, usually in a smaller format, a term used in the seventeeth century), *newsletters* (hand-written but still somewhat periodical newssheets) and *news pamphlets* (non-periodical texts, usually about a single news item.

The British Library's periodical collection today comes from a few sources, covering different periods and acquired at different times. The earliest significant acquisition of periodicals is

the *Thomason Tracts*, a collection of civil-war-era pamphlets and periodicals collected by the bookseller George Thomason (d. 1666), gifted to the Library in 1762 (Harris 1998, 19). The contents of this has been digitised and are available through Early English Books Online.

The next significant acquisition is the Burney Collection, purchased from the estate of Rev. Charles Burney (1757-1817) in 1818. This is a collection of about 700 bound volumes, within which are about one million pages of seventeenth and eighteenth century newspapers, across 1,290 titles (Prescott 2018). This is the largest single collection of pre-1800 newspapers, and formed the basis for the Library's collection of early periodicals. In fact, when additional material pre-1800 was later acquired by the Library, it was inserted into the Burney Collection retroactively. The entire Burney collection has been digitised and is available through Gale Cengage.

As the volume of newspaper printing increased, there was a need to collect material straight from the firehose, so to speak. In 1822, the Library arranged for the Stamp Office, which kept a copy of all newly-printed newspapers for two years after publication, to send the copies to the Library once that period was up. This meant that from then on, the Library's newspaper collection increased substantially.

This does not mean, however, that all material we would consider newspapers was collected from then onward. Many publications deliberately changed their format so that they would not be counted under the Stamp Act (and therefore taxed), and, as a result, were not collected and sent to the Library. Additionally, once legal deposit came into force, only one edition of each was sent: many papers had multiple editions (such as evening versions), not all of which have been preserved.

This practice continued until 1869, when a new mechanism, Legal Deposit, came into force. This meant that every newspaper publication was, by law, compelled to send a copy of each issue directly to the Library. From this date onwards, the newspaper collection of the British Library is pretty comprehensive, with an important caveat that only one edition per issue was generally sent (meaning evening editions, which might be substantially different, were not always kept).

Naturally, these newspapers were not collected in the form they were originally distributed to customers: unbound and folded sheets of paper. Instead, all of these collections (Thomason, Burney, Stamp Act Newpapers, Legal Deposit) come into the Library as *bound volumes*: hard-backed, folio sized volumes, each containing, usually, one year's worth of issues for a single title (though not always, as this blog post explains). It is worth bearing this in mind because these volumes, and the way titles have been bound within them, have influenced the way that newspapers have been microfilmed and digitised.

The newspapers themselves continue to be collected by legal deposit, but the story of the collection doesn't stop there. The huge volume of newspaper material collected meant that a new repository was built on the outskirts of London, near Hendon, in 1905. All newspapers after 1801 were intended to be kept there. However, it was soon realised that this was still not enough space and in 1932, on the same site, the 'Colindale Newspaper Library' opened.

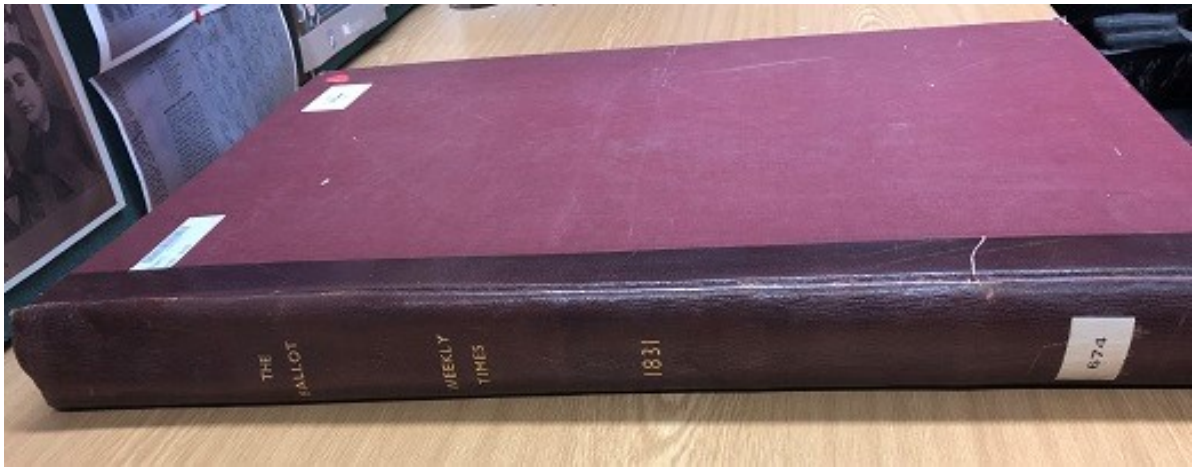


Figure 4.1: A single newspaper volume. From https://blogs.bl.uk/thenewsroom/2019/01/multi-title-newspaper-volumes-historic-practice-vs-modern-process.html, copyright unknown.

Finally, Colindale itself was closed in 2013, and the entire newspaper collection (post-1801) was moved to a custom-built facility in Boston Spa, Yorkshire, where it is housed today, in a facility which is temperature-controlled and where newspaper volumes are retrieved by robots.

4.0.2 From Newspaper to Microfilm

Many of these volumes were subsequently put on microfilm. This began in 1948, and by 1951, about 75% of the newspapers published from between 1801 and 1820 had been microfilmed, though the process of microfilmed historic newspapers slowed down to concentrate on newly-acquired ones (Harris 1998, 602). Large-scale microfilm of the post-1801 collection resumed in the 1960s, and continued right up until the 1990s (Fyfe 2016, 559). Other collections of periodicals were microfilmed around the same time: first, what would become Early English Books Online , and between 1977 and 1979, the entire Burney collection was microfilmed because of worries about its condition (Prescott 2018).

It is these microfilms which form the basis for much of the Library's digitised newspapers. This intermediate step of microfilm before digitisation has impacted how the latter has been done: Paul Fyfe notes, for example, that the way the microfilm scanner cradle held open the volumes of newspapers may have influenced the choice of the single page, rather than a two-page spread, as the basic unit of digitisation (Fyfe 2016, 557).

Figure 4.2: The now-defunct Newspaper Library in Colindale, North London. Attribution: Caroline Ford, CC BY-SA 3.0 <http://creativecommons.org/licenses/by-sa/3.0/>, via Wikimedia Commons

Figure 4.3: National Newspaper Building The British Library's National Newspaper Building, Boston Spa, Yorkshire. Attribution: Luke McKernan, CC BY-SA 2.0, via Flickr

4.0.3 From Microfilm (and newspaper) to Digital Image

The next stage in this journey is the process by which digital copies of these newspapers were created. Again, this has a complex and multi-layered history. The first set of projects, JISC 1 and JISC 2, began in the early 2000s. The digitisation involved scanning newspapers from rolls of microfilm - in many cases new microfilm was created before the process of scanning, according to Fyfe (2016). The result of this were high-quality TIFF or JPEG2 images, suitable for preservation and storage.

4.0.4 From Digital Image to Text Data

The next step in digitisation is processing the images so that the pixels which make up the text within them can be turned into a machine-readable form. In newspapers, this is made more difficult by the fact that newspapers have complex structures - the text does not run from left to right in most cases, but is separated into columns and individual articles. Therefore, a process of segmentation must be carried out before anything else. Both of these steps are still very error-prone.

4.0.5 OCR/OLR

The next layer is the results of the OLR (Optical Layout Recognition, which segments newspapers into articles and columns) and OCR (Optical Character Recognition, which turns the pixels which make up the text into a machine-readable form). Roughly speaking, a newspaper is scanned or photographed, and the resulting digital image is run through OLR and OCR software, such as ABBYY Finereader, or the open-source Tesseract. This generally produces a new set of files which contain the layout and text. These files are usually in XML format, and contain not only the text but hierarchical tags describing the various sections, headers, and even the coordinates of the words on the page.

Figure 4.4: Front page of The Sun Newspaper, January 1, 1850. The paper's seven columns and small text make it a difficult challenge for automatic segmentation. Public domain image, hosted on the British Newspaper Archive <https://www.britishnewspaperarchive.co.uk/viewer/BL/0002194/18500101/001/0001>

Both of these processes are notoriously error-prone and inconsistent. Much of this is due to the fact that the conditions and layout of newspapers is not consistent. Many have different fonts, complicated headers, and in many cases, the digitised images on which the OCR is based contain faults, from the microfilm or scanning, or from problems with the original physical item. However, even with lower-quality OCR, many types of text analysis can still produce meaningful results (see the recommended reading).

4.0.5.1 METS/ALTO

The industry-standard for these files is a format known as METS/ALTO. This has been used for all the major digitisation projects *after* JISC, which used a different XML standard. These are the files you are likely to find yourself with at the beginning stage of working with newspaper data.

METS/ALTO consists of two parts: first a set of ALTO files, one for each page of an issue of a newspaper. These ALTO pages contain each word derived from the OCR, along with coordinates which tell where it appears on the page, divided into 'text blocks' and 'text lines', as well as some basic metadata, which might change depending on the specifics of the software used. Each issue also has a single METS file, which contains information on how the lines and blocks in the ALTO files should be knitted together to recreate the newspaper.

Chapter 5 will go into a little more detail, and provide some information on where you can find resources to work with these METS/ALTO files. A tutorial in the second part will show how you can extract the text and save it separately, using R.

4.1 Conclusion

As this is meant as a practical guide,it is worth reflecting on how this may effect your own experience of working with newspaper data. First of all, the complex history of the collection means that depending on the time period and the type of material, it may be available from very different sources, or in very different formats. It is possible that material of interest is spread across multiple collections, and that some of it has been digitised and others not, often for arbitrary reasons. A second takeaway is that these collection, microfilm, and digitisation practices have a huge effect on the resulting newspaper data available today, and are worth understanding before undertaking any research project which hopes to use newspaper data in any kind of representative way.

4.2 Recommended Reading

Torget, Andrew J. "Mapping Texts: Examining the Effects of OCR Noise on Historical Newspaper Collections." In *Digitised Newspapers – A New Eldorado for Historians?*, edited by Estelle Bunout, Maud Ehrmann, and Frédéric Clavert, 47–66. De Gruyter, 2022. https://doi.org/10.1515/9783110729214-003.

Van Strien, Daniel, Kaspar Beelen, Mariona Ardanuy, Kasra Hosseini, Barbara McGillivray, and Giovanni Colavizza. "Assessing the Impact of OCR Quality on Downstream NLP Tasks:" In *Proceedings of the 12th International Conference on Agents and Artificial Intelligence*, 484–96. Valletta, Malta: SCITEPRESS - Science and Technology Publications, 2020. https://doi.org/10.5220/0009169004840496.

Why You (A Humanist) Should Care About Optical Character Recognition

5 Working with METS/ALTO

As mentioned in the last chapter, most of the 'newspaper data' produced by various projects is an output of XML files, usually with information on the location on the page of each word, and further metadata.

The most common standard used is called METS/ALTO, a standard maintained by the Library of Congress for digitising complex documents such as newspapers. Generally, an issue of a digitised newspaper will consist of two components:

- A series of ALTO files, one for each page of the issue.
- A single METS file for each issue

In this chapter I'll give a brief overview of the standard, and provide some information on how it might be used. Ultimately, much of the standard digital humanities analyses will use some version of the 'plain text', and requires extracting that from the very complex structure of the original documents (though there are in theory ways the full METS/ALTO information might be useful, perhaps for understanding how physical layout relates to text or images, for example).

For more information on the METS/ALTO standards, take a look at the Library of Congress information pages for ALTO and for METS.

Put simply, ALTO documents record the physical coordinates of text on a page, generally divided into text blocks, text lines, and words. Each part of the text is tagged in a hierarchical tagging system, as can be seen from the screenshot below. The ALTO document also records some basic metadata about the process, such as the predicted word accuracy (which is a measure of confidence of the software, and doesn't compare to a 'ground truth', so it may not be accurate), and the source image. You can see below, that the whole page is contained within a `<Layout></Layout>` tag, and within that are contained `<TextBlock></Textblock>`, within that, `<Textline></Textline>`, and finally `<String></String>` tags for each word of text.

On the other hand, the METS file takes the IDs of these strings, textblocks, and textlines, and provides information on how they all fit together to make the structure of the newspaper. Generally, newspapers will be divided into different articles, and contain other items such as headers and titles. This information is recorded by the METS file. It also means that we can extract and differentiate between different articles when it comes to extracting the text.

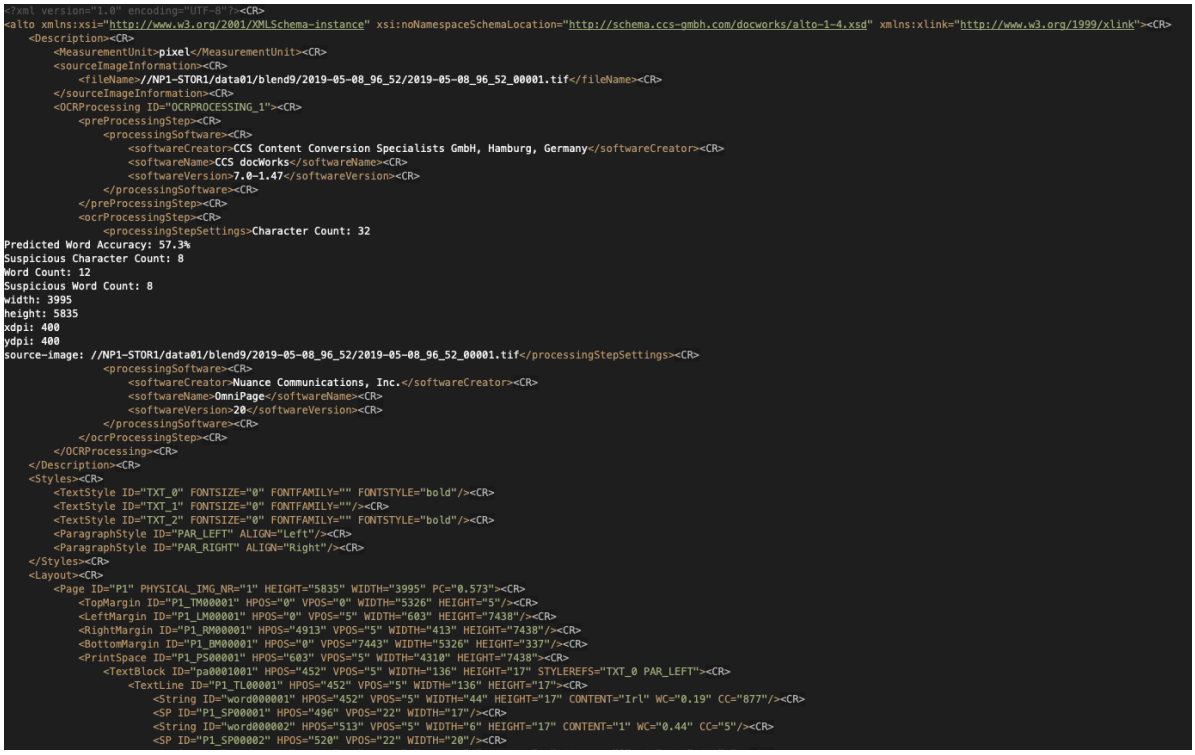


Figure 5.1: Screenshot showing a page of a newspaper in ALTO format.

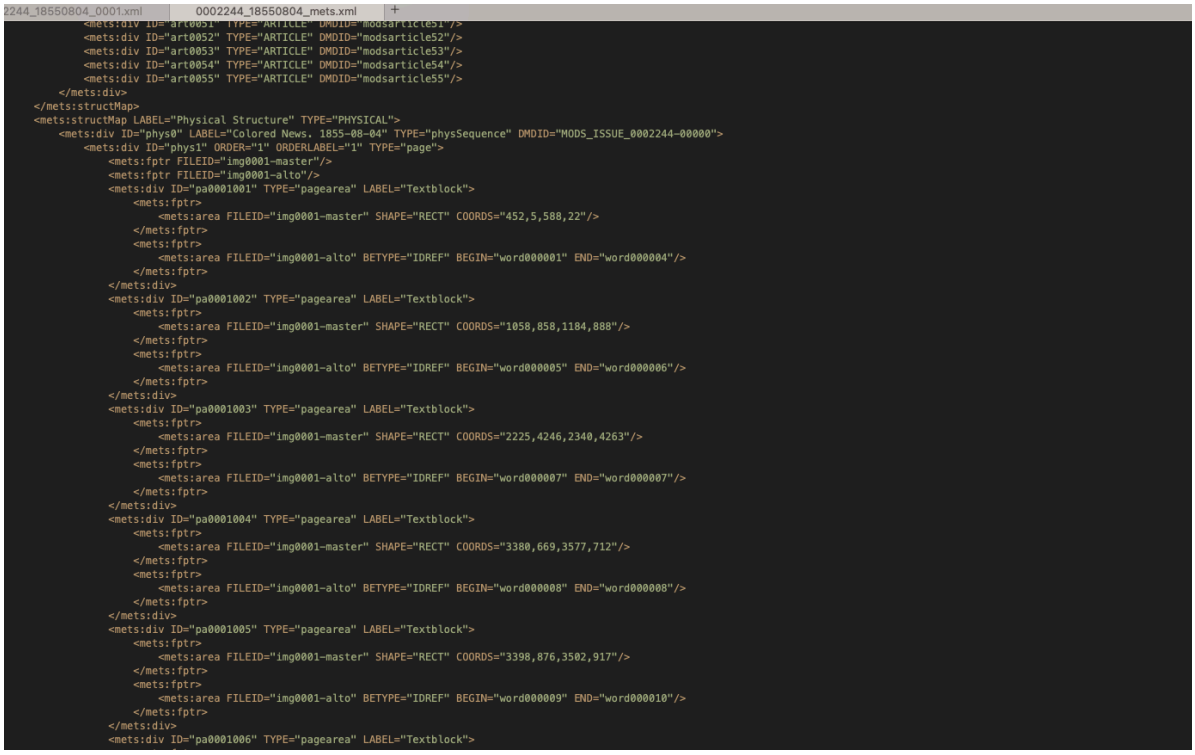


Figure 5.2: Screenshot showing the METS file for a newspaper issue

5.1 How to work with these

The structure of these files can be daunting, and it's true that it is not straightforward to extract the text in any meaningful way. In many cases, you'll find that others have already created plain-text versions of newspapers, or tools to make them yourself, such as the resources created by the National Library of Luxembourg, the Impresso project, and Living with Machines. The latter, alto2text, is a command line tool which converts METS/ALTO into plain text.

The full documentation for alto2text is available via Github. It uses the python programming language, and expects the newspapers the folder format in which they can be downloaded from the Shared Research Repository.

5.1.1 Using R

In Chapter 9, you'll find a series of steps in order to extract text from METS/ALTO. It's much slower and a little clunkier than the method above, but it may be an alternative if you want to do everything within a single coding language.

5.2 Recommended Reading

Making sense of the METS and ALTO XML standards

Part II

Methods

This part of the book is a series of practical tutorials, written in R, which teach how to download, extract, and analyse newspaper data, using openly available newspaper data from the Shared Research Repository, digitised by Heritage Made Digital and Living with Machines.

6 Using R and the tidyverse

R is a programming language, much like Python. It is widely used by data scientists, those using digital humanities techniques, and in the social sciences.

It has some advantages for a novice to programming. Its widely-used platform (called an Integrated Programming Environment- an IDE) for the language called R-Studio makes it relatively easy for beginners. A lot of this is because of developers who have extended the functionality of the base language greatly, particularly a suite of extra functions known collectively as the 'tidyverse'.

The methods section of this book may be challenging if you are a complete beginner to R and have not learned any other programming language. If you'd like to get started with R and the tidyverse in earnest, the canonical texts is R for Data Science, written by Hadley Wickham.

6.1 Getting started

The only requirements to get through these tutorials are to install R and R-Studio, as well as some data which needs to be downloaded separately.

6.1.1 Download R and R-Studio

R and R-Studio are two separate things. R will work without R-studio, but not the other way around, and so it should be downloaded first. Go to the download page, select a download mirror, and download the correct version for your operating system. Follow the installation instructions if you get stuck.

Next, download R-Studio. You'll want the desktop version and again, download the correct version and follow the instructions. When both of these are installed, open R-Studio, and it should run the underlying software R automatically.

At this point, I would highly recommend reading a beginners guide to R and R-studio, such as this one, to familiarise yourself with the layout and some of the basic functionality of R-Studio. Once you understand where to type and save code, where your files and dataframes live, and how to import data from spreadsheets, you should be good to start experimenting with newspaper data.

R relies on lots of additional packages for full functionality, and you'll need to install these by using the function `install.packages()`, followed by the package name, in inverted commas. I recommend doing this to install the tidyverse suite of packages by running `install.packages('tidyverse')` in the Console window (the bottom-left of the screen in R-Studio) as you'll end up using this all the time.

6.2 Using R

6.2.1 'Base' R.

Commands using R without needing any additional packages are often called 'base' R. Here are some important ones to know:

You can assign a value to an object using `=` or `<-`:

```
x = 1

y <- 4
```

Entering the name of a variable in the console and pressing return will return that value in the console. The same will happen if you enter it in a notebook cell (like here below), and run the cell. This is also true of any R object, such as a dataframe, vector, or list.

```
y
```

```
[1] 4
```

You can do basic calculations with `+`, `-`, `*` and `/`:

```
x = 1+1

y = 4 - 2

z = x * y

z
```

```
[1] 4
```

You can compare numbers or variables using `==` (equals), `>` (greater than), `<`, (less than) `!=` (not equal to). These return either `TRUE` or `FALSE`:

```
1 == 1
```

```
[1] TRUE
```

```
x > y
```

```
[1] FALSE
```

```
x != z
```

```
[1] TRUE
```

6.2.2 Basic R data structures

It is worth understanding the main types of data that you'll come across, in your environment window.

A variable is a piece of data stored with a name, which can then be used for various purposes. The simplest of these are single **elements**, such as a number:

```
x = 1

x
```

```
[1] 1
```

Next is a vector. A vector is a list of **elements**. A vector is created with the command `c()`, with each item in the vector placed between the brackets, and followed by a comma. If your vector is a vector of words, the words need to be in inverted commas or quotation marks.

```
fruit = c("apples", "bananas", "oranges", "apples")
colour = c("green", "yellow", "orange", "red")
amount = c(2,5,10,8)
```

Next are dataframes. These are the spreadsheet-like objects, with rows and columns, which you'll use in most analyses.

You can create a dataframe using the `data.frame()` command. You just need to pass the function each of your vectors, which will become your columns.

We can also use the `glimpse()` or `str()` commands to view some basic information on the dataframe (particularly useful with longer data).

```
fruit_data = data.frame(fruit, colour, amount, stringsAsFactors = FALSE)

str(fruit_data)
```

```
'data.frame':   4 obs. of  3 variables:
 $ fruit : chr  "apples" "bananas" "oranges" "apples"
 $ colour: chr  "green" "yellow" "orange" "red"
 $ amount: num  2 5 10 8
```

6.2.3 Data types

Notice that to the right of the third column, the amount, has `<dbl>`under it, whereas the other two have ‘. That’s because R is treating the third as a number and others as a string of characters. It’s often important to know which data type your data is in: you can’t do arithmetic on characters, for example. R has 6 data types:

- character
- numeric (real or decimal)
- integer
- logical
- complex
- Raw

The most commonly-used ones you’ll come across are `character`, `numeric`, and `logical`. `logical` is data which is either `TRUE` or `FALSE`. In R, all the items in a vector are *coerced* to the same type. So if you try to make a vector with a combination of numbers and strings, the numbers will be converted to strings, as in the example below:

```
fruit = c("apples", 5, "oranges", 3)

str(fruit)
```

```
 chr [1:4] "apples" "5" "oranges" "3"
```

6.2.4 Installing and loading packages:

R is extended through the use of 'packages': pre-made sets of functions, usually with a particular task or theme in mind. To work with networks, for example, we'll use a set of third-party packages. If you complete the exercises using the CSC cloud notebooks, these are already installed for you in most cases. To install a package, use the command `install.packages()`, and include the package name within quotation marks:

```
install.packages('igraph')
```

To load a package, use the command `library()`. This time, the package name is not within quotation marks

```
library(igraph)
```

```
Warning: package 'igraph' was built under R version 4.0.5
```

```
Attaching package: 'igraph'
```

```
The following objects are masked from 'package:stats':

    decompose, spectrum
```

```
The following object is masked from 'package:base':

    union
```

6.3 Tidyverse

Most of the work in these notebooks is done using a set of packages developed for R called the 'tidyverse'. These enhance and improve a large range of R functions, with a more intuitive nicer syntax. It's really a bunch of individual packages for sorting, filtering and plotting data frames. They can be divided into a number of different categories.

All these functions work in the same way. The first argument is the thing you want to operate on. This is nearly always a data frame. After come other arguments, which are often specific columns, or certain variables you want to do something with.

```
library(tidyverse)
```

Here are a couple of the most important ones

6.3.1 select(), pull()

`select()` allows you to select columns. You can use names or numbers to pick the columns, and you can use a `-` sign to select everything *but* a given column.

Using the fruit data frame we created above: We can select just the fruit and colour columns:

```
select(fruit_data, fruit, colour)
```

```
    fruit colour
1  apples  green
2 bananas yellow
3 oranges orange
4  apples    red
```

Select everything but the colour column:

```
select(fruit_data, -colour)
```

```
    fruit amount
1  apples      2
2 bananas      5
3 oranges     10
4  apples      8
```

Select the first two columns:

```
select(fruit_data, 1:2)
```

```
    fruit colour
1  apples  green
2 bananas yellow
3 oranges orange
4  apples    red
```

6.3.2 group_by(), tally(), summarise()

The next group of functions group things together and count them. Sounds boring but you would be amazed by how much of data science just seems to be doing those two things in various combinations.

`group_by()` puts rows with the same value in a column of your dataframe into a group. Once they're in a group, you can count them or summarise them by another variable.

First you need to create a new dataframe with the grouped fruit.

```
grouped_fruit = group_by(fruit_data, fruit)
```

Next we use `tally()`. This counts all the instances of each fruit group.

```
tally(grouped_fruit)
```

```
# A tibble: 3 x 2
  fruit       n
  <chr>   <int>
1 apples      2
2 bananas     1
3 oranges     1
```

See? Now the apples are grouped together rather than being two separate rows, and there's a new column called `n`, which contains the result of the count.

If we specify that we want to count by something else, we can add that in as a 'weight', by adding `wt =` as an argument in the function.

```
tally(grouped_fruit, wt = amount)
```

```
# A tibble: 3 x 2
  fruit       n
  <chr>   <dbl>
1 apples     10
2 bananas     5
3 oranges    10
```

That counts the amounts of each fruit, ignoring the colour.

6.3.3 filter()

Another quite obviously useful function. This filters the dataframe based on a condition which you set within the function. The first argument is the data to be filtered. The second is a condition (or multiple condition). The function will return every row where that condition is true.

Just red fruit:

```
filter(fruit_data, colour == 'red')
```

```
   fruit colour amount
1 apples    red      8
```

Just fruit with at least 5 pieces:

```
filter(fruit_data, amount >=5)
```

```
    fruit colour amount
1 bananas yellow      5
2 oranges orange     10
3  apples    red      8
```

You can also filter with multiple terms by using a vector (as above), and the special command `%in%`:

```
filter(fruit_data, colour %in% c('red', 'green'))
```

```
   fruit colour amount
1 apples  green      2
2 apples    red      8
```

6.3.4 slice_max(), slice_min()

These functions return the top or bottom number of rows, ordered by the data in a particular column.

```
fruit_data %>% slice_max(order_by = amount, n = 1)
```

```
    fruit colour amount
1 oranges orange     10
```

```
fruit_data %>% slice_min(order_by = amount, n = 1)
```

```
   fruit colour amount
1 apples  green      2
```

These can also be used with `group_by()`, to give the top rows for each group:

```
fruit_data %>% group_by(fruit) %>% slice_max(order_by = amount, n  =  1)
```

```
# A tibble: 3 x 3
# Groups:   fruit [3]
  fruit   colour amount
  <chr>   <chr>   <dbl>
1 apples  red         8
2 bananas yellow      5
3 oranges orange     10
```

Notice it has kept only one row per fruit type, meaning it has kept only the apple row with the highest amount?

6.3.5 sort(), arrange()

Another useful set of functions, often you want to sort things. The function `arrange()` does this very nicely. You specify the data frame, and the variable you would like to sort by.

```
arrange(fruit_data, amount)
```

```
    fruit colour amount
1  apples  green      2
2 bananas yellow      5
3  apples    red      8
4 oranges orange     10
```

Sorting is ascending by default, but you can specify descending using `desc()`:

```
arrange(fruit_data, desc(amount))
```

```
    fruit colour amount
1 oranges orange     10
2  apples    red      8
3 bananas yellow      5
4  apples  green      2
```

If you 'sort`arrange()` by a list of characters, you'll get alphabetical order:

```
arrange(fruit_data, fruit)
```

```
    fruit colour amount
1  apples  green      2
2  apples    red      8
3 bananas yellow      5
4 oranges orange     10
```

You can sort by multiple things:

```
arrange(fruit_data, fruit, desc(amount))
```

```
    fruit colour amount
1  apples    red      8
2  apples  green      2
3 bananas yellow      5
4 oranges orange     10
```

Notice that now red apples are first.

6.3.6 left_join(), inner_join(), anti_join()

Another set of commands we'll use quite often in this course are the `join()` 'family'. Joins are a very powerful but simple way of selecting certain subsets of data, and adding information from multiple tables together.

Let's make a second table of information giving the delivery day for each fruit type:

```
fruit_type = c('apples', 'bananas','oranges')
weekday = c('Monday', 'Wednesday', 'Friday')

fruit_days = data.frame(fruit_type, weekday, stringsAsFactors = FALSE)

fruit_days
```

```
  fruit_type   weekday
1     apples    Monday
2    bananas Wednesday
3    oranges    Friday
```

This can be 'joined' to the fruit information, to add the new data on the delivery day, without having to edit the original table (or repeat the information for apples twice). This is done using `left_join`.

Joins need a common `key`, a column which allows the join to match the data tables up. It's important that these are unique (a person's name makes a bad key by itself, for example, because it's likely more than one person will share the same name). Usually, we use codes as the join keys. If the columns containing the join keys have different names (as ours do), specify them using the syntax below:

```
joined_fruit = left_join(fruit_data, fruit_days, by = c("fruit" = "fruit_type"))

joined_fruit
```

```
    fruit colour amount   weekday
1  apples  green      2    Monday
2 bananas yellow      5 Wednesday
3 oranges orange     10    Friday
4  apples    red      8    Monday
```

In this new dataframe, the correct weekday is now listed beside the relevant fruit type.

6.3.7 Piping

Another useful feature of the tidyverse is that you can 'pipe' commands through a bunch of functions, making it easier to follow the logical order of the code. This means that you can do one operation, and pass the result to another operation. The previous dataframe is passed as the first argument of the next function by using the pipe `%>%` command. It works like this:

```r
fruit_data %>%
  filter(colour != 'yellow') %>% # remove any yellow colour fruit
  group_by(fruit) %>% # group the fruit by type
  tally(amount) %>% # count each group
  arrange(desc(n)) # arrange in descending order of the count
```

```
# A tibble: 2 x 2
  fruit       n
  <chr>   <dbl>
1 apples     10
2 oranges    10
```

That code block, written in prose: "take fruit data, remove any yellow colour fruit, count the fruits by type and amount, and arrange in descending order of the total"

6.3.8 Plotting using ggplot()

The tidyverse includes a plotting library called `ggplot2`. To use it, first use the function `ggplot()` and specify the dataset you wish to graph using `data =`. Next, add what is known as a 'geom': a function which tells the package to represent the data using a particular geometric form (such as a bar, or a line). These functions begin with the standard form `geom_`.

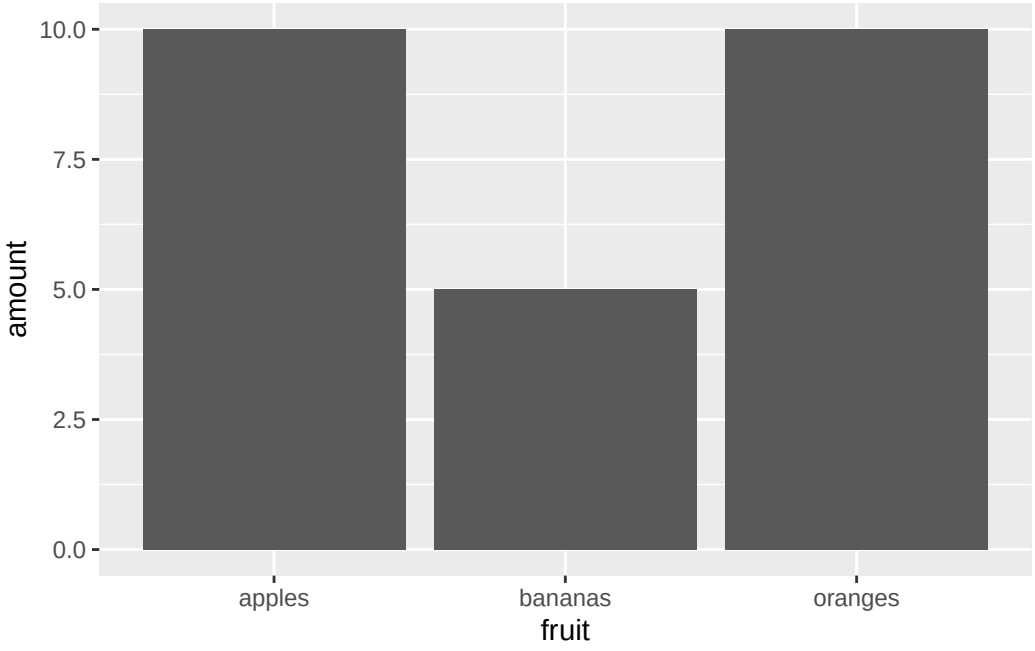
Within this geom, you'll add 'aesthetics', which specify to the package which part of the data needs to be mapped to which particular element of the geom. The most common ones include `x` and `y` for the x and y axes, `color` or `fill` to map colors in your plot to particular data.

`ggplot` is an advanced package with many options and extensions, which cannot be covered here.

Some examples using the fruit data:

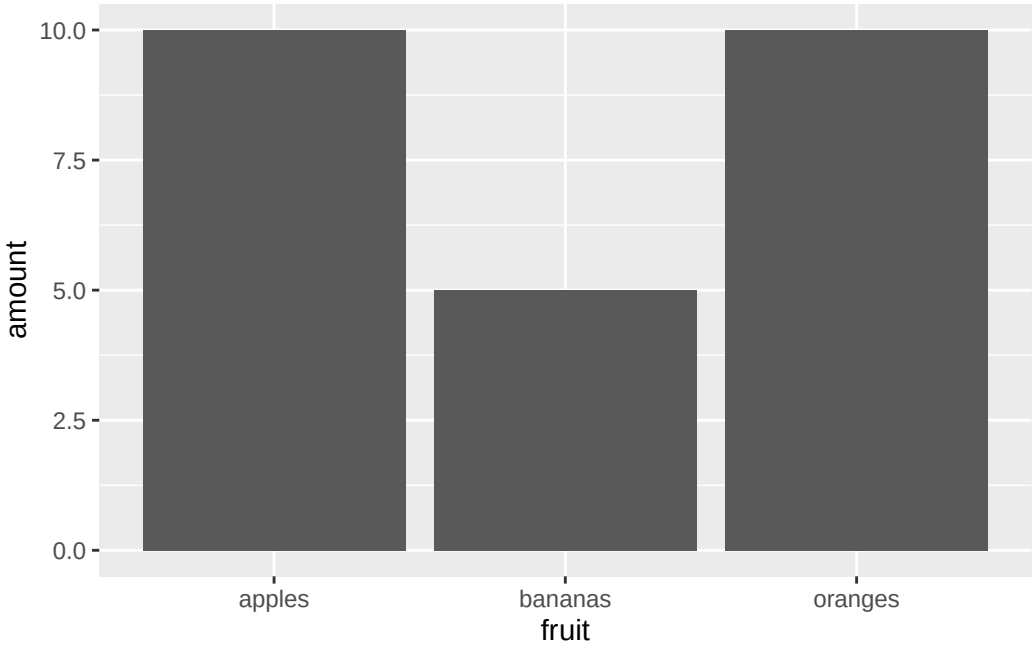
Bar chart of different types of fruit (one each of bananas and oranges, two types of apple)

```r
ggplot(data = fruit_data) + geom_col(aes(x = fruit, y = amount))
```



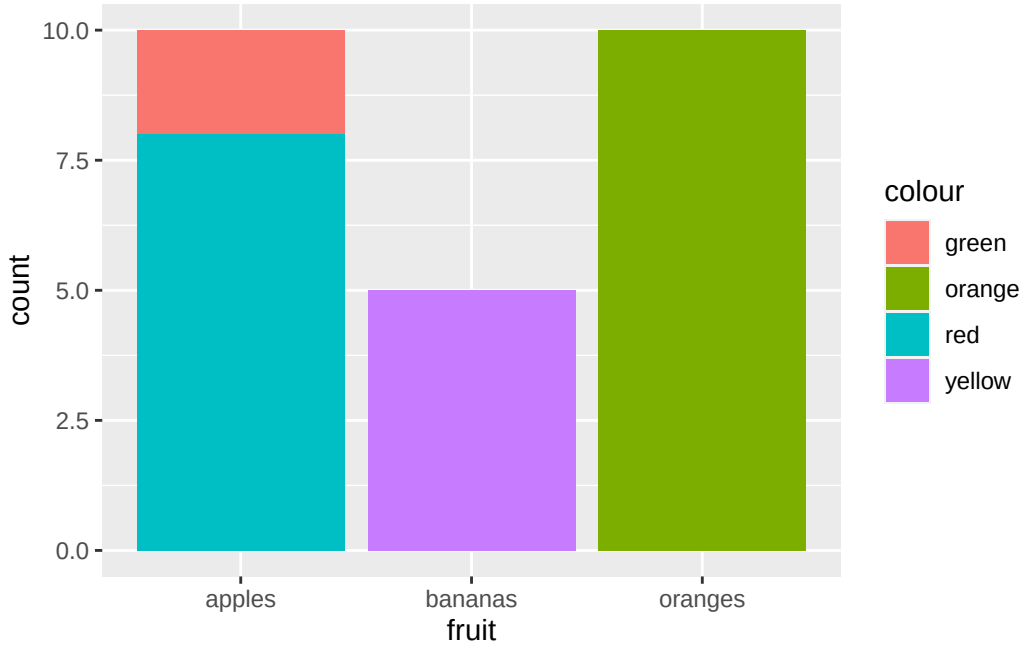
Counting the total amount of fruit:

```
ggplot(fruit_data) + geom_col(aes(x = fruit, y = amount))
```



Charting amounts and fruit colours:

```
ggplot(data = fruit_data) + geom_bar(aes(x = fruit, weight = amount, fill = colour))
```



6.4 Reading in external data

Most of the time, you'll be working with external data sources. These most commonly come in the form of comma separated values (.csv) or tab separated values (.tsv). The tidyverse commands to read these are `read_csv()` and `read_tsv`. You can also use `read_delim()`, and specify the type of delimited using `delim = ','` or `delim = '/t`. The path to the file is given as a string to the argument `file=`.

```
df = read_csv(file = 'aus_titles.csv') # Read a .csv file as a network, specify the path t
```

```
Rows: 2137 Columns: 7
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (4): newspaper_title, state, place_id, place
dbl (3): title_id, latitude, longitude

i Use `spec()` to retrieve the full column specification for this data.
```

```
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
df
```

```
# A tibble: 2,137 x 7
   title_id newspaper_title              state place_id place latitude longitude
      <dbl> <chr>                        <chr> <chr>    <chr>    <dbl>     <dbl>
 1      984 Adelaide Chronicle and Sout~ SA    SA00558~ Adel~    -34.9      139.
 2      986 Adelaide Chronicle and Sout~ SA    SA00558~ Adel~    -34.9      139.
 3      174 Adelaide Morning Chronicle ~ SA    SA00558~ Adel~    -34.9      139.
 4      821 Adelaide Observer (SA : 184~ SA    SA00558~ Adel~    -34.9      139.
 5     1100 Adelaide Times (SA : 1848 -~ SA    SA00558~ Adel~    -34.9      139.
 6      277 Adelaider Deutsche Zeitung ~ SA    SA00558~ Adel~    -34.9      139.
 7      434 Adelong and Tumut Express a~ NSW   NSW81112 Adel~    -35.3      148.
 8      434 Adelong and Tumut Express a~ NSW   NSW60433 Tumb~    -35.8      148.
 9      434 Adelong and Tumut Express a~ NSW   NSW79906 Tumut    -35.3      148.
10      625 Adelong and Tumut Express (~ NSW   NSW81112 Adel~    -35.3      148.
# i 2,127 more rows
```

Notice that each column has a data type beside it, either for text or for numbers. This is important if you want to sort or run calculations on the data.

6.4.1 Doing this with newspaper data

Let's load a dataset of metadata for all the titles held by the library, and practise some counting and sorting on real-world data.

Download from here: British Library Research Repository

You would need to extract into your project folder first, if you're following along:

`read_csv` reads the csv from file.

```
title_list = read_csv('data/BritishAndIrishNewspapersTitleList_20191118.csv')
```

```
Rows: 24927 Columns: 24
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (18): publication_title, edition, preceding_titles, succeeding_titles, p...
dbl  (6): title_id, nid, nlp, first_date_held, publication_date_one, publica...
```

```
i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

Select some particularly relevant columns:

```
title_list %>%
  select(publication_title,
         first_date_held,
         last_date_held,
         country_of_publication)
```

```
# A tibble: 24,927 x 4
   publication_title       first_date_held last_date_held country_of_publication
   <chr>                             <dbl> <chr>          <chr>
 1 "Corante, or, Newes fr~            1621 1621           The Netherlands
 2 "Corante, or, Newes fr~            1621 1621           The Netherlands
 3 "Corante, or, Newes fr~            1621 1621           The Netherlands
 4 "Corante, or, Newes fr~            1621 1621           England
 5 "Courant Newes out of ~            1621 1621           The Netherlands
 6 "A Relation of the lat~            1622 1622           England
 7 "A Relation of the lat~            1622 1622           England
 8 "A Relation of the lat~            1622 1622           England
 9 "A Relation of the lat~            1622 1622           England
10 "A Relation of the lat~            1622 1622           England
# i 24,917 more rows
```

Arrange in order of the latest date of publication, and then by the first date of publication:

```
title_list %>%
  select(publication_title,
         first_date_held,
         last_date_held,
         country_of_publication) %>%
  arrange(desc(last_date_held), first_date_held)
```

```
# A tibble: 24,927 x 4
   publication_title       first_date_held last_date_held country_of_publication
   <chr>                             <dbl> <chr>          <chr>
 1 Shrewsbury chronicle               1773 Continuing     England
 2 London times|The Times~            1788 Continuing     England
```

```
 3 Observer (London)|Obse~            1791 Continuing     England
 4 Limerick chronicle                 1800 Continuing     Ireland
 5 Hampshire chronicle|Th~            1816 Continuing     England
 6 The Inverness Courier,~            1817 Continuing     Scotland
 7 Sunday times (London)|~            1822 Continuing     England
 8 The Impartial Reporter~            1825 Continuing     Northern Ireland
 9 Impartial reporter and~            1825 Continuing     Northern Ireland
10 Aberdeen observer                  1829 Continuing     Scotland
# i 24,917 more rows
```

Group and count by country of publication:

```
title_list %>%
  select(publication_title,
         first_date_held,
         last_date_held,
         country_of_publication) %>%
  arrange(desc(last_date_held)) %>%
  group_by(country_of_publication) %>%
  tally()
```

```
# A tibble: 40 x 2
   country_of_publication              n
   <chr>                           <int>
 1 Bermuda Islands                    24
 2 Cayman Islands                      1
 3 England                         20465
 4 England|Hong Kong                   1
 5 England|India                       2
 6 England|Iran                        2
 7 England|Ireland                    10
 8 England|Ireland|Northern Ireland   10
 9 England|Jamaica                     7
10 England|Malta                       2
# i 30 more rows
```

Arrange again, this time in descending order of number of titles for each country:

```
title_list %>%
  select(publication_title,
         first_date_held,
```

```
          last_date_held,
          country_of_publication) %>%
    arrange(desc(last_date_held)) %>%
    group_by(country_of_publication) %>%
    tally() %>%
    arrange(desc(n))
```

```
# A tibble: 40 x 2
   country_of_publication              n
   <chr>                           <int>
 1 England                         20465
 2 Scotland                         1778
 3 Ireland                          1050
 4 Wales                            1019
 5 Northern Ireland                  415
 6 England|Wales                      58
 7 Bermuda Islands                    24
 8 England|Scotland                   13
 9 England|Ireland                    10
10 England|Ireland|Northern Ireland   10
# i 30 more rows
```

Filter only those with more than 100 titles:

```
  title_list %>%
    select(publication_title,
           first_date_held,
           last_date_held,
           country_of_publication) %>%
    arrange(desc(last_date_held)) %>%
    group_by(country_of_publication) %>%
    tally() %>%
    arrange(desc(n)) %>%
    filter(n>=100)
```

```
# A tibble: 5 x 2
  country_of_publication     n
  <chr>                  <int>
1 England                20465
2 Scotland                1778
3 Ireland                 1050
```

```
4 Wales                   1019
5 Northern Ireland         415
```

Make a simple bar chart:

```
title_list %>%
  select(publication_title,
         first_date_held,
         last_date_held,
         country_of_publication) %>%
  arrange(desc(last_date_held)) %>%
  group_by(country_of_publication) %>%
  tally() %>%
  arrange(desc(n)) %>%
  filter(n>=100) %>%
  ggplot() +
  geom_bar(aes(x = country_of_publication, weight = n))
```

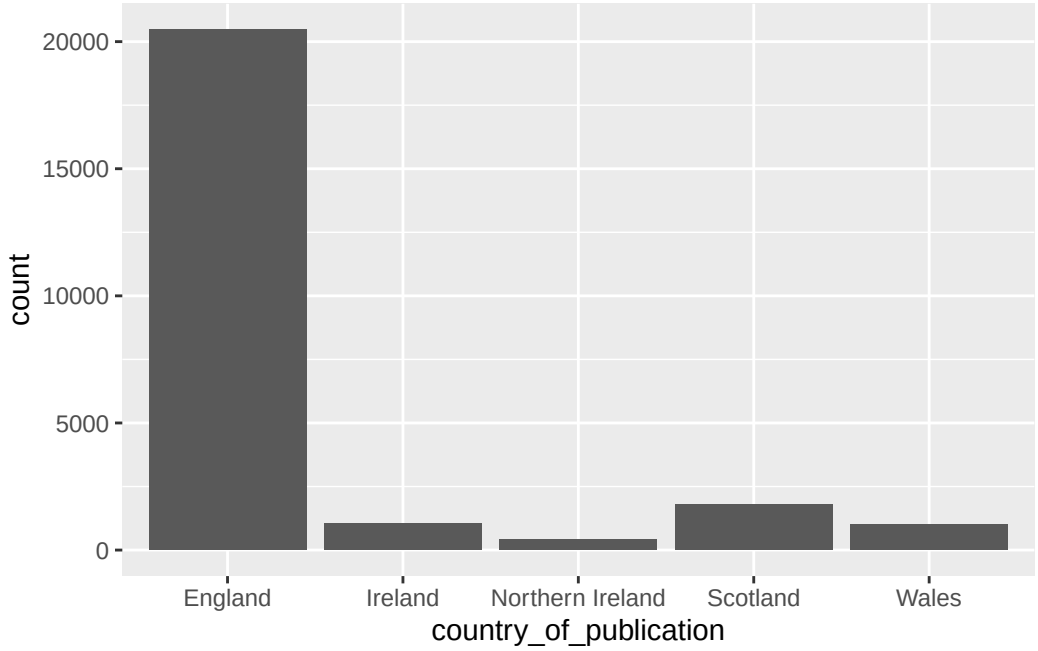


Figure 6.1: barchart

6.5 Recommended Reading

This has been a very quick introduction to R. There are lots of resources available to learn more, including:

R-studio cheat sheets

The Pirate's Guide to R, a good beginners guide to base R

R for data science, which teaches the tidyverse in detail

Learn how to make a book like this using Bookdown

7 Working with Metadata: Mapping the British Library Newspaper Collection

7.1 News Metadata

While much of the rest of this book deals with text data, that is only one type of what might be considered newspaper data. Also important to consider is newspaper *metadata*, which can be used to understand and analyse newspaper collections and is valuable in its own right.

Key metadata sources for the British Library newspaper collection are a series of title-level lists covering UK, worldwide, and even television news, and a series of press directories, containing information on the newspapers from contemporary sources.

7.1.1 Title-level lists

The title list has been produced by the British Library and is published on their institutional repository. It contains metadata taken from the Library's catalogue of every newspaper published in Britain and Ireland up until the year 2019, a total of about 24,000 titles. There is more information available in a published data paper (Ryan and McKernan 2021).

This list contains a number of fields for each title, including the name of the publication, subsequent and previous title names, several fields for geographic coverage, the first and last dates held, and some other information. Where digitised versions have been created, these are noted and links are provided. It can be used to get statistical information about the Library's collection, such as the number of new titles per year, the proportion of material which has been digitised, and how the collection is spread geographically.

7.1.2 Press Directories

The Living with Machines project has digitised a number of press directories, between 1846 and 1920. These directories were produced by a number of companies, and were intended for advertisers and others as a guide to the content and coverage of printed newspapers. They contain information on the newspaper owners, geographic coverage, political leaning, and circulation. This, along with the title list, can serve as an excellent source for mapping, analysing, and visualising the newspaper collection at scale.

The directories have been digitised and the 'raw' .xml from the OCR process has been made available. Additionally, a single file where the information has been extracted and standardised, can also be downloaded.

7.2 Producing Maps With Metadata and R

The following two short tutorials aim to show how this kind of metadata can also be used to produce visualisations, and use the title list to produce a data map of the newspaper collection in Great Britain. They also aim to teach the reader how R can be used a GIS (Geographic Information System), and produce high-quality maps.

There are two, related tutorials. First, the title list is used to draw a map of points, relating to the number of newspaper titles published in each location. The second tutorial will guide you through creating a thematic, or choropleth map, using the same data.

7.3 Mapping Data as Points

7.3.1 Lesson Steps

The basic steps to create a map of points are as follows: 1) download a base map of the UK and Ireland, 2) count the total number of titles of each city and merge this with coordinate information on those cities and 3) create a shapefile out of that information, 4) draw both the basic map and the points on top of it.

To do this we'll need three elements:

- A background map of the UK and Ireland
- A count of the total titles for each city
- A list of coordinates for each place mentioned in the title list.

7.3.2 Requirements

This lesson assumes that you have some basic knowledge of R, and in particular, the package ggplot2. Not every step using this package is fully explained. You might want to refresh your knowledge by reading over Chapter 6, which gives a very short introduction.

On top of the packages used already, most importantly `tidyverse`, you'll need to install a few more packages if you don't have them already:

- the `sf` package, which we'll use to create and work with the geographic data

- The `rnaturalearth` package. This will be used to download the underlying map of the UK and Ireland, on which the points will be overlaid.

To install all the necessary packages, you can use the following code:

```
install.packages('tidyverse')
install.packages('rnaturalearth')
install.packages('sf')
```

Lastly, you'll need to download a file containing the coordinate locations for the places in the title list, because this information is not included. This file can be downloaded here.

7.3.3 Download a 'basemap'

The first step is to download a basemap: data containing the shape of the background area onto which the points will be overlaid. The easiest way to do this in R is to install the `rnaturalearth` package. This package makes it easy to download shapefiles: data containing polygons of various maps of the earth, containing shapes for everything from political borders to lakes and rivers.

Once you have installed the `rnaturalearth` and `sf` packages, load them as well as the `tidyverse`

```
library(tidyverse)
library(rnaturalearth)
library(sf)
```

Next, use the `rnaturalearth` package to download a map of the UK and Ireland. There are a number of different shapefiles in the Natural Earth database we could choose from. One is 'countries', which is a set of connected shapes, one for each country on Earth. The other is 'coastline', which you could think of as a single continuous shape tracing the entire coastline of the planet.

Both have advantages and disadvantages. Using 'countries' makes it easy to just download the information for specific countries. However, it does mean that the shapes contain modern internal borders, which might not be suitable in all cases (if you are making a history map of Europe, for example).

In this case, we'll use the 'countries' database. Use the `ne_countries` function from the rnaturalearth package to download the relevant shapes and save to a variable called `uk_ireland_sf`. We have to set a number of parameters in the function:

`scale` is set to `"medium"`: this is the resolution of the shapes (how detailed they will be). Medium is fine at this level.

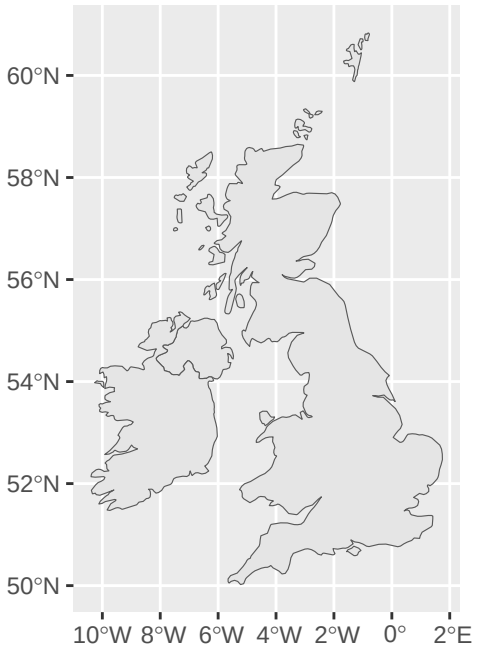
`country` is set to a vector of the country names we want, in this case Ireland and the UK.

`returnclass` is set to `sf`. This is the format for the geographic information which we'll use with the `sf` package. `sf` stands for 'shapefile', and is a standardised way of representing geographic data, which is also easy to query as a dataset, for example to filter or summarise the data. `rnaturalearth` can return a file in this format automatically.

```
uk_ireland_sf = ne_countries(scale = 'medium',
                             country = c("United Kingdom", "Ireland"),
                             returnclass = 'sf')
```

Once you run this, you should have a new variable called uk_ireland_sf. We can draw this to check exactly what has been downloaded. To do this, we can use a function in ggplot2 called `geom_sf()`. This geom draws geographic information from shapefiles.

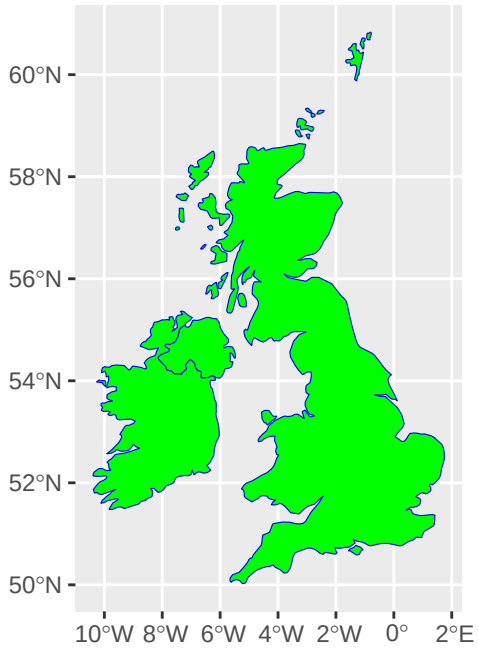
```
ggplot() + geom_sf(data = uk_ireland_sf)
```



Now is a good moment to demonstrate some features of ggplot2 which are particularly relevant for mapping. This map is a good starting point, but some parts of it should be improved to make it more readable: we don't need the border in Northern Ireland to be drawn, we don't want the coordinates written on the x and y axes, and we want a simpler, single colour background.

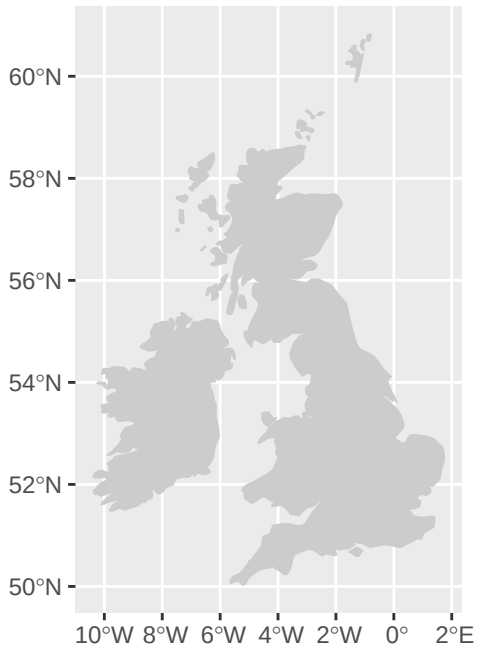
Let's start by making some changes to the colour. We can change both the line and the fill colours. To do this, we add some parameters to the `geom_sf()` function. The `color` parameter will set the outer borders, and the `fill` parameter the inside of the shape. To make the differences obvious I'll use two very distinct colours:

```
ggplot() +
  geom_sf(data = uk_ireland_sf, color = 'blue', fill = 'green')
```



Remember we chose 'countries' rather than 'coastline' above? One small drawback is that we can't draw a map using the countries data where a distinct line surrounds both islands. One quite legible way to draw the map is to have a lightly-coloured background and render the entire map shape, both internal and external lines, in a single colour. To do this, we'll set both `color` and `fill` to a medium/light gray:

```
ggplot() +
  geom_sf(data = uk_ireland_sf, color = 'gray80', fill = 'gray80')
```



Next, we'll get rid of the background clutter. The following code will remove the background and axis text:

```
ggplot() +
  geom_sf(data = uk_ireland_sf, color = 'gray80', fill = 'gray80') +
  theme(axis.text = element_blank(),
        axis.ticks = element_blank(),
        panel.background = element_blank()
        )
```

Now we have a map onto which we can draw the points.

7.3.4 Cropping the Map

In some cases, you may want to crop a map further. We can do that with `coord_sf()`. `coord_fixed()` is used to fix the aspect ratio of a coordinate system, but can be used to specify a bounding box by using two of its arguments: `xlim=` and `ylim=`.

These each take a vector (a series of numbers) with two items A vector is created using `c()`. Each item in the vector specifies the limits for that axis. So `xlim = c(0,10)` means *restrict the x-axis to 0 and 10.*

The axes correspond to the lines of longitude (x) and latitude (y). We'll restrict the x-axis to `c(-10, 4)` and the y-axis to `c(50.3, 60)` which should just about cover the UK and Ireland.

```
ggplot() +
  geom_sf(data = uk_ireland_sf, color = 'gray80', fill = 'gray80') +
  theme(axis.text = element_blank(),
        axis.ticks = element_blank(),
        panel.background = element_blank()
        ) +
  coord_sf(xlim = c(-10,3),
             ylim = c(50.3, 59))
```

Figure 7.1: Empty Map

7.3.5 Add Sized Points By Coordinates

On top of this basemap, we will draw a set of points, sized by the count of newspapers from each place in a metadata list of newspaper titles from the British Library.

Doing this involves a few steps:

1. Download a metadata list of newspaper titles held by the British Library
2. Generate a new dataset, containing a total count the newspapers from each place in this title list.
3. Merge this with a dataset of coordinates for each place
4. Turn this information into a shapefile, as the map downloaded above.
5. Draw the points on top of the base map.

7.3.5.1 Download the metadata list

To start with, we'll need a complete list of the newspaper titles held in by in the British Library's collection. This is available as a free download through the Open Research Repository. Go to the repository page and download the file BritishAndIrishNewspapersTitleList_20191118.zip (or just click the link to directly download).

Unzip this file and note where you save it. It will contain a .csv file with the metadata list. Import this to R with the following (replace the `'data/BritishAndIrishNewspapersTitleList_20191118.csv'` with the path to your file):

```
title_list = read_csv('data/BritishAndIrishNewspapersTitleList_20191118.csv')
```

Next, create a new dataframe, which will simply be count of the appearances of each place in the metadata. Use the `dplyr` commands `group_by()` and `tally()`, which will count the instances of each different combination of the three geographic information columns:

```
location_counts = title_list %>%
  group_by(country_of_publication,
           general_area_of_coverage,
           coverage_city) %>%
  tally()
```

This new dataframe, `location_counts`, will have one row for each combination of country/general area/city in the data, along with a new column, `n`, containing the count:

```
location_counts %>% head(5) %>% kableExtra::kbl()
```

country_of_publication	general_area_of_coverage	coverage_city	n
Bermuda Islands	NA	Hamilton	23
Bermuda Islands	NA	Saint George	1
Cayman Islands	NA	Georgetown	1
England	Aberdeenshire (Scotland)	Peterhead	1
England	Acton (London, England)	Ealing (London, England)	1

7.3.6 Get hold of a list of geocoordinates

To create the shapefile and visualise the data, the next step is to merge this text information on the places, to a dataset of coordinates. For this, you'll need existing coordinate information, which has been created separately. This file is available online from the following source. If you run this code, it will import the coordinate file directly into R from its storage place on Github.

```
geocorrected = read_csv('https://raw.githubusercontent.com/yann-ryan/r-for-news-data/maste
```

This file also needs some pre-processing. A preliminary step here is to change the column names so that they match those found in the metadata:

```
library(snakecase)
colnames(geocorrected) = to_snake_case(colnames(geocorrected))
```

There are a few further pieces of pre-processing to do before we can merge this to the `location_counts` dataframe. We'll change some of the column names and remove some unnecessary ones, and change the `na` values to `NA`, which is properly recognised by R. Last, we'll change the coordinate information to numeric values, and then filter out any rows with missing coordinate information:

```
colnames(geocorrected)[7:8] = c('lat',
                                'lng')

geocorrected = geocorrected %>%
  select(-1, -5,-9,-10, -11, -12)

geocorrected = geocorrected %>%
  mutate(country_of_publication = replace(country_of_publication,
                                          country_of_publication == 'na', NA)) %>%
  mutate(general_area_of_coverage = replace(general_area_of_coverage,
                                            general_area_of_coverage == 'na', NA)) %>%
  mutate(coverage_city = replace(coverage_city,
                                 coverage_city == 'na', NA))

geocorrected = geocorrected %>%
  mutate(lat = as.numeric(lat)) %>%
  mutate(lng = as.numeric(lng)) %>%
  filter(!is.na(lat)) %>% filter(!is.na(lng))

geocorrected %>% head(5)%>% kableExtra::kbl()
```

coverage_city	general_area_of_coverage	country_of_publication	wikititle	
Afghanistan	Afghanistan	England	Afghanistan	33.00
Airdrie	Airdrie	England	Airdrie,_North_Lanarkshire	55.86
Albania	Albania	England	Albania	41.00
Australia	Australia	England	Australia	-25.00
Bahrain	Bahrain	England	Bahrain	26.02

The result is a dataframe with a set of longitude and latitude points (they come from Wikipedia, which is why they are prefixed with *wiki*) for every combination of city/county/country in the list of titles. These can be joined to the full title list with the following method:

Using `left_join()` we will merge these dataframes, joining up each set of location information

to its coordinates and standardised name. `left_join()` is a very common command in data analysis. It merges two sets of data by matching a value known as a key.

Here the key is actually a combination of three values - city, county and country, and it matches up the two sets of data by 'joining' two rows together, if they share all three of these values. Store this is a new variable called `lc_with_geo`.

```
lc_with_geo = location_counts %>%
  left_join(geocorrected,
            by = c('coverage_city' ,
                   'general_area_of_coverage',
                   'country_of_publication')) %>%
  filter(!is.na(lat))
```

If you look at this new dataset, you'll see that now the counts of locations have merged with the geocorrected data. Now we have an amount and coordinates for each place.

```
head(lc_with_geo, 10)%>% kableExtra::kbl()
```

country_of_publication	general_area_of_coverage	coverage_city	n	wikititle
England	Avon	Bath	88	UNKNOWN
England	Avon	Bristol	175	Bristol
England	Avon	Clevedon	14	Clevedon
England	Avon	Keynsham	4	Keynsham
England	Avon	Nailsea	2	Nailsea
England	Avon	Norton	4	UNKNOWN
England	Avon	Portishead	2	Portishead,_Somerset
England	Avon	Radstock	6	Radstock
England	Avon	Thornbury	3	Thornbury,_Gloucestersh
England	Avon	Weston-super-Mare	23	Weston-super-Mare

A next step is to use `group_by()` and `tally()` again, this time on the the *wikititle*, *lat* and *lng* columns. This is because the wikititle is a standardised title, which means it will group together cities properly, rather than giving a different row for slightly different combinations of the three geographic information columns (incidentally, it could also be used to link to wikidata). At the same time, filter again to ensure no rows are missing latitude or longitude information:

```
lc_with_geo_counts = lc_with_geo %>%
  group_by(wikititle, lat, lng) %>%
  tally(n) %>% filter(!is.na(lat)& !is.na(lng))
```

Now we've got a dataframe with counts of total newspapers, for each standardised wikipedia title in the dataset.

```
knitr::kable(head(lc_with_geo_counts,10))
```

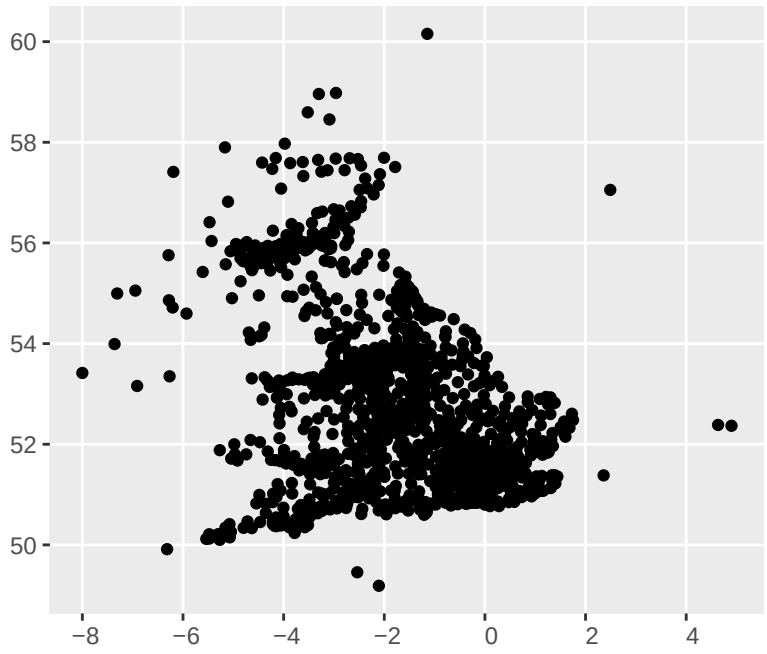
wikititle	lat	lng	n
Abbots_Langley	51.7010	-0.4160	1
Aberavon_(UK_Parliament_constituency)	51.6000	-3.8120	1
Aberdare	51.7130	-3.4450	20
Aberdeen	57.1500	-2.1100	82
Abergavenny	51.8240	-3.0167	9
Abergele	53.2800	-3.5800	8
Abersychan	51.7239	-3.0587	2
Abertillery	51.7300	-3.1300	2
Aberystwyth	52.4140	-4.0810	31
Abingdon-on-Thames	51.6670	-1.2830	23

Finally, create the 'shapefile' object. To do this from a dataframe, we'll use a function `st_as_sf` from the `sf` package. Also specify the columns this function should use as the coordinates for the shapefile, using the `coords =` parameter, and specify in a vector the longitude and latitude columns from the data:

```
lc_with_geo_counts_sf = lc_with_geo_counts %>%
  st_as_sf(coords = c('lng', 'lat'))
```

We can draw this shapefile using `geom_sf`, to check and see that it looks reasonable:

```
ggplot() + geom_sf(data = lc_with_geo_counts_sf)
```



7.3.7 Setting a Coordinate Reference System (CRS)

To create the final map, there's one more important step. A shapefile is not just a list of coordinates, but also includes a 'coordinate reference system' (CRS), which tells how the coordinates in that shapefile should be interpreted. As a first step, we'll set both our shapefiles to have the same CRS:

```
uk_ireland_sf = uk_ireland_sf %>% st_set_crs(4326)

lc_with_geo_counts_sf = lc_with_geo_counts_sf %>% st_set_crs(4326)
```

7.3.8 Creating the Final Map

First we'll start with the base map from the first steps:

```
ggplot() +
  geom_sf(data = uk_ireland_sf, color = 'gray80', fill = 'gray80') +
  theme(axis.text = element_blank(),
        axis.ticks = element_blank(),
        panel.background = element_blank()
        )
```

Figure 7.2: Blank Map of UK and Ireland

Next, plot the newspaper place information on top of this using another `geom_sf`. The second one is added to the first layer using a `+` sign:

```
ggplot() +
  geom_sf(data = uk_ireland_sf, color = 'gray80', fill = 'gray80') +
  theme(axis.text = element_blank(),
        axis.ticks = element_blank(),
        panel.background = element_blank()
        ) +
  geom_sf(data = lc_with_geo_counts_sf)
```

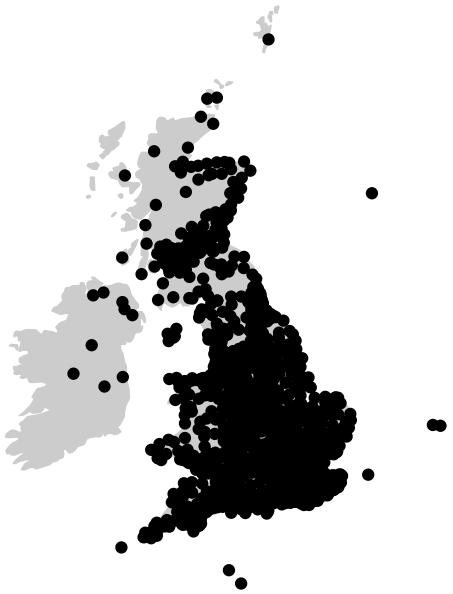


Figure 7.3: Blank Map of UK and Ireland

We see that the points are drawn on top of the base map. To make the map more informative and legible, there are a few more things we can do. First, size the points by the count of the instances (column 'n'). This is done by setting the `size` aesthetic in ggplot to the column name, inside the `aes()`. We'll also add the command `scale_size_area()` to the code, which better represents the relationship between the numeric value and the circle size:

```
ggplot() +
  geom_sf(data = uk_ireland_sf, color = 'gray80', fill = 'gray80') +
  theme(axis.text = element_blank(),
        axis.ticks = element_blank(),
        panel.background = element_blank()
        ) +
  geom_sf(data = lc_with_geo_counts_sf, aes(size = n)) + scale_size_area()
```

We can also reduce the transparency of the points to make them more readable, using the `alpha` aesthetic:

```
ggplot() +
  geom_sf(data = uk_ireland_sf, color = 'gray80', fill = 'gray80') +
  theme(axis.text = element_blank(),
        axis.ticks = element_blank(),
        panel.background = element_blank()
        ) +
  geom_sf(data = lc_with_geo_counts_sf, aes(size = n), alpha = .6) +
  scale_size_area()
```

Using `labs()`, add a title, and with `scale_size_area()` and `scale_color_viridis_c()`, make some changes to the size and colours, respectively.

7.4 Drawing a newspaper titles 'Choropleth' map with R and the sf package

Another type of map is known as a 'choropleth'. This is where the data is visualised by a certain polygon area rather than a point. Typically these represent areas like parishes, counties or countries. Using the library `sf` a choropleth map can be made quite quickly.

To do this, we will use the `sf` package to merge the information at the coordinate level with information on the geographic county borders. Next, we'll count the number of titles within each county, and use this total to color or shade the map. The good thing about this method is that once you have a set of coordinates, they can be situated within any shapefile - a historic map, for example. This is particularly useful for anything to do with English counties, which have changed several times throughout history.

This section uses data from .visionofbritain.co.uk, which needs to be downloaded separately. You could also use the free boundary data here: https://www.ordnancesurvey.co.uk/business-government/products/boundaryline, which contains boundaries for both modern and historic counties.

This is an excellent source, and the file includes a range of boundaries including counties but also districts and constituencies, under an 'Open Government Licence'.

7.4.1 Choropleth map steps

The steps to create this type of map:

- Download shapefiles for England and scotland from here
- Turn into sf object Download list of points, turn into sf object
- Use st join to get county information
- Join to the title list and deselect everything except county and titles - maybe 19th century only..
- Join that to the sf object Plot using geom_sf()

Load libraries

```
library(tidyverse)
library(sf)
sf::sf_use_s2(FALSE)
```

7.4.2 Get county information from the title list

Next, download (if you haven't already) the title list from the British Library open repository.

```
title_df = read_csv('data/BritishAndIrishNewspapersTitleList_20191118.csv')
```

7.4.3 Download shapefiles

First, download the relevant shapefiles. These don't necessarily have to be historic ones. Use `st_read()` to read the file, specifying its path. Do this for England, Wales and Scotland (we don't have points for Ireland).

7.5 Transform from UTM to lat/long using st_transform()

These shapefiles use points system known as UTM, which stands for 'Universal Transverse Mercator'. According to wikipedia,

> it differs from global latitude/longitude in that it divides earth into 60 zones and projects each to the plane as a basis for its coordinates.

It needs to be transformed into lat/long coordinates, because the coordinates we have are in that format. This is easy with `st_transform()`:

```
eng_1851 = st_transform(eng_1851, crs = 4326)
```

```
scot_1851  = st_transform(scot_1851, crs = 4326)
```

Bind them both together, using `rbind()` to make one big shapefile for Great Britain.

```
gb1851 = rbind(eng_1851, scot_1851 %>%
                  select(-UL_AUTH))
```

7.5.1 Download and merge the title list with a set of coordinates.

Next, load and pre-process the set of coordinates:

```
geocorrected = read_csv('data/geocorrected.csv')
```

Change the column names:

```
library(snakecase)
colnames(geocorrected) = to_snake_case(colnames(geocorrected))
```

Change some column names further, select just the relevant columns, change the NA values and get rid of any empty entries.

```
colnames(geocorrected)[6:8] = c('wikititle', 'lat', 'lng')

geocorrected = geocorrected %>% select(-1, -9,-10, -11, -12)

geocorrected = geocorrected %>%
  mutate(country_of_publication = replace(country_of_publication,
                                          country_of_publication == 'na', NA)) %>% mutate(

  mutate(coverage_city = replace(coverage_city,
                                 coverage_city == 'na', NA))

geocorrected = geocorrected %>%
  mutate(lat = as.numeric(lat)) %>%
  mutate(lng = as.numeric(lng)) %>%
  filter(!is.na(lat)) %>%
  filter(!is.na(lng))
```

Next, join these points to the title list, so that every title now has a set of lat/long coordinates.

```
title_df = title_df %>%
  left_join(geocorrected) %>%
  filter(!is.na(lat)) %>%
  filter(!is.na(lng))
```

7.5.2 Using st_join to connect the title list to the shapefile

To join this to the shapefile, we need to turn it in to an simple features item. To do this we need to specify the coordinates and the CRS. The resulting file will contain a new column called 'geometry', containing the lat/long coordaintes in the correct simple features format.

```
st_title = st_as_sf(title_df, coords = c('lng', 'lat'))

st_title = st_title  %>% st_set_crs(4326)
```

Now, we can use a special kind of join, which will join the points in the title list, if they are within a particular polygon. The resulting dataset now has the relevant county, as found in the shapefile.

```
st_counties = st_join(st_title, gb1851)
```

Make a new dataframe, containing just the counties and their counts.

```
county_tally = st_counties %>%
  select(G_NAME) %>%
  group_by(G_NAME) %>%
  tally() %>%
  st_drop_geometry()
```

7.5.3 Draw using ggplot2 and geom_sf()

Join this to the shapefile we made earlier, which gives a dataset with the relevant counts attached to each polygon. This can then be visualised using the `geom_sf()` function from ggplot2, and all of ggplot2's other features can be used.

```
gb1851 %>%
  left_join(county_tally) %>%
  ggplot() +
  geom_sf(lwd = .5,color = 'black', aes(fill = n)) +
  theme_void() +
  lims(fill = c(10,4000)) +
  scale_fill_viridis_c(option = 'plasma') +
  labs(title = "British Library Newspaper\nTitles, by County",
       fill = 'No. of Titles:') +
  theme(legend.position = 'left') +
  theme(title = element_text(size = 12),
        legend.title = element_text(size = 8))
```

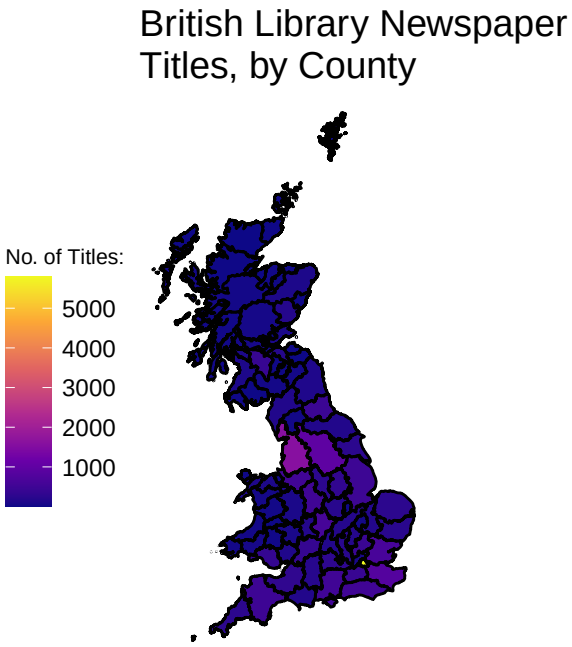


Figure 7.4: Choropleth Made with geom_sf()

7.6 Recommended Reading

This article uses both of these metadata files (the title list and the press directories) as its historical evidence:

Kaspar Beelen and others, Bias and representativeness in digitized newspaper collections: Introducing the environmental scan, *Digital Scholarship in the Humanities*, Volume 38, Issue 1,

April 2023, Pages 1–22, https://doi.org/10.1093/llc/fqac037

The book 'Geocomputation with R' is a fantastic resource for learning about mapping: https://geocompr.robinlovelace.net.

8 Accessing Newspaper Data from the Shared Research Repository

Most of the rest of this book uses newspaper data from a number of newspaper digitisation projects connected to the British Library, as written about in Chapter 2. These projects have made the 'raw data' for a number of titles freely available for anyone to use. This chapter explains the structure of the repository which holds them, and walks through a method for downloading titles in bulk.

8.1 Shared Research Repository

The titles released so far are available on the British Library's Shared Research Repository.

The items in the repository are organised into collections. All the newspaper-related data released on to the repository can be found within the British Library News Datasets collection. Clicking on this link will bring up a list of all the items collected under this headings. There are also three sub-headings: Title Lists, Newspapers, and Press Directories. Clicking on the first of these, Newspapers, will display just the newspaper data items.

8.1.1 Newspaper File Structure

Each separate title (if a newspaper changed title, they are combined together) is listed here as a dataset. Clicking into one of these, you'll see that each year of that title is available as a separate downloadable .zip file.

If you download one of these and decompress it, you'll see the structure of the title. It contains a root folder, containing the name and year of the title, as well as a unique title code. Contained within this folder are further folders, one for each day of the newspaper. These folders are named using the month and day of publication.

Within this folder are the actual newspaper files: one .xml file for each page of that day's paper, plus one more METS file.

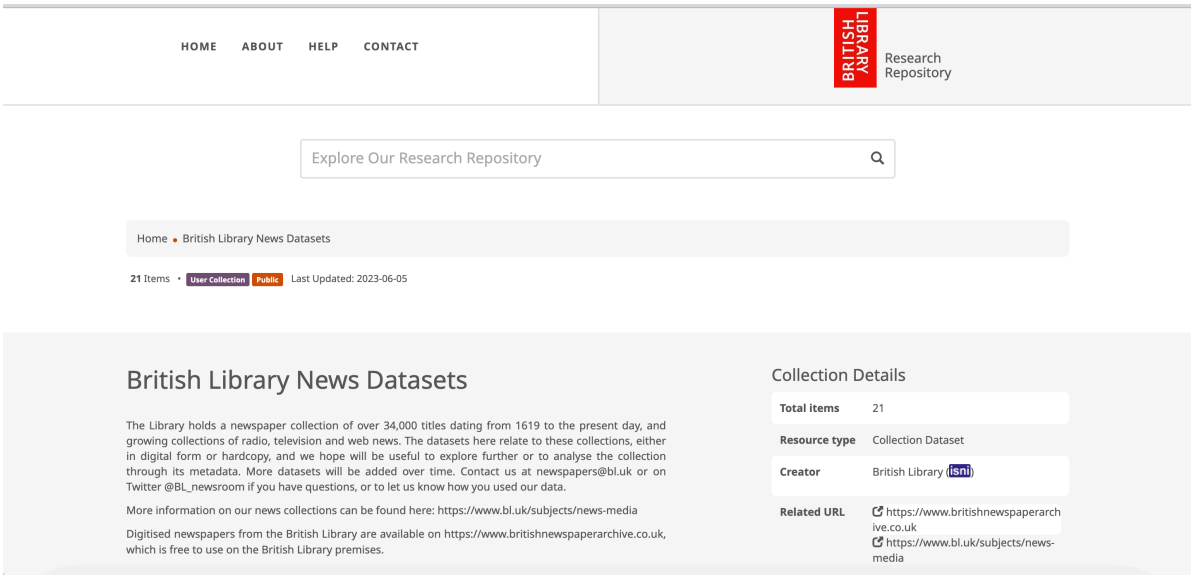


Figure 8.1: Screenshot showing the Shared Research Repository maintained by the British Library, on the newspaper collections page.

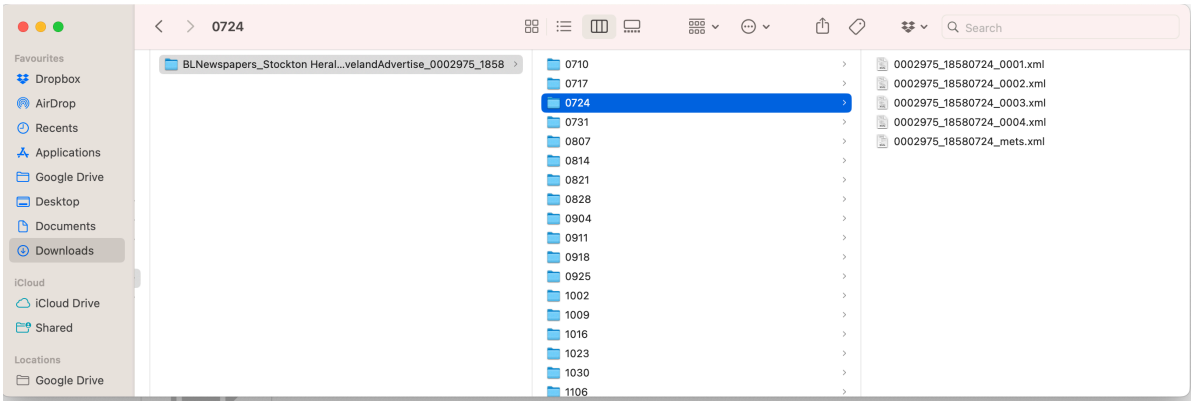


Figure 8.2: Screenshot showing the folder structure of a decompressed newspaper-year file

8.2 Downloading Titles in Bulk

Acquiring a dataset on which to work can be cumbersome if you have to manually download each file. The first part of this tutorial will show you how to bulk download all or some of the available titles. This was heavily inspired by the method found here. If you want to bulk download titles using a more robust method (using the command line, so no need for R), then I really recommend checking out that repository.

To do this there are three basic steps:

- Make a list of all the links in the repository collection for each title
- Go into each of those title pages, and get a list of all links for each zip file download.
- Optionally, you can specify which titles you'd like to download, or even which year.
- Download all the relevant files to your machine.

8.2.1 Create a dataframe of all the newspaper files in the repository

First, load the libraries needed:

```
library(tidyverse)
library(XML)
library(xml2)
library(rvest)
```

Next, grab all the pages of the collection:

```
urls = paste0("https://bl.iro.bl.uk/collections/9a6a4cdd-2bfe-47bb-8c14-c0a5d100501f?local
```

Use `lapply` to go through the list, use the function `read_html` to read the page into R, and store each as an item in a list:

```
list_of_pages <- lapply(urls, read_html)
```

Next, write a function that takes a single html page (as downloaded with `read_html`), extracts the links and newspaper titles, and puts it into a dataframe.

```
make_df = function(x){

  all_collections =  x %>%
  html_nodes(".search-result-title") %>%
  html_nodes('a') %>%
  html_attr('href') %>%
```

```
  paste0("https://bl.iro.bl.uk",.)

all_collections_titles = x %>%
  html_nodes(".search-result-title") %>%
  html_text()

all_collections_df = tibble(all_collections, all_collections_titles) %>%
  filter(str_detect(all_collections, "concern\\/datasets"))

all_collections_df

}
```

Run this function on the list of html pages. This will return a list of dataframes. Merge them into one with `rbindlist` from `data.table`.

```
l = purrr::map(list_of_pages, make_df)

l = data.table::rbindlist(l)
l %>% knitr::kable('html')

l = l %>% mutate(pages = paste0(all_collections, "&page="))

sequence <- 1:10

# Expand the dataframe and concatenate with the sequence
expanded_df <- l %>%
  crossing(sequence) %>%
  mutate(pages = paste(pages, sequence, sep = ""))
```

Now we have a dataframe containing the url for each of the titles in the collection. The second stage is to go to each of these urls and extract the relevant download links.

Write another function. This takes a url, extracts all the links and IDs within it, and turns it into a dataframe. It then filters to just the relevant links (which have the ID 'file_download').

```
get_collection_links = function(c){
  tryCatch({
collection = c  %>%  read_html()

links = collection%>% html_nodes('a') %>% html_attr('href')
```

```
id = collection %>% html_nodes('a') %>% html_attr('id')

text = collection%>% html_nodes('a') %>% html_attr('title')

links_df = tibble(links, id, text)

}, error = function(e) {
    # Action to perform when an error occurs
    result <- NA
  })

return(links_df)



}
```

Use `lapply` to run this function on the column of urls from the previous step,and merge it with `rbindlist`. Keep just links which contain the text `Download BLNewspapers`.

```
t = pbapply::pblapply(expanded_df$pages, get_collection_links)

names(t) = expanded_df$all_collections_titles[1]

t_df = t %>%
  data.table::rbindlist(idcol = 'title')

t_df = t_df %>%
  filter(str_detect(text, "BLNewspapers"))
```

The new dataframe needs a bit of tidying up. To use the `download.file()` function in R we need to also specify the full filename and location where we'd like the file to be put. At the moment the 'text' column is what we want but it needs some alterations. First, remove the 'Download' text from the beginning.

Next, separate the text into a series of columns, using either _ or . as the separator. Create a new 'filename' column which pastes the different bits of the text back together without the long code.

Add `/newspapers/` to the beginning of the filename, so that the files can be downloaded into that folder.

```
t_df = t_df %>% distinct(text, .keep_all = TRUE) %>%
  mutate(year = str_extract(text, "(?<=_)[0-9]{4}(?=[_.])")) %>%
  mutate(nid = str_extract(text, "[0-9]{7}")) %>% mutate(filename = str_extract(text, '(?<

t_df = t_df %>% mutate(links = paste0("https://bl.iro.bl.uk",links )) %>%
  mutate(destination = paste0('/Users/Yann/Documents/non-Github/r-newspaper-quarto/newspap
```

The result is a dataframe which can be used to download either all or some of the files.

8.2.2 Filter the download links by date or title

You can now filter this dataframe to produce a list of titles and/or years you're interested in. For example, if you just want all the newspapers for 1855:

```
files_of_interest = t_df %>% filter(as.numeric(year) == 1855)
files_of_interest%>% knitr::kable('html')
```

To download these we use the `Map` function, which will apply the function `download.file` to the vector of links, using the dest colum we created as the file destination. `download.file` by default times out after 100 seconds, but these downloads will take much longer. Increase this using `options(timeout=9999)`.

Before this step, you'll need to create a new folder called 'newspapers', within the working directory of the R project.

```
options(timeout=9999)

Map(function(u, d) download.file(u, d, mode="wb"), files_of_interest$links, files_of_inter
```

8.2.3 Folder structure

Once these have downloaded, you can quickly unzip them using R. First it's worth understanding a little about the folder structure you'll see once they're unzipped.

Each file will have a filename like this:

BLNewspapers_TheSun_0002194_1850.zip

This is made from the following pieces of information:

BLNewspapers - this identifies the file as coming from the British Library

TheSun - this is the title of the newspaper, as found on the Library's catalogue.

0002194 - This is the *NLP*, a unique code given to each title. This code is also found on the Title-level list, in case you want to link the titles from the repository to that dataset.

1850 - The year.

8.3 Contruct a Corpus

At this point, and for the rest of the tutorials in the book, you might want to construct a 'corpus' of newspapers, using whatever criteria you see fit. Perhaps you're interested in a longitudinal study, and would like to download a small sample of years spread out over the century, or maybe you'd like to look at all the issues in a single newspaper, or perhaps all of a single year across a range of titles.

The tutorials will make most sense and produce similar results if your corpus is the same as above: all newspapers in the repository from the year 1855. You can also download a single .zip file with the extracted text from these titles here.

8.4 Bulk extract the files using unzip() and a for() loop

R can be used to unzip the files in bulk, which is particularly useful if you have downloaded a large number of files. It's very simple, there's just two steps. This is useful if you're using windows and have a large number of files to unzip.

First, use `list.files()` to create a vector, called `zipfile` containing the full file paths to all the zip files in the 'newspapers' folder you've just created.

```
zipfiles = list.files("/Volumes/T7/zipfiles/", full.names = TRUE)
zipfiles
```

Now, use this in a loop with `unzip()`.

Loops in R are very useful for automating simple tasks. The below takes each file named in the 'zipfiles' vector, and unzips it. It takes some time.

```
purrr::map(zipfiles, unzip)
```

Once this is done, you'll have a new (or several new) folders in the *project* directory (not the newspapers directory). These are named using a numeric code, called the 'NLP', so they should look like this in your project directory:

To tidy up, put these **back** into the **newspapers** folder.

These files contain the METS/ALTO .xml files with the newspaper text. If you have followed the above and downloaded all newspapers for the year 1855, you should have seven different titles and a few hundred newspaper issues. In the next chapter, you'll extract this text from the .xml and save it in a more convenient format. These final files will form the basis for the following tutorials which process and analyse the text of the newspapers.

9 Make a Text Corpus

Unforunately, downloading the titles as explained in Chapter 8 is not the final step in having newspaper data to run analysis on. If you download and extract a single .zip file, you'll see the newspapers themselves are not simply a set of text files ready to use.

First of all, each issue is contained within its own folder, named by its day and month of publication. For example, an issue published on the first of January 1850 will be contained in a folder called 0101. Within this folder 0101, you'll see some more files. These are the METS/ALTO files produced by the OCR process. They are the output which contains the text of the newspapers, but also detailed information on the layout and sections of the newspapers.

Most typical computational or digital humanities uses, such as counting word frequencies or generating word embeddings, will ultimately expect plain text as the input. Therefore, the first stage of using this data is to extract the plain text from the complicated structure of the METS/ALTO. This chapter presents one way of doing this, directly through R. There is also an existing tool called Alto2Text, created by the Living with Machines project, which will do the same in a quicker and more robust way.

In the British Library, the METS file contains information on *textblocks*. Each textblock has a code, which can be found in one of the ALTO files - of which there are one per page. The ALTO files list each individual word in each textblock. The METS file also contains information on which textblocks make up each article, which means that the newspaper can be recreated, article by article. The output will be a csv for each issue - these can be combined into a single dataframe afterwards, but it's nice to have the .csv files themselves first of all.

9.1 Folder structure

Download and extract the newspapers you're interested in, and put them in the same folder as the project you're working on in R.

The folder structure of the newspapers is [nlp]->year->issue month and day-> xml files. The nlp is a unique code given to each digitised newspaper. This makes it easy to find an individual issue of a newspaper.

Load some libraries: all the text extraction is done using tidyverse and furrr for some parallel programming.

```
require(furrr)
require(tidyverse)
library(tidytext)
library(purrr)
```

There are two main functions: `get_page()`, which extracts words and their corresponding textblock, and `make_articles()`, which extracts a table of the textblocks and corresponding articles, and joins them to the words from `get_page()`. get_page() also cleans up the text, removing words in super and sub-script, for example. This is because within the .xml, these words are duplicated so can be safely removed. It also replaces the .xml which indicates split words, with a hyphen.

Here's `get_page()`:

```
get_page = function(alto){
 page = alto %>%  read_file() %>%
        str_split("\n", simplify = TRUE) %>%
        keep(str_detect(., "CONTENT|<TextBlock ID=")) %>%
        str_extract("(?<=CONTENT=\")(.*?)(?=WC)|(?<=<TextBlock ID=)(.*?)(?= HPOS=)")%>%
        discard(is.na) %>%
    as.tibble() %>%
    mutate(pa = ifelse(str_detect(value, "pa[0-9]{7}"),
                       str_extract(value, "pa[0-9]{7}"), NA)) %>%
    fill(pa) %>%
    filter(str_detect(pa, "pa[0-9]{7}")) %>%
    filter(!str_detect(value, "pa[0-9]{7}"))%>%
   mutate(value = str_remove_all(value,
                                 "STYLE=\"subscript\" ")) %>%
   mutate(value = str_remove_all(value,
                                 "STYLE=\"superscript\" "))%>%
   mutate(value = str_remove_all(value,
                                 "\"")) %>%
   mutate(value = str_replace_all(value,
                                 ' SUBS_TYPE=HypPart1 SUBS_CONTENT=.*', '-'))%>%
   mutate(value = str_remove_all(value,
                                 ' SUBS_TYPE=HypPart2 SUBS_CONTENT=.*'))
}
```

If you want to understand how it works, I have broken the function down into components below.

First read the alto page, which should be an argument to the function. Here's one page to use as an example:

```
alto = "newspapers/0002194/1855/0101//0002194_18550101_0001.xml"

altofile = alto %>%  read_file()
```

Split the file on each new line, resulting in a character vector of the length of the number of lines in the page:

```
altofile = altofile %>%
        str_split("\n", simplify = TRUE)

altofile %>% glimpse()
```

Just keep lines which contain either a `CONTENT` or `TextBlock` tag. This

```
altofile = altofile %>% keep(str_detect(., "CONTENT|<TextBlock ID="))

altofile %>% glimpse()
```

Turn it into a dataframe (a tibble in this case):

```
altofile = altofile %>%
  str_extract("(?<=CONTENT=\")(.*?)(?=WC)|(?<=<TextBlock ID=)(.*?)(?= HPOS=)") %>%
        #discard(is.na) %>%
  as_tibble()

altofile %>% head(20)
```

This dataframe has a single column, containing every textblock, textline and word in the ALTO file. Now we need to extract the textblock IDs, put them in a separate column, and then fill() each textblock ID down until it reaches the next one.

```
altofile = altofile %>%
  mutate(pa = ifelse(str_detect(value,
                                "pa[0-9]{7}"),
                     str_extract(value, "pa[0-9]{7}"), NA)) %>%
    fill(pa)
```

The final step removes the textblock IDs from the column which should contain only words, and cleans up some .xml tags we don't want:

```
altofile = altofile %>%
    filter(str_detect(pa, "pa[0-9]{7}")) %>%
    filter(!str_detect(value, "pa[0-9]{7}"))%>%
   mutate(value = str_remove_all(value,
                                 "STYLE=\"subscript\" ")) %>%
   mutate(value = str_remove_all(value,
                                 "STYLE=\"superscript\" "))%>%
   mutate(value = str_remove_all(value,
                                 "\"")) %>%
   mutate(value = str_replace_all(value,
                                 ' SUBS_TYPE=HypPart1 SUBS_CONTENT=.*', '-'))%>%
   mutate(value = str_remove_all(value,
                                 ' SUBS_TYPE=HypPart2 SUBS_CONTENT=.*'))
```

The final output is a dataframe with individual words on one side and the text block IDs on the other.

```
head(altofile)
```

This is the second function:

```
make_articles <- function(foldername){

  files <- list.files(foldername, full.names = TRUE)

  csv_files_exist <- any(xfun::file_ext(files) == "csv")

  if (!csv_files_exist) {

   metsfilename =  str_match(list.files(path = foldername,
                                        all.files = TRUE,
                                        recursive = TRUE,
                                        full.names = TRUE),
                             ".*mets.xml") %>%
     discard(is.na)

    csvfoldername = metsfilename %>% str_remove("_mets.xml")

    metsfile = read_file(metsfilename)

    page_list =  str_match(list.files(path = foldername,
                                      all.files = TRUE,
```

```
                                        recursive = TRUE,
                                        full.names = TRUE),
                                ".*[0-9]{4}.xml") %>%
    discard(is.na)



        metspagegroups = metsfile %>%
          str_split("<mets:smLinkGrp>")%>%
    flatten_chr() %>%
    as_tibble() %>%
          filter(str_detect(value, '#art[0-9]{4}')) %>%
          mutate(articleid = str_extract(value,"[0-9]{4}"))


     t = future_map(page_list, get_page)
     t = t[sapply(t, nrow) > 0]
     t %>%
       bind_rows()  %>%
       left_join(extract_page_groups(metspagegroups$value) %>%
                                       unnest() %>%
        mutate(art = ifelse(str_detect(id, "art"),
                            str_extract(id, "[0-9]{4}"), NA)) %>%
        fill(art) %>%
          filter(!str_detect(id,
                             "art[0-9]{4}")),
        by = c('pa' = 'id')) %>%
      group_by(art) %>%
      summarise(text = paste0(value, collapse = ' ')) %>%
       mutate(issue_name = metsfilename ) %>%
       write_csv(path = paste0(csvfoldername, ".csv"))

  } else {
    message(cat("Skipping folder:", foldername, "- .csv files already exist.\n"))
  }


}
```

It's a bit more complicated, and a bit of a fudge. Because there are multiple ALTO pages for one METS file, we need to read in all the ALTO files, run our `get_pages()` function on them within *this* function, bind them altogether, and then join that to a METS file which contains

an article ID and all the corresponding textBlocks. Again, if you're interested, the function has been broken down into components below. You can ignore this section if you just want to run the function and extract text from your own files.

The function takes an argument called 'foldername'. This folder should correspond to the folders within the downloaded newspaper files from the BL repository. Later, we can pass a *list* of folder names to the function using `lapply()` or `future_map()`, and it will run the function on each folder in turn.

This is how it works with a single folder:

```
foldername = "newspapers/0002194/1855/0101/"
```

Using the folder name as the last part of the file path, and then a regular expression to get only a file ending in mets.xml, this will get the correct METS file name and read it into memory:

```
metsfilename =  str_match(list.files(path = foldername, all.files = TRUE, recursive = TRUE
    discard(is.na)

metsfilename
```

```
metsfile = read_file(metsfilename)
```

We also need to call the .csv (which we're going to have as an output) a unique name:

```
csvfoldername = metsfilename %>% str_remove("_mets.xml")
```

Next we have to grab all the ALTO files in the same folder, using the same method:

```
page_list =  str_match(list.files(path = foldername, all.files = TRUE, recursive = TRUE, f
    discard(is.na)
```

Next we need the file which lists all the pagegroups and corresponding articles.

```
metspagegroups = metsfile %>%
  str_split("<mets:smLinkGrp>") %>%
    flatten_chr() %>%
    as_tibble() %>%
  filter(str_detect(value, '#art[0-9]{4}')) %>%
  mutate(articleid = str_extract(value,"[0-9]{4}"))
```

The next bit uses a function written by brodrigues called extractor()

```
extractor <- function(string, regex, all = FALSE){
    if(all) {
        string %>%
            str_extract_all(regex) %>%
            flatten_chr() %>%
            str_extract_all("[:alnum:]+", simplify = FALSE) %>%
            purrr::map(paste, collapse = "_") %>%
            flatten_chr()
    } else {
        string %>%
            str_extract(regex) %>%
            str_extract_all("[:alnum:]+", simplify = TRUE) %>%
            paste(collapse = " ") %>%
            tolower()
    }
}
```

We also need another function which extracts the correct pagegroups:

```
extract_page_groups <- function(article){

    id <- article %>%
        extractor("(?<=<mets:smLocatorLink xlink:href=\"#)(.*?)(?=\" xlink:label=\")",
                  all = TRUE)

    type =
    tibble::tribble(~id,
                    id)
}
```

Next this takes the list of ALTO files, and applies the get_page() function to each item, then binds the four files together vertically. I'll give it a random variable name, even though it doesn't need one in the function because we just pipe it along to the csv.

```
t = future_map(page_list, get_page)
t = t[sapply(t, nrow) > 0]
t = t %>%
  bind_rows()

head(t)
```

This extracts the page groups from the mets dataframe we made, and turns it into a dataframe

with the article ID as a number, again extracting and filtering using regular expressions, and using fill(). The result is a dataframe of every word, plus their article and text block.

```
t = t %>%
  left_join(extract_page_groups(metspagegroups$value) %>%
                                    unnest() %>%
        mutate(art = ifelse(str_detect(id, "art"),
                            str_extract(id,
                                        "[0-9]{4}"), NA))%>%
          fill(art),
        by = c('pa' = 'id')) %>%
  fill(art)

head(t, 50)
```

Next we use summarise() and paste() to group the words into the individual articles, and add the mets filename so that we also can extract the issue date afterwards.

```
 t = t %>%
    group_by(art) %>%
  summarise(text = paste0(value, collapse = ' ')) %>%
       mutate(issue_name = metsfilename )

head(t, 10)
```

And finally write to .csv using the csvfoldername we created:

```
t %>%
       write_csv(path = paste0(csvfoldername, ".csv"))
```

To run it on a bunch of folders, you'll need to make a list of paths to all the issue folders you want to process. You can do this using `list_dirs`. You *only* want these final-level issue folders, otherwise it will try to work on an empty folder and give an error. This means that if you want to work on multiple years or issues, you'll need to figure out how to pass a list of just the issue level folder paths.

In this case, I used the package `fs`:

```
library(fs)


get_all_deepest_folders <- function(folder_path) {
```

```
  if (!file.exists(folder_path) || !file.info(folder_path)$isdir) {
    stop("Invalid folder path or folder does not exist.")
  }

  find_deepest_folders_recursive <- function(dir_path) {
    subdirs <- list.dirs(dir_path, full.names = TRUE, recursive = FALSE)

    if (length(subdirs) == 0) {
      return(dir_path)
    }

    deepest_subdirs <- character(0)
    for (subdir in subdirs) {
      deepest_subdirs <- c(deepest_subdirs, find_deepest_folders_recursive(subdir))
    }

    return(deepest_subdirs)
  }

  deepest_folders <- find_deepest_folders_recursive(folder_path)
  return(unique(deepest_folders))
}
```

```
starting_folder <- "../../../Downloads/TheSun_sample/"
deepest_folders <- get_all_deepest_folders(starting_folder)
```

Finally, this applies the function make_articles() to everything in the folderslist vector. It will write a new .csv file into each of the folders, containing the article text and codes. You can add whatever folders you like to a vector called folderlist, and it will generate a csv of articles for each one.

```
future_map(deepest_folders, make_articles, .progress = TRUE)
```

It's not very fast (I think it can take 10 or 20 seconds per issue, so bear that in mind), but *eventually* you should now have a .csv file in each of the issue folders, with a row per line.

These .csv files can be re-imported and used for text mining tasks such as:

- word frequency count
- tf-idf scores
- sentiment analysis
- topic modelling
- text reuse

10 N-gram Analysis

The first thing you might want to do with a large dataset of text is to count the words within it. Doing this with newspaper data may help to discover and quantify trends, understand more about events, and make comparisons between the language in several titles. In this tutorial, we'll use text mining to look for changing patterns in word use across the months of a single year.

We should also think about the specific nature of each newspaper, and how this will affect the results. These are from a single year, but some are much more frequent than others. One, *The Sun*, is a daily, and so much more of the data comes from this title. They will also have different levels of OCR errors.

For this tutorial, you'll make use of a few more R packages: `tidyverse`, which you should already have installed, `data.table`, and `tidytext`. If necessary, install these using the `install.packages()` command:

```
install.packages('tidyverse')
install.packages('data.table')
install.packages('tidytext')
install.packages('tidytable')
```

Once this is done (or if you have them installed already, load them:

```
library(tidyverse)
library(data.table)
library(tidytext)
library(tidytable)
```

10.1 Load the news dataframe and relevant libraries

For this tutorial, you'll need a set of .csv files containing newspaper article text in a specific format. Chapter 8 and Chapter 9 walk through the processing of downloading and creating these files. If you want to construct your own newspaper corpus and use it for this chapter, I recommend going back and checking those out. Alternatively, if you want to use a ready-made corpus, you can download a compressed file containing all the articles from a single year on the

repository: 1855. This file is available on Zenodo. Once you have downloaded it, decompress it and make a note of where it is stored on your local machine.

The first step is to load all these files and turn them into a single dataframe. The command `list.files()`, with the parameters set below, will list all the files with the text `csv` in them: Swap the `path=` parameter for the location you have saved the .csv files, if appropriate.

```
news_sample_dataframe = list.files(path = "newspaper_text",
                                   pattern = "csv",
                                   recursive = TRUE,
                                   full.names = TRUE)
```

Next, import the files themselves. The function `lapply` is a bit like a loop: it will take list or a vector (in this case, a vector containing the file names we want to import), and run the same function over them all, storing the result as a list. In this case, we'll run the function `fread`, which will read a single file into R as a dataframe. Passing it a list of filenames means it will read all of them into dataframes, and store them as a list of dataframes.

With this list of dataframes, we'll use another function `rbindlist` to transform them from a list of dataframes to a single, long dataframe. Essentially, merging them together.

```
all_files = lapply(news_sample_dataframe, data.table::fread)

names(all_files) = news_sample_dataframe

all_files_df = data.table::rbindlist(all_files, idcol = 'filename')
```

Make a new object, `news_df`, which takes the information about the newspaper found in the filename, and uses it as metadata, stored in columns.

One first step is to get the actual title names for the titles. The information in the .csv only contains a unique code, the NLP. To do this, we'll make a small dataframe with the title names and NLP codes, and then join this to the data.

The result is a dataframe with a column for the issue date, the article number, and the full article text.

```
title_names_df = tibble(newspaper_id = c('0002090', '0002194', '0002244', '0002642', '0002

news_df = all_files_df %>%
  mutate(filename = basename(filename))


news_df = news_df %>%
```

```
  separate(filename,
           into = c('newspaper_id', 'date'), sep = "_") %>% # separate the filename into t
  mutate(date = str_remove(date, "\\.csv")) %>% # remove .csv from the new data column
  select(newspaper_id, date, art, text) %>%
  mutate(date = ymd(date)) %>% # turn the date column into date format
  mutate(article_code = 1:n()) %>% # give every article a unique code
  select(article_code, everything()) %>%
  left_join(title_names_df, by = 'newspaper_id')# select all columns but with the article
```

10.2 Text Analysis with Tidytext

The package we're going to use for analysing the text is called 'tidytext'. This package has many features for understanding and working with text, such as tokenising (splitting into words) and calculating the tf-idf scores of words in a group of documents. The authors of the package have published an entire book, Text Mining with R, which is a very good introduction to text analysis with R.

It's a little different to the approach taken by, say, python, because it works on the principle of turning text into dataframes, in a format which is easy to work with in R. When we tokenise, for example, tidytext will create a dataframe which contains one row for each token. This is then easy to count, sort, filter, and so forth, using standard tidyverse tools.

Simple word counts as a method are increasingly outdated, and modern text analysis is much more likely to use more sophisticated metrics and take the context into account. But a simple statistical analysis of text is still a useful and quick way of understanding a group of documents and getting an overview of their contents. It also is often the first step in further analysis, such as topic modelling or word embeddings.

As a first step, take a look at the first few entries in the dataframe:

```
news_df %>% mutate(text = str_trunc(text, 500)) %>% # show just the first part of the text
  head(5) %>% # show the first 5 rows
  kableExtra::kbl()
```

article_code	newspaper_id	date	art	text
1	0002090	1855-06-19	1	esP e allall J o Po"' .d. o f f t/' ea tice 11, P ate° s' i
2	0002090	1855-06-19	2	Eqp t , I , WILLIAMSON-SQUARE. i'lle . AL, — elott„t
3	0002090	1855-06-19	3	Tri C 3JI all 1101 Tau atl) AND GENERAL COMMERCI
4	0002090	1855-06-19	4	A NNIVERSARY OF THE NATIONAL SCHOOLS. The A
5	0002090	1855-06-19	5	A LOT OF THE VERY BEST F RENCH PRINTED MUS

We can see that it has a number of rows (about 97,000 if you're using the data from the previous tutorials) and 5 columns. Each row is a different article, and the fifth column `text`, contains the full text of that article. You'll also probably notice that the text itself is pretty garbled, because of OCR errors. It's worth pointing out that because we're looking at the first few articles, which are usually advertisements, the OCR is likely to be much worse than with ordinary articles within the paper.

As a first task, we'll simply use the tidytext package to make a count of the words found in the data.

10.3 Tokenise the text using unnest_tokens()

The first step is to *tokenise* the text. This is the starting point for many of the basic text analyses which will follow. Tokenising simply divides the text into 'tokens': smaller, equally-sized 'units' of some text. A unit is often a word, but could be a bigram (a sequence of two consecutive words), or a trigram, a sequence of three consecutive words.

To do this using the library `tidytext`, we will pass the dataframe of text to the function `unnest_tokens()`. This function takes a column of text in a dataframe and splits it into tokens. The function has a set of default values, but, as we will see, we can change the function to create other types of functions.

To understand what tokenising is doing, I'll make a dataframe containing a single 'article', which in this case is a single sentence.

```
df = tibble(article_code = 1, text = "The quick brown fox jumped over the lazy dog.")

df %>%
  kableExtra::kbl()
```

article_code	text
1	The quick brown fox jumped over the lazy dog.

To tokenise this dataframe, we'll use the `unnest_tokens` function. The two most important parameters to `unnest_tokens` are `output` and `input`. This is fairly self-explanatory. Pass the name of the column you'd like to tokenise as the `input`, and the name you would like to give the tokenised words as the `output`.

```
df %>%
  unnest_tokens(output = word, input = text) %>%
  kableExtra::kbl()
```

article_code	word
1	the
1	quick
1	brown
1	fox
1	jumped
1	over
1	the
1	lazy
1	dog

Run this code, and you'll see that the dataframe has been transformed. Each word in the sentence is now on a separate row, in a new column called `word`. Also note that the other data (in this case the article code) is kept and duplicated.

You can also specify an argument for `token`, allowing you to split the text into sentences, characters, lines, or n-grams. If you split into `n-grams`, you need to use the argument `n=` to specify how many consecutive words you'd like to use.

Like this:

```
df %>%
  unnest_tokens(output = word,
                input = text,
                token = 'ngrams',
                n =3) %>%
  kableExtra::kbl()
```

article_code	word
1	the quick brown
1	quick brown fox
1	brown fox jumped
1	fox jumped over
1	jumped over the
1	over the lazy
1	the lazy dog

You can also use other tokenizers such as character shingles, or supply your own method for splitting the text, such as on new lines, if you have them in your text:

```
df = tibble(article_code = 1, text = "The quick brown fox\njumped over the lazy dog.")
```

```
df %>%
  unnest_tokens(output = word,
                input = text, token = stringr::str_split, pattern = "\n") %>%
  kableExtra::kbl()
```

article_code	word
1	the quick brown fox
1	jumped over the lazy dog.

Now, it's time to do this to our article text. Create a new object, `news_tokens`, using `unnest_tokens()`, passing the text column as the input column:

```
news_tokens = news_df %>% unnest_tokens(output = word, input = text)

news_tokens %>% head(10) %>%
  kableExtra::kbl()
```

article_code	newspaper_id	date	art	newspaper_title
1	0002090	1855-06-19	1	The Liverpool Standard And General Commercial Advertis
1	0002090	1855-06-19	1	The Liverpool Standard And General Commercial Advertis
1	0002090	1855-06-19	1	The Liverpool Standard And General Commercial Advertis
1	0002090	1855-06-19	1	The Liverpool Standard And General Commercial Advertis
1	0002090	1855-06-19	1	The Liverpool Standard And General Commercial Advertis
1	0002090	1855-06-19	1	The Liverpool Standard And General Commercial Advertis
1	0002090	1855-06-19	1	The Liverpool Standard And General Commercial Advertis
1	0002090	1855-06-19	1	The Liverpool Standard And General Commercial Advertis
1	0002090	1855-06-19	1	The Liverpool Standard And General Commercial Advertis
1	0002090	1855-06-19	1	The Liverpool Standard And General Commercial Advertis

The result is a very large dataset of words - one row for each word in the dataset, a total of about 66 million using the tutorial data.

10.3.1 Speeding things up with {Tidytable}

The next step is to use `tidyverse` commands to count and analyse the data. However, doing this with a dataframe of 66 million rows is not ideal. For this, we'll introduce a new package, called `tidytable`. Tidytable allows us to use tidyverse verbs, but it translates them into another package, `data.table`, behind the scenes. Data.table is much faster in most cases.

To use the tidytable equivalent to a tidyverse verb, add a period (`.`) just before the parentheses. For example, `mutate()` becomes `mutate.()`

Note that tidytable does not have a `group_by` command. Instead, you'll use the parameter `.by =` within another command to specify the group you want it to apply to.

Once this is done, it is relatively easy to count and analyse the data using standard tidyverse verbs. The following will count the instances of each word and show them in descending order:

```
news_tokens %>%
  summarise.(n = n.(), .by = word) %>%
  arrange.(desc.(n)) %>% head(20) %>%
  kableExtra::kbl()
```

word	n
the	4483357
of	2473323
to	1674183
and	1625027
a	1097019
in	977684
that	593476
at	530306
i	529302
for	471661
be	467150
was	456650
on	454623
is	453208
by	415020
it	413959
with	341337
e	334563
t	332604
1	329737

As you can see, the top words are entirely made up of short, common words such as `the`, `of`, `i`, and so forth. These are unlikely to tell us much about the text or reveal patterns about the content (though they may have other uses, for example for identifying authors, but let's ignore that for now).

To get something more meaningful out of these top results, it's probably best to do some text cleaning.

When doing your own research, particularly using sources such as newspapers which will often

look quite different and have messy OCR, you'll often need to go back and forth, checking and adding additional cleaning and pre-processing steps to get something meaningful.

In this case, we'll remove these short, common words (known as 'stop words'), and also, later, do some more text cleaning.

10.4 Removing stop words

To do this, we load a dataframe of stopwords, which is included in the tidytext package:

```
data("stop_words")
```

This will load a dataframe called `stop_words` into the R environment. This dataframe contains a column called `word`. We want to merge this to our tokenised data, and remove any matches.

Next use the function `anti_join()`. This basically removes any word in our word list which is also in the stop words list:

```
news_tokens = news_tokens %>%
  anti_join(stop_words)
```

```
Joining with `by = join_by(word)`
```

Let's take another look at the top words, using the same code as above:

```
news_tokens %>%
  summarise.(n = n.(), .by = word) %>%
  arrange.(desc.(n)) %>% head(20)%>%
  kableExtra::kbl()
```

word	n
1	329737
4	205696
0	200869
amp	152582
3	99579
6	84694
11	84643
day	84447
2	81151
5	80543
lord	78383
street	75711
10	73835
time	68863
st	68110
7	67775
london	67363
house	59095
war	56102
de	53433

Now the stopwords are removed, we can use the `filter()` command to remove any other unwanted tokens: numbers are also particularly common in newspapers (these titles often publish lists of stocks and shipping information, for example) and don't tell us anything about the texts. Furthermore, ssome horizontal lines have been picked up by the OCR as punctuation. Let's filter both of these out:

```
news_tokens = news_tokens %>%
  filter(!str_detect(word, "[0-9]")) %>%
  filter(!str_detect(word, "__"))
```

The command `str_detect()` within 'filter() removes any word which matches a given regular expressions pattern. In this case, the pattern is *simply any occurence of a number*. That's a bit blunt, but it will be effective at least.

Let's check the top words again:

```
news_tokens %>%
  summarise.(n = n.(), .by = word) %>%
  arrange.(desc.(n)) %>% head(20) %>%
  kableExtra::kbl()
```

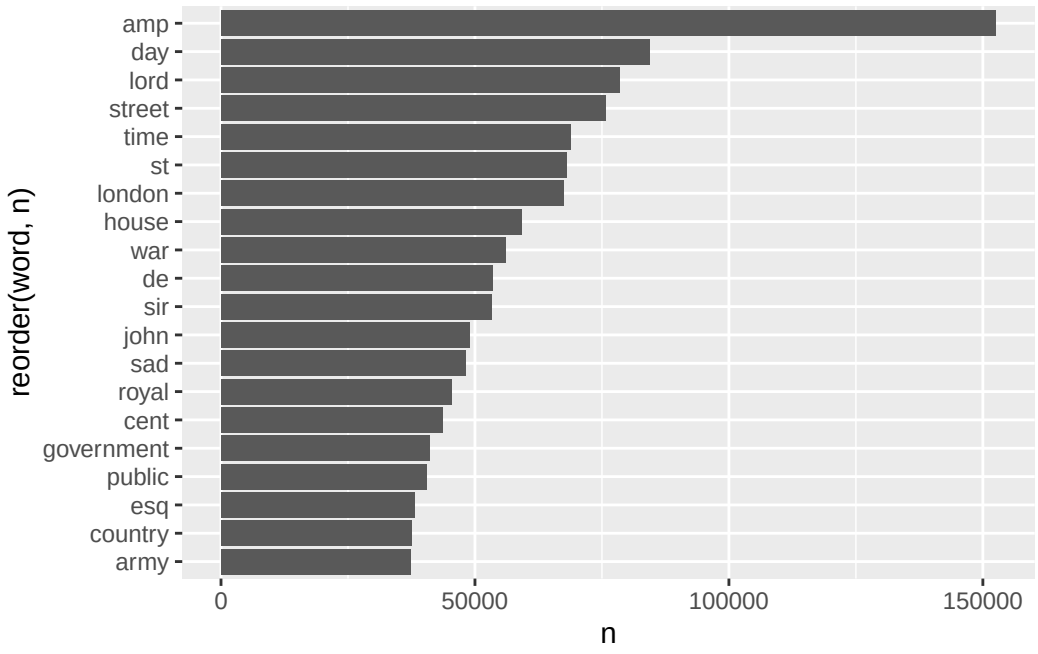
word	n
amp	152582
day	84447
lord	78383
street	75711
time	68863
st	68110
london	67363
house	59095
war	56102
de	53433
sir	53288
john	49003
sad	48236
royal	45438
cent	43548
government	41093
public	40450
esq	38063
country	37544
army	37313

The list looks a bit more sensible now, with words which are plausibly found often particularly within news sources.

10.5 Visualising using ggplot

The next step is to visualise this, which is very easy to do using `ggplot2`.

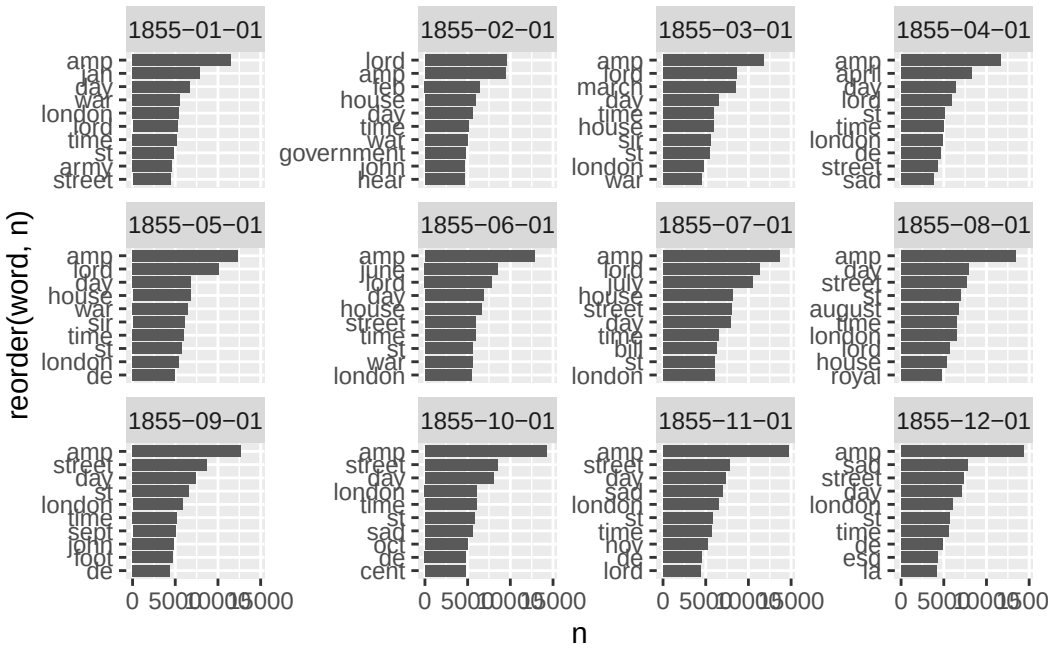
```
news_tokens %>%
  summarise.(n = n.(), .by = word) %>%
  arrange.(desc.(n)) %>% head(20) %>%
  ggplot() + geom_col(aes(x = reorder(word, n),y = n)) + coord_flip()
```



As well as a basic count of everything, we can use tidyverse/tidytable to do some more specific counts and visualisations.

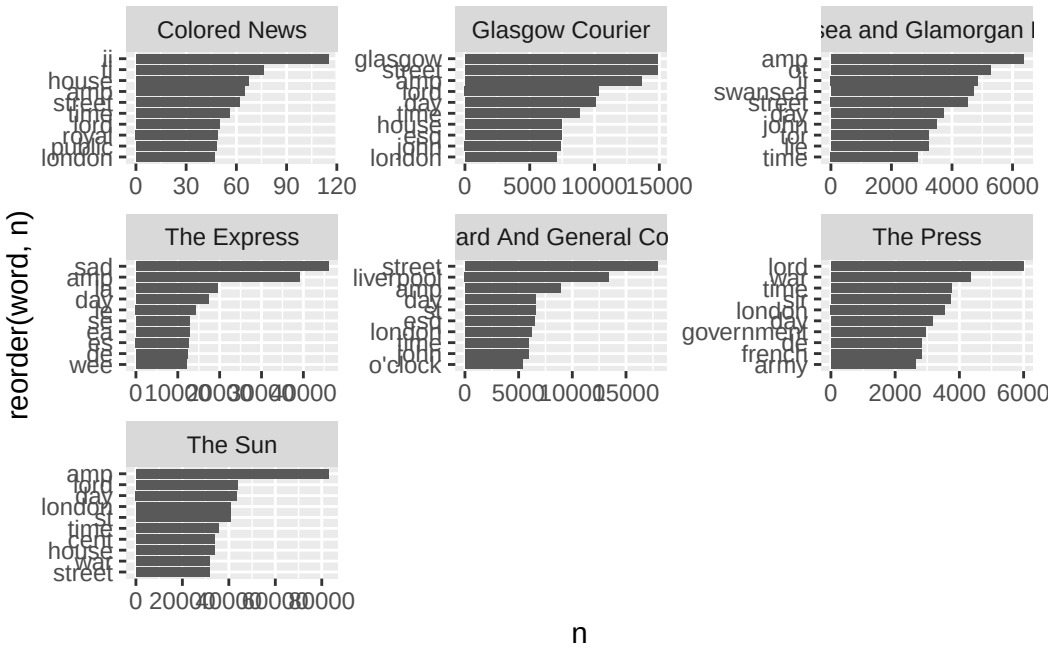
For example, a count of the top five words for each month in the data. For this, we need to create a 'month' column, and use `facet_wrap()` from ggplot2 to graph each month separately. In order to have the results show correctly, we make use of the function `reorder_within` from the tidytext package. This is to ensure that the words are properly ordered within each facet, rather than overall.

```
news_tokens %>%
  mutate(month = as.character(cut(date, 'month')))%>%
  summarise.(n = n.(), .by = c(month, word)) %>%
  group_by(month) %>%
  slice_max(order_by = n, n = 10)  %>%
    mutate(month = as.factor(month),
           word = reorder_within(word, n, month))%>%
  ggplot() +
  geom_col(aes(x = reorder(word, n), y = n)) +
  coord_flip() +
  facet_wrap(~month, scales = 'free_y')+
    scale_x_reordered()
```



We can also get the top words per newspaper title:

```
news_tokens %>%
  summarise.(n = n.(), .by = c(newspaper_title, word)) %>%
  group_by(newspaper_title) %>%
  slice_max(order_by = n, n = 10) %>%
    mutate(newspaper_title = as.factor(newspaper_title),
           word = reorder_within(word, n, newspaper_title))%>%
  ggplot() +
  geom_col(aes(x = reorder(word, n), y = n)) +
  coord_flip() +
  facet_wrap(~newspaper_title, scales = 'free')+
    scale_x_reordered()
```



We can see some differences between the titles, although many of the word lists are quite similar. Many of these are typical words which appear in advertisements or report - words related to times or places (street, day, clock, etc.). There are also regional differences, with the place associated with the title (`glasgow`, `swansea`, `liverpool` etc.) also showing up. One title, 0002645, seems to have more words related to 'serious' news (`war`, `government`). Some further cleaning or adding words to the stop word list would be helpful too, clearly.

10.5.1 Change over time

Another thing to look at is the change in individual words over time. 'War' is a common word: did its use change over the year? To do this, we first filter the token dataframe, using `filter()` to keep only the word (or words) we're interested in.

In many cases (as definitely here), the spread of data over the entire period is not even - some months have many more words than others. For an analysis to be in any way meaningful, you should think of some way of normalising the results, so that the number is of a percentage of the total words in that title, for example. The raw numbers may just indicate a change in the total volume of text.

We'll do this with an extra step. First, make a count of the total number of each word, per week. Second, make a new column which divides the total per word, by the total number of words per week. This number is the frequency - basically what proportion of the total words for that week is a particular word. Lastly, filter to just the word of interest:

10.5.2 Words over time

```
news_tokens %>%
    mutate.(week = ymd(cut(date, 'week'))) %>%
    summarise.(n = n(), .by = c(word, week))  %>%
    mutate.(freq = n/sum(n), .by = week) %>%
    filter.(word == 'war') %>%
    ggplot() + geom_col(aes(x = week, y = freq))
```

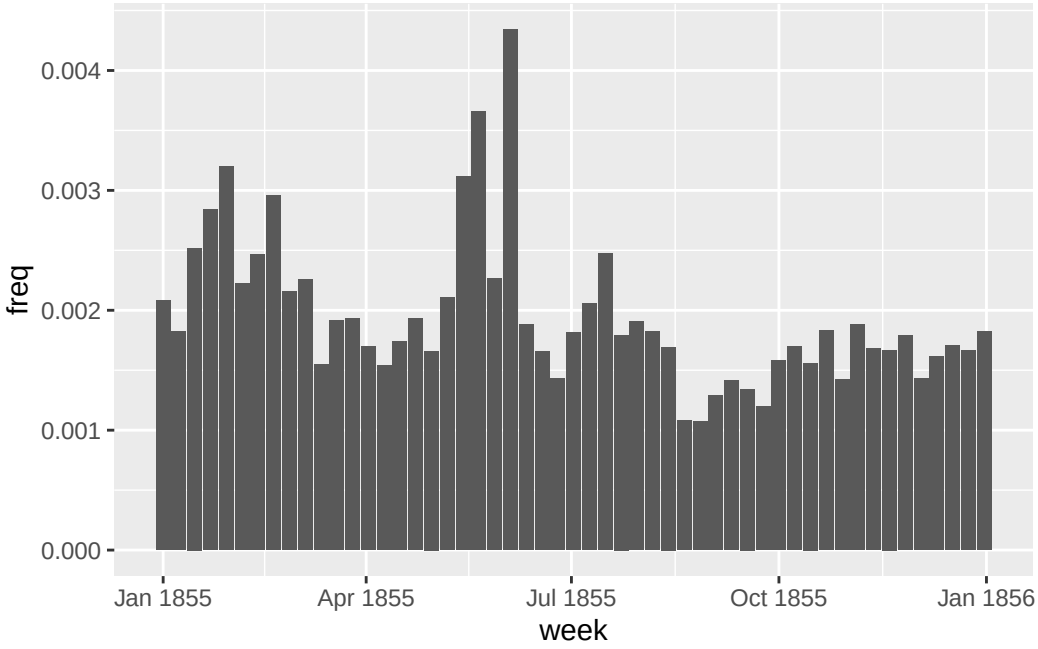
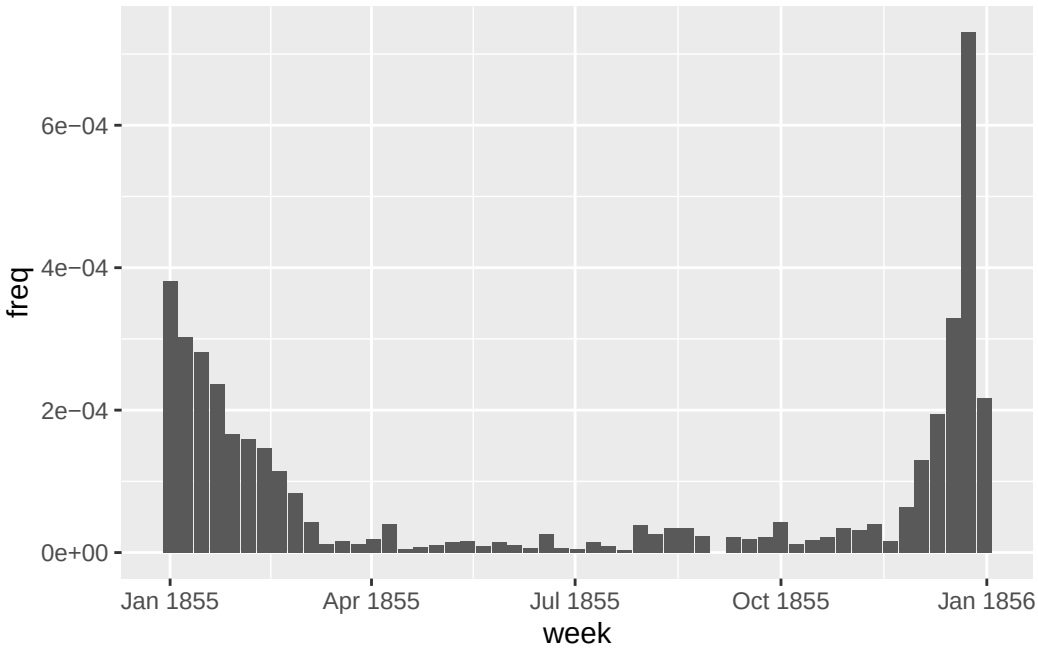


Figure 10.1: Chart of the Word 'ship' over time

We can also look at very seasonal words, to test whether it really makes sense:

```
news_tokens %>%
    mutate.(week = ymd(cut(date, 'week'))) %>%
    summarise.(n = n(), .by = c(word, week))  %>%
    mutate.(freq = n/sum(n), .by = week) %>%
    filter.(word == 'christmas') %>%
    ggplot() + geom_col(aes(x = week, y = freq))
```



Unsurprisingly, there is a seasonal pattern to the word `christmas` in the dataset.

Counting tokens like this in many cases says more about the dataset and its collection than anything about the content or the historical context. In fact, many of the words seem to be coming from advertisements rather than news articles. In a future chapter, we'll build a classifier to detect these and remove them.

10.6 Tf-idf

This section deals uses R and tidytext to do another very typical word frequency analysis, known as the **tf-idf** score. This is a measurement of how 'unique' a word is in a given document, by comparing its frequency in one document to its frequency overall. Counting tokens, as above, will generally result in a listen of very popular words, which occur very often in all newspapers, and so don't really give any interesting information. Using a metric such as tf-idf can be a way to understand the most 'significant' words within a given document.

In this case, the word *document* can be misleading. It could be a single issue or article, or it could be something completely different, depending on our needs. A document can be any way of splitting up the text. For instance, we could consider all articles from a given month as a single 'document', and then calculate the words most unique to that month. This might give us a better understanding of what unique topics were being discussed at a particular time in the newspapers.

To do this, we use a function from tidytext called `bind_tf_idf`. This function expects a list of words per document and a raw count. We'll do this as a first step. To keep things simple, we'll use the newspaper ID as the 'document', meaning the metric should find words unique to each title. First, create a new object `news_tokens_count`, which contains the counts of the words in each newspaper ID:

```
news_tokens_count = news_tokens %>%
  count(newspaper_id, word)

news_tokens_count %>% head(10) %>%
  kableExtra::kbl()
```

newspaper_id	word	n
0002090	_a	45
0002090	_a_	1
0002090	_aa	2
0002090	_aaa	1
0002090	_aacislain	1
0002090	_aad	1
0002090	_abo	1
0002090	_about	2
0002090	_acaltitash	1
0002090	_accept	1

Next, use the `bind_tf_idf` function from tidytext. This needs to be passed the word column (`term`), the `document` column, and the column with the word counts (`n`)

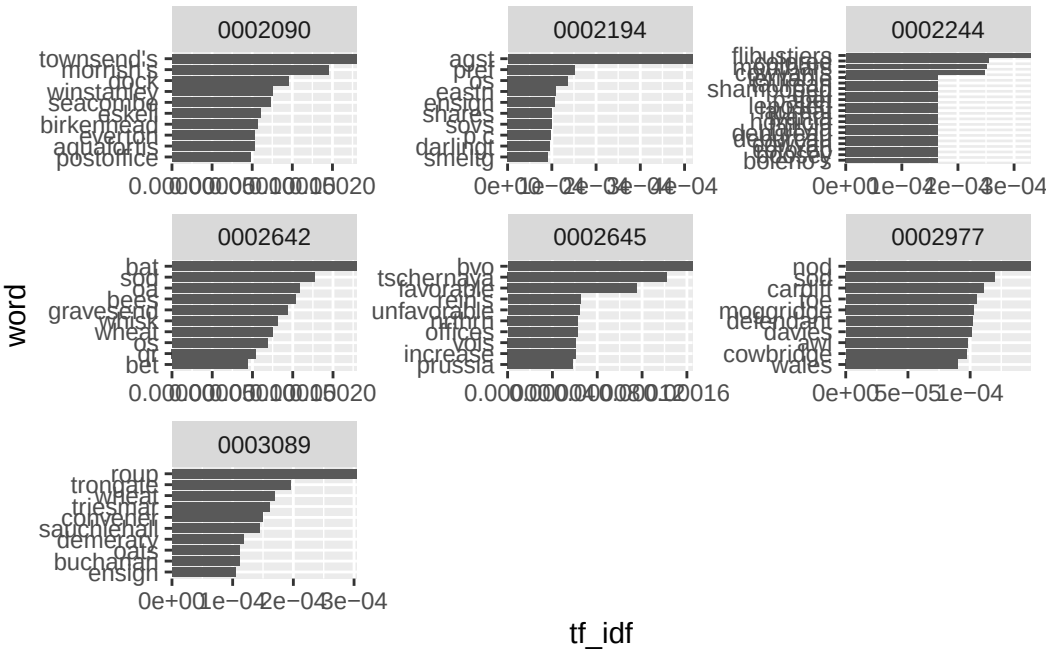
```
news_tfidf = news_tokens_count %>%
  bind_tf_idf(term = word, document = newspaper_id, n = n)

news_tfidf %>% head(10) %>%
  kableExtra::kbl()
```

newspaper_id	word	n	tf	idf	tf_idf
0002090	_a	45	1.73e-05	0.0000000	0e+00
0002090	_a_	1	4.00e-07	0.5596158	2e-07
0002090	_aa	2	8.00e-07	0.8472979	7e-07
0002090	_aaa	1	4.00e-07	1.9459101	7e-07
0002090	_aacislain	1	4.00e-07	1.9459101	7e-07
0002090	_aad	1	4.00e-07	1.2527630	5e-07
0002090	_abo	1	4.00e-07	1.9459101	7e-07
0002090	_about	2	8.00e-07	0.8472979	7e-07
0002090	_acaltitash	1	4.00e-07	1.9459101	7e-07
0002090	_accept	1	4.00e-07	1.9459101	7e-07

Now, we have a new object with new columns. The first is `tf`, which is simply the frequency of that term as a proportion of all words in the document. Next is `idf`, which is the inverse of the frequency of the word over all the documents. The less frequent a word is overall, the larger the number in this column. Third is `tf_idf`, which multiples one by the other.

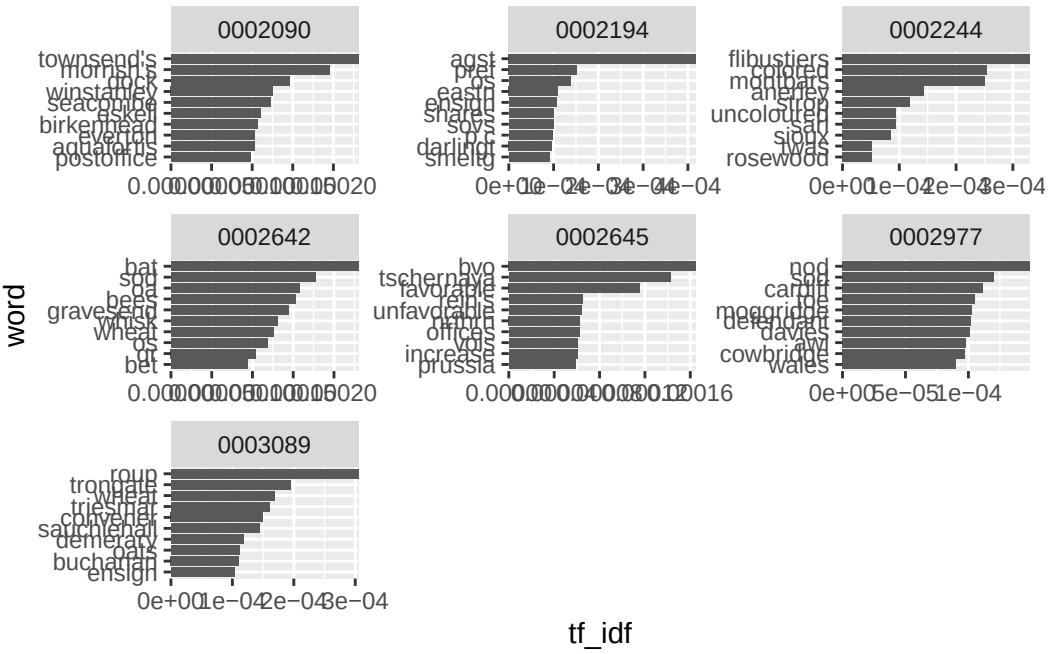
To make use of this, we want to find the words with the highest tf-idf scores for each of the documents. Let's do this and plot the results:

```
news_tfidf %>%
  group_by(newspaper_id) %>%
  slice_max(order_by = tf_idf, n = 10) %>%
  ungroup() %>%
    mutate(newspaper_id = as.factor(newspaper_id),
           word = reorder_within(word, tf_idf, newspaper_id)) %>%
  ggplot() + geom_col(aes(word, tf_idf)) +
  facet_wrap(~newspaper_id, scales = 'free') +
    scale_x_reordered()+
    scale_y_continuous(expand = c(0,0))+ coord_flip()
```



There's one final problem worth considering with using tf-idf. A word can score very highly if it occurs just a couple of times in one document and not at all in others. This may mean that the highest tf-idf words are not actually significant but just extremely rare. One solution to this is to filter so that we only consider words that occur at least a few times in the documents:

```
news_tfidf %>%
  filter(n>3) %>%
  group_by(newspaper_id) %>%
  slice_max(order_by = tf_idf, n = 10) %>%
  ungroup() %>%
    mutate(newspaper_id = as.factor(newspaper_id),
           word = reorder_within(word, tf_idf, newspaper_id)) %>%
  ggplot() + geom_col(aes(word, tf_idf)) +
  facet_wrap(~newspaper_id, scales = 'free') +
    scale_x_reordered()+
    scale_y_continuous(expand = c(0,0))+ coord_flip()
```



The results are still not very satisfactory - at least at first glance, it's hard to get anything about the

10.7 Small case study

To demonstrate how counting frequencies can be used as a form of analysis, this section is a small case study looking at bigrams, that is, pairs of words found in the data. Looking at the immediate context of a word can give some clue as to how it is being used. For example, the word 'board' in the bigram 'board game' has a different meaning to the word board in the bigram 'board meeting'.

In this case study, we'll count bigrams containing the word 'liberal', to see how the meaning of that word changed over time.

Because we want to look at temporal, or diachronic change, we'll need a different datatset to the year 1855 used in the chapter so far. Instead, we'll use a dataset containing all the issues of a single title *The Sun*, for the years 1802 and 1870. These are the earliest and latest years in the data, and using one title means it's more likely we'll have at least slightly consistent results.

I have already extracted the text from these years and they are available as .zip file here. Once you have downloaded this, decompress and put the path to the folder name in the code below. Otherwise, follow the same steps as in the code at the beginning of the chapter.

```
theSun = list.files(path = "../../../Downloads/TheSun_sample/",
                                    pattern = "csv",
                                    recursive = TRUE,
                                    full.names = TRUE)

theSunall_files =   lapply(theSun, data.table::fread)

names(theSunall_files) = theSun

theSunall_files_df = data.table::rbindlist(theSunall_files, idcol = 'filename')

theSunall_files_df = theSunall_files_df %>%
  mutate(filename = basename(filename))

theSunall_files_df = theSunall_files_df %>%
  separate(filename,
           into = c('newspaper_id', 'date'), sep = "_") %>% # separate the filename into t
  mutate(date = str_remove(date, "\\.csv")) %>% # remove .csv from the new data column
  select(newspaper_id, date, art, text) %>%
  mutate(date = ymd(date)) %>% # turn the date column into date format
  mutate(article_code = 1:n()) %>% # give every article a unique code
  select(article_code, everything()) %>%
  left_join(title_names_df, by = 'newspaper_id')# select all columns but with the article
```

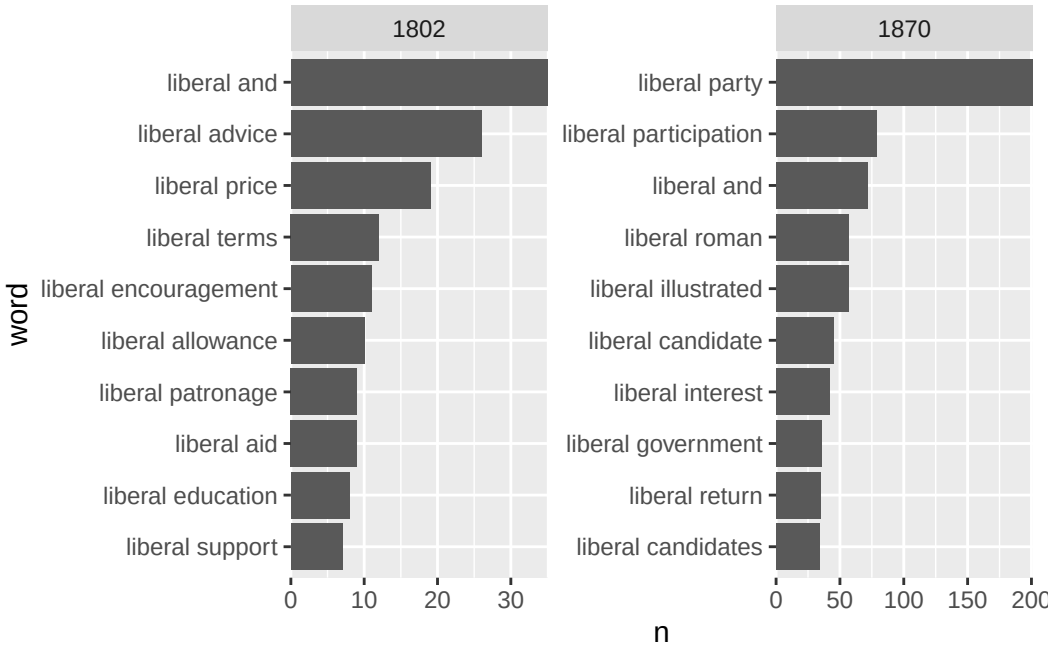
Use `unnest_tokens` to tokenise the data. Set the `n` parameter to 2, which will divide the text into bigrams:

```
theSunNgrams = theSunall_files_df %>%
  unnest_tokens(word, text, token = 'ngrams', n =2)
```

With this new dataset, filter to include only bigrams which contain the word liberal. Count these and visualise the top ten results:

```
theSunNgrams %>%
  filter(str_detect(word, "^liberal ")) %>%
  mutate(year = year(date)) %>%
  count(word, year) %>%
  group_by(year) %>%
  slice_max(order_by = n, n = 10) %>%
    mutate(year = as.factor(year),
           word = reorder_within(word, n, year)) %>%
```

```
ggplot() + geom_col(aes(word, n)) +
facet_wrap(~year, scales = 'free') +
  scale_x_reordered()+
  scale_y_continuous(expand = c(0,0))+ coord_flip()
```



The results point to a huge change in the way that the word liberal is used in these two years of the newspaper. At the beginning of the century, the most common bigrams point to the general meaning of the word liberal, but by the end, the words are all related to liberal as a political ideology and party.

10.8 Further reading

The best place to learn more is by reading the 'Tidy Text Mining' book available at https://www.tidytextmining.com. This book covers a whole range of text mining topics, including those used in the next few chapters.

11 Topic Modelling

Topic modelling tried to statistically discern a number of 'topics' within a set of documents, expressed as a set of significant keywords. The number of topics are set in advance, but the keywords which make up each are found through unsupervised learning. In the case of newspapers, topic modelling may be useful in understanding the subject of the articles within them. Imagine a set of newspapers, with five articles on various subjects. One might be a foreign news story, another sports, and a third some advertisements.

Topic modelling might help us to sort each of these articles together, based on the keywords they use. A typical topic model might be something like the following. We set the number of topics in advance based on our prior knowledge of how we expect the articles to look, in this case, 3 topics. Next the topic model will look at the keywords which make up each document, and, using an unsupervised learning algorithm, will attempt to figure out the set of three topics which best describe the documents as a whole, and then also give a metric for how each document is made up of these topics. The output would look something like:

- **Topic 1 keywords:** score, match, goal, win
- **Topic 2 keywords:** French, war, diplomat, office
- **Topic 3 keywords:** offer, bargain, sale, price

From this list of keywords, its up to the user to determine, if any, the unifying subject of each topic. In this case it's fairly obvious, that topic 1 is sport, topic 2 is foreign news, and so forth, but this may not necessarily be the case.

As well as the topics, the model will tell us the proportion of each topic which makes up each document. The output for our 4 articles will look something like this:

- **Article 1:** topic 1 (.89), topic 2 (.11)
- **Article 2:** topic 2 (.95), topic 3 (.05)
- **Article 3:** topic 3 (.99), topic 3 (.01)
- **Article 4:** topic 1 (.6), topic 3 (.4)

And so forth. The numbers in parentheses represent the mixture of topics within those documents. Most cover one topic primarily, but some, such as article 4, have keywords which indicate they are more of a mixture.

From this output, plus our interpretation of the topics using the keywords above, we can begin to tell something about the makeup of the articles. Topic modelling, when it works well, can be very useful. It does have its limitations, however. It works best if each document covers one clear subject, and if the documents are a mix of material it may be more difficult to get meaningful topics from your corpus. It is also quite sensitive, in some ways, to things like OCR errors. Oftentimes one of the topics will be made up of keywords which are just OCR errors. Interpretation is a key part of topic modelling.

11.1 Methods

There are numerous ways to extract topics from documents. One which has been used extensively in digital humanities is Latent Dirichlet Allocation (LDA). The basic aim of the LDA algorithm is to figure out the mixture of words in each topic. It starts out by randomly assigning all the words into topics, and then, for each word in the data:

- It assumes that all other words are in the correct topics.
- It moves the word in question to each topic, and calculates the probability that each is the correct topic, based on the other words currently in that topic, and their relationship to the word and the other documents it appears within.
- It does this lots of times until it reaches an optimal, stable state.

LDA topic modelling has its limitations, for example it treats documents as 'bag of words', meaning their order is not taken into account. Often, the resulting topics (which are represented simply as a group of keywords) are not easy to interpret. Newer methods of topic modelling, such as contextual topic modelling (CTM) are available which take into account context and word order but these are not, as of now, implemented within any R packages.

11.2 Topic modelling with the library 'topicmodels'

This chapter uses three libraries: `tidyverse`, `tidytext`, and `topicmodels`. If you are missing any of these, you can install them with the following code:

```
install.packages('tidyverse')
install.packages('tidytext')
install.packages('topicmodels')
```

```
library(tidyverse)
library(tidytext)
library(topicmodels)
```

11.3 Load the news dataframe and relevant libraries

Topic modelling can be quite computationally-intensive. To speed things up, we'll just look at a single newspaper title for the year 1855.

Either construct your own corpus by following Chapter 8 and Chapter 9, or download and open the ready-made .zip file with all issues from 1855. Next, get these articles into the correct format. See Chapter 10 for an explanation of this code:

```
news_sample_dataframe = list.files("newspaper_text/",
                                   pattern = "csv",
                                   recursive = TRUE,
                                   full.names = TRUE)


all_files = lapply(news_sample_dataframe, data.table::fread)

names(all_files) = news_sample_dataframe

all_files_df = data.table::rbindlist(all_files, idcol = 'filename')


news_df = all_files_df %>%
  mutate(filename = basename(filename))

title_names_df = tibble(newspaper_id = c('0002090', '0002194', '0002244', '0002642', '0002

news_df = news_df %>% separate(filename,
                               into = c('newspaper_id', 'date'), sep = "_") %>% # separate
  mutate(date = str_remove(date, "\\.csv")) %>%
  select(newspaper_id, date, art, text) %>%
  mutate(date = ymd(date)) %>% # make a date column, and turn it into date format
  mutate(article_code = 1:n()) %>% # give every article a unique code
  select(article_code, everything()) %>%
  left_join(title_names_df, by = 'newspaper_id')
```

Make a new dataset of only one title: the *Swansea and Glamorgan Herald*, using its unique ID:

```
news_for_tm = news_df %>%
  filter(newspaper_id == '0003089')
```

Next, use `unnest_tokens` to tokenise the data. For topic modelling, it can be good to remove stop words, and words which don't occur very frequently. We'll also remove any words made up of numbers, as an additional cleaning step. This is done using `anti_join()` and `filter()` with a regular expression to match any number.

```
data("stop_words")

news_for_tm = news_for_tm %>%
  unnest_tokens(output = word, input = text) %>%
  anti_join(stop_words) %>%
  filter(!str_detect(word, "[0-9]"))
```

```
Joining with `by = join_by(word)`
```

11.4 Create a dataframe of word counts with tf_idf scores

The LDA algorithm expects a count of the words found in each document. We will generate the necessary statistics using the `tidytext` package, as used in the previous chapter. First, make a dataframe of the words in each document, with the count and the tf-idf score. This will be used to filter and weight the text data.

First, get the word counts for each article:

```
issue_words = news_for_tm %>%
  group_by(article_code, word) %>%
  tally() %>%
  arrange(desc(n))
```

Next, use `bind_tf_idf()` to get the tf_idf scores:

```
issue_words_tf_idf = issue_words %>%
  bind_tf_idf(word, article_code, n)
```

11.5 Make a 'document term matrix'

Using the function `cast_dtm()` from the topicmodels package, make a document term matrix. This is a matrix with all the documents on one axis, all the words on the other, and the number of times that word appears as the value. We'll also filter out words with a low tf-idf score, and only include words that occur at least 5 times.

```
dtm_long <- issue_words_tf_idf %>%
    filter(tf_idf > 0.00006) %>%
  filter(n>5) %>%
    cast_dtm(article_code, word, n)
```

Use the `LDA()` functionfrom the `topicmodels` package to compute the model. For this function we need to specify the number of topics in advance, using the argument `k`, and we'll set the random seed to a set number for reproducibility. It can take some time to run the model, depending on the size of the corpus.

```
lda_model_long_1 <- LDA(dtm_long,k =12, control = list(seed = 1234))
```

The object `lda_model_long_1` is a list containing, amongst other things, a list of the topics and the words which make them up, and a list of the documents with the mixture of topics within them. To view this object easily, we can use the `tidy` function from the `tidytext` package. The tidy function is a generic, meaning it can be used in different ways depending on the input. In this case, the tidytext package contains a method specifically created for turning the outputs of the LDA topic model into a readable format. We can choose to look either at the `beta` (the mixture of terms in topics) or `gamma` (the mixture of topics in documents).

```
beta_result <- tidytext::tidy(lda_model_long_1,matrix = 'beta')

gamma_result <- tidytext::tidy(lda_model_long_1,matrix = 'gamma')
```

What information do these contain? Well, let's look first at the mixture of keywords within a certain topic, using the 'beta' matrix:

```
beta_result %>% filter(topic ==1) %>%
  slice_max(order_by = beta, n = 10) %>%
  kableExtra::kbl()
```

topic	term	beta
1	street	0.1591182
1	amp	0.1189573
1	glasgow	0.0875793
1	esq	0.0351185
1	st	0.0316020
1	john	0.0266981
1	sale	0.0241553
1	buchanan	0.0209430
1	james	0.0198344
1	public	0.0127428

We can plot the top words which make up each of the topics, to get an idea of how the articles have been categorised as a whole. Some of these make sense: there's a topic which seems to be about university and education, one with words relating to poor laws, and a couple about disease in the army, as well as some more which contain words probably related to the Crimean war.

```
beta_result %>%
  group_by(topic) %>%
  slice_max(order_by = beta, n = 10) %>%
    ungroup()%>%
    mutate(topic = as.factor(topic),
           term = reorder_within(term, beta, topic)) %>%
    ggplot(aes(term, beta, fill = factor(topic))) +
    geom_col(show.legend = FALSE) +
    facet_wrap(~ topic, scales = "free")+
    scale_x_reordered() +
    coord_flip()
```

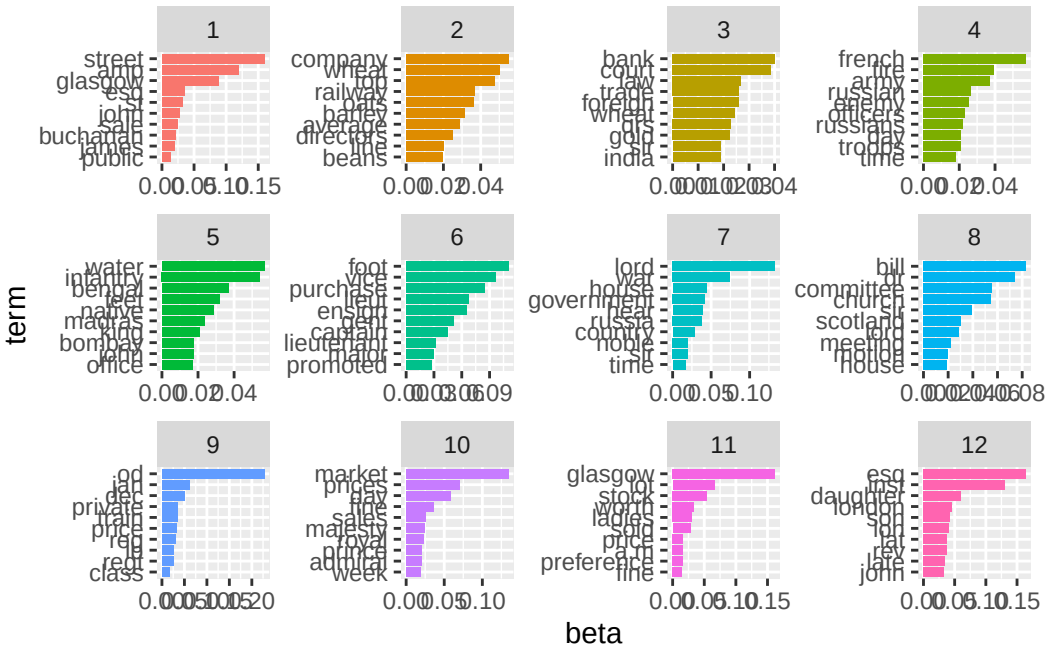


Figure 11.1: Sample Topics

The result seems to have picked some sensible topics: Each is a mixture of words, and the most important words seem to be sensibly connected to each other. It looks like there is a topic of government news, one on the Crimean war, one on markets and the economy/trade, and so forth.

We can also look at the other view, and look at how each article has been assigned to topics. Let's take a look at the document which has been signed the highest probability for topic 4, which seems to be about the Crimean War:

```
gamma_result <- tidytext::tidy(lda_model_long_1, 'gamma')

gamma_result %>% filter(topic == 4) %>% arrange(desc(gamma)) %>% head(10) %>%
  kableExtra::kbl()
```

document	topic	gamma
94868	4	0.9996461
90659	4	0.9995358
90839	4	0.9986704
89320	4	0.9984336
89587	4	0.9978388
89378	4	0.9978062
94037	4	0.9977378
94766	4	0.9977020
92196	4	0.9976650
95635	4	0.9976268

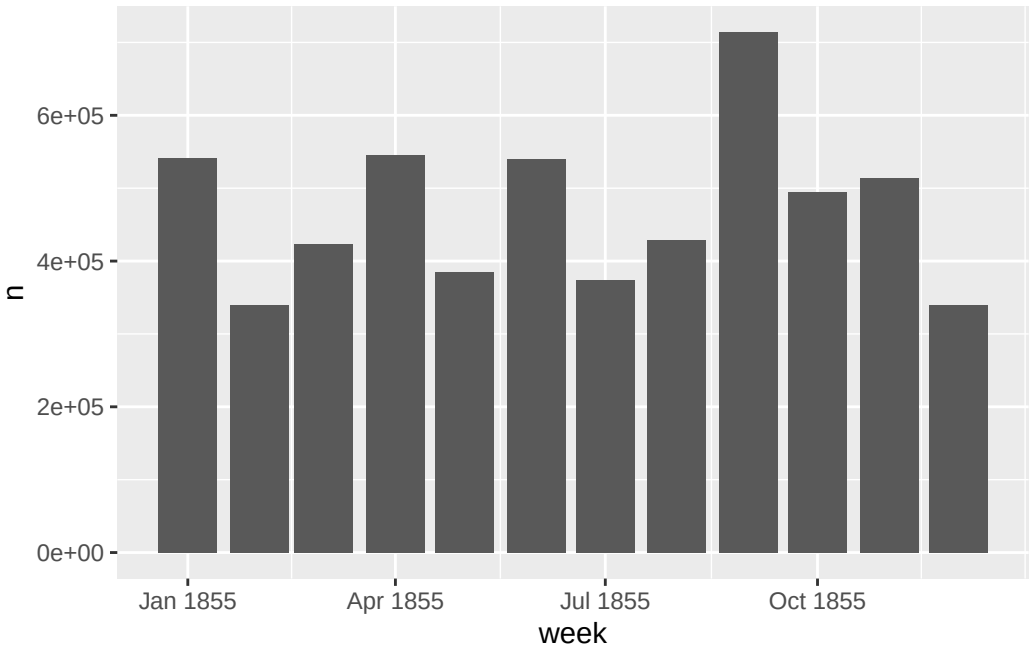
We can now read this document:

```
top_article = gamma_result %>% filter(topic == 4) %>% slice_max(order_by = gamma, n = 1) %

news_df %>% filter(article_code ==top_article ) %>%
  mutate(text = str_trunc(text, 5000)) %>%
  kableExtra::kbl()
```

article_code	newspaper_id	date	art	text
94868	0003089	1855-09-29	2	TILE ASSAULT ON SEBASTOPOL. STORMING AND C

Looking at this, it looks like it might be a good way of understanding and categorising the articles in the newspaper. Let's take another perspective. Take the documents which have as their highest-probability topic number 4, and see how they distribute over time. This might tell us something about how the war was reported over the year.

```
gamma_result %>% group_by(document) %>%
  slice_max(order_by = gamma, n = 1) %>%
  filter(topic == 4) %>%
  left_join(news_df %>%
              mutate(article_code = as.character(article_code)),
            by =c('document' = 'article_code'))  %>%
    mutate(week = ymd(cut(date, 'month'))) %>% ungroup() %>%
  mutate(article_word_count = str_count(text)) %>%
  count(week, wt = article_word_count) %>%
  ggplot() + geom_col(aes(x = week, y = n))
```



This gives us some sense of when reporting about the Crimean War may have peaked. September 1855 was an important time, because of the battle of Sevastepol. But it also looks like news about the war was fairly consistent across the year.

You can also group the articles by their percentage of each 'topic', and use this to find common thread between them - for more on this, see here:

11.6 Recommended Reading

Marjanen, Jani, Elaine Zosa, Simon Hengchen, Lidia Pivovarova, and Mikko Tolonen. "Topic Modelling Discourse Dynamics in Historical Newspapers," 2020. https://doi.org/10.48550/ARXIV.2011.10428.

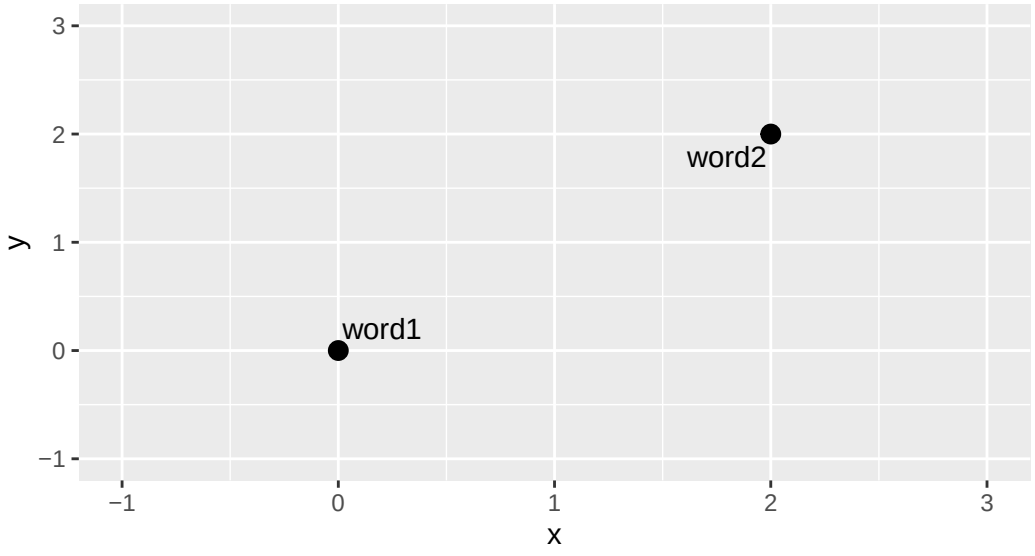
https://www.tidytextmining.com/topicmodeling.html

12 Word Embeddings

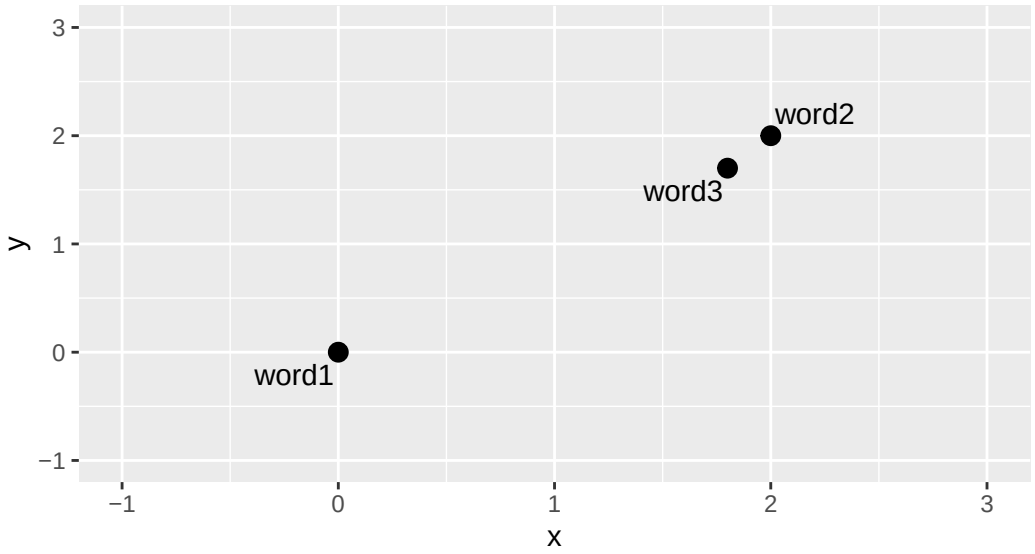
12.1 What are Word Embeddings?

The use of word embeddings is a valuable tool for understanding the meaning of words within texts. They can be used to understand semantic change within vocabularies - how the meaning of words changed over time. Word embeddings have been heavily used in the context of historical newspapers, for example to examine the changing biases around gender in Dutch newspapers (Wevers 2019), or to understand concepts Wevers and Koolen (2020). In a nutshell, word embeddings are a mathematical representation of words in a corpus, taking into account how that word is used and its relationship to other words. This representation can be used for further tasks, such as measuring semantic change, as a form of text search, or as the input for machine learning models, for classification for example.

The basic premise of word embeddings is to assign a mathematical value to each word in a corpus. An embedding is a ultimately a point in Euclidean space. For example, a word in two-dimensional Euclidean space might be represented by the point `c(2,2)`. A second word might be represented by the point `c(1,1)`. If we draw these on a graph it would look like this:



When represented on a graph like this, it's easy to do mathematical calculations such as measuring the distance between the words. We can also easily calculate whether a word is closer to one or another. If we introduce a third word to the above:



We can easily tell (and also calculate, using a simple distance formula) that word 3 is closer to word 2 than it is to word 1.

We can also represent the words in more than two dimensions. It becomes more difficult to draw them, but the mathematics stays the same.

Word embedding methods, then, essentially turn each word in a corpus into a vector - meaning a series of numbers representing its position in multi-dimensional space. In the example above, the words are represented by a vector of length two, such as `c(1,1)`.

There are lots of ways of determining these word embeddings, and there is no single 'true' representation. A popular method is to look at the context of words and use this information as as way to determine the most appropriate vector by which each word can be represented. A model can be trained which means a set of embeddings can be developed where words which appear often in context with each other will be closer together within the multi-dimensional space of that corpus.

Imagine the above word 1 is something like **cat**, word 2 is a colour, say **yellow**, and word 3 is another colour, say **red**. Red and yellow are much more likely to be used in the same context (**I have a red/yellow raincoat**) than the word cat. The sentence **I have a cat raincoat** is less likely to occur in a corpus, though not impossible… This means that red and yellow are likely to be semantically similar. A good contextual word embedding model will mean they will be most likely be placed closer together in multi-dimensional space.

12.2 Word Embedding Algorithms

There are a number of algorithms available for doing this. Two popular ones are word2vec, created in 2013 by researchers at Google, and the GloVe algorithm, created at Stanford in 2014. Both of these look at the context of words, and iterate over a process which tries to maximise in some way the relationship between the vectors and the context of the words, using neural networks.

GloVe takes into account the overall word co-occurrence of the words in the data, alongside the local co-occurrence, which can be more efficient. Jay Alammar has an excellent explainer of word2vec, much of which also applies to GloVe.

Essentially, these use neural networks to learn the best set of embeddings which are as good as possible at predicting the next word in the sentences found in the data - a form of unsupervised learning.

12.3 Creating Word Embeddings with R and text2vec.

In R, we can access these algorithms through a package called `text2vec`. Text2vec is a package which can do a number of NLP tasks, including word vectorisation. The package has a vignette which explains how to use the GloVe algorithm. I also recommend reading this tutorial by Michael Clark, which also uses text2vec.

On a practical level, the steps to use this package to generate embeddings are the following:

- Construct the input data from the full texts of the newspapers extracted in previous chapters. This involves tokenising the data, and creating a count of the appearances of words in the data.
- Next, create a term co-occurrence matrix. This is a large matrix which holds information on how often words occur together. For the gloVe algorithm, we pick a 'window'. The co-occurrence statistics count all instances of terms occurring together within this window, over the whole dataset.
- Run the gloVe algorithm on this co-occurrence matrix.
- Construct the vectors from the resulting output.

12.3.1 Load libraries

This tutorial uses libraries used previously, plus a new one called `text2vec`. If you haven't installed these (or some of them), you can do so with the following:

```
install.packages('text2vec')
install.packages('tidyverse')
install.packages('tidytext')
```

Once you're done, load the packages:

```
#| warning: false
#| message: false

library(text2vec)
library(tidyverse)
library(tidytext)
```

12.3.2 Load and create the newspaper dataset

Either construct your own corpus by following [Chapter -@sec-download] and [Chapter -@sec-extract], or download and open the ready-made .zip file with all issues from 1855. Next, get these articles into the correct format. See [Chapter -@sec-count] for an explanation of this code:

```
news_sample_dataframe = list.files(path = "newspaper_text/",
                                   pattern = "csv",
                                   recursive = TRUE,
                                   full.names = TRUE)


all_files = lapply(news_sample_dataframe, data.table::fread)

names(all_files) = news_sample_dataframe

all_files_df = data.table::rbindlist(all_files, idcol = 'filename')


title_names_df = tibble(newspaper_id = c('0002090', '0002194', '0002244', '0002642', '0002

news_df = all_files_df %>%
  mutate(filename = basename(filename))


news_df = news_df %>%
```

```
  separate(filename,
           into = c('newspaper_id', 'date'), sep = "_") %>% # separate the filename into t
  mutate(date = str_remove(date, "\\.csv")) %>% # remove .csv from the new data column
  select(newspaper_id, date, art, text) %>%
  mutate(date = ymd(date)) %>% # turn the date column into date format
  mutate(article_code = 1:n()) %>% # give every article a unique code
  select(article_code, everything()) %>% # select all columns but with the article code fi
  left_join(title_names_df, by = 'newspaper_id') # join the titles
```

12.3.3 Create the correct input data

The first step is to create the vocabulary which will be used to later construct the term co-occurrence matrix.

First, use `unnest_tokens()` (see the previous chapter) to get a list of the tokens within the data. Store the tokens column (called word) as a list:

```
news_tokens = news_df %>%
  unnest_tokens(output = word, input = text )

news_words_ls = list(news_tokens$word)
```

Next, create the 'iterator' using a function from the `text2vec` package, `itoken()`. An iterator is an object which will tell our function how to move through the list of words.

```
it = itoken(news_words_ls, progressbar = FALSE)
```

Create the vocabulary list by passing the iterator to the function `create_vocabulary()`. Furthermore, use `prune_vocabulary()` to remove very infrequent words, which won't contain much information and in many cases may be OCR artefacts. You can experiment with this value, depending on the size of your dataset.

```
news_vocab = create_vocabulary(it)

news_vocab = prune_vocabulary(news_vocab, term_count_min = 10)
```

Construct the term co-occurrence matrix. To begin with, create a vectorizer using the function `vocab_vectorizer()`. As with `iterator` above, this creates an object whose job is to describe how to do something - in this case, how to map words in the correct way for the term co-occurrence matrix.

Using this, use the function `create_tcm()` to create the term co-occurrence matrix, using the iterator and the vectorizer created above. Specify the `skip_grams_window` paramter, which defines how large a context to consider when calculating the co-occurrence. The result, if you look at it, is a large sparse matrix, containing each pair of co-occurring words, and the number of times they co-occur in the data.

```
vectorizer = vocab_vectorizer(news_vocab)

# use window of 10 for context words
news_tcm = create_tcm(it, vectorizer, skip_grams_window = 10)
```

12.3.4 Run the GloVe algorithm

Next, run the GloVe algorithm. This is done by creating an R6 class object, rather than simply running a function. This is an example of using object-orientated programming. The process is slightly different. First, create an empty object of the GlobalVectors class using `GlobalVectors$new()` . Next, run the neural network using `glove$fit_transform`, specifying how many iterations and threads (processors) it should use. This will take some time to run.

```
glove = GlobalVectors$new(rank = 50, x_max = 10)
wv_main = glove$fit_transform(news_tcm, n_iter = 10, convergence_tol = 0.01, n_threads = 5
```

```
INFO  [11:40:52.558] epoch 1, loss 0.1577
INFO  [11:42:37.783] epoch 2, loss 0.1245
INFO  [11:44:17.373] epoch 3, loss 0.1148
INFO  [11:46:00.080] epoch 4, loss 0.1098
INFO  [11:47:48.712] epoch 5, loss 0.1066
INFO  [11:49:30.689] epoch 6, loss 0.1044
INFO  [11:51:15.383] epoch 7, loss 0.1027
INFO  [11:52:55.266] epoch 8, loss 0.1014
INFO  [11:54:34.365] epoch 9, loss 0.1004
INFO  [11:56:13.541] epoch 10, loss 0.0995
INFO  [11:56:13.543] Success: early stopping. Improvement at iterartion 10 is less then conv
```

On the advice from the package vignette, the following takes the average of the main and context vectors, which usually produces higher-quality embeddings.

```
news_wv_context = glove$components
```

```
news_word_vectors = wv_main + t(news_wv_context)
```

With this, using the following we can extract the embeddings for a single word, and find its cosine similarity to all other words in the data, using the function `sim2`

```
king = news_word_vectors["king", , drop = FALSE]

cos_sim = sim2(x = news_word_vectors, y = king, method = "cosine", norm = "l2")


head(sort(cos_sim[,1], decreasing = TRUE), 10)
```

```
     king     queen  nicholas   emperor    prince   brother  napoleon      duke
1.0000000 0.7983400 0.7630099 0.7362687 0.7284119 0.7085910 0.7042042 0.6785979
  majesty  sardinia
0.6635410 0.6563405
```

12.3.5 Limitations of Word Embeddings

As a warning, word embeddings are going to be **very closely related to the specific context of the corpus you are using**. This can be a problem, but could also easily be exploited to find out interesting things about a particular corpus.

Newspapers, like all texts, have their own particular history and way of writing. This means in some cases we may get surprising or unexpected results. To show how this works in practice, we'll compare the most-similar embeddings for two European cities: Paris and Madrid.

We might assume that these two would be very similar and have similar 'interchangeable' words. They're both European capital cities. The words used in a similar context to a city are, generally, other cities. Newspapers tend to talk about cities in a very similar way, after all. In terms of the news, one city is to a certain extent interchangeable with any other.

To test this, let's extract the vectors for these two cities, using a similar method to above. First, Madrid:

```
madrid = news_word_vectors["madrid", , drop = FALSE]

cos_sim = sim2(x = news_word_vectors, y = madrid, method = "cosine", norm = "l2")


head(sort(cos_sim[,1], decreasing = TRUE), 20)
```

```
      madrid       hamburg       trieste         cadiz      portugal
   1.0000000     0.7960927     0.7777172     0.7668066     0.7097533
  petersburg         spain     marseilles        brings         genoa
   0.7001306     0.6992612     0.6908350     0.6907667     0.6874887
     lisbon     barcelona        states       advices    california
   0.6869859     0.6809349     0.6794201     0.6769651     0.6549632
constantinople     dates        berlin     gibraltar           ult
   0.6469534     0.6455977     0.6348662     0.6217109     0.6061422
```

As expected, the most-similar words are other cities, Trieste, Cadiz, and so forth. Semantically, one city is sort of interchangeable for any other. Now let's look at Paris:

```
paris = news_word_vectors["paris", , drop = FALSE]

cos_sim = sim2(x = news_word_vectors, y = paris, method = "cosine", norm = "l2")


head(sort(cos_sim[,1], decreasing = TRUE), 20)
```

```
    paris     vienna     berlin       news petersburg       says   moniteur
1.0000000  0.7180955  0.6790837  0.6702161  0.6694427  0.6624528  0.6564738
 received     states     london     france  announces despatches       rome
0.6509878  0.6439363  0.6434359  0.6382281  0.6289350  0.6243353  0.6223005
    visit    gazette    emperor   journals  yesterday         de
0.6155782  0.6148640  0.6147882  0.6101875  0.6084938  0.6045791
```

The result is a list which is much more mixed, semanttically. In this list there are a few cities (Vienna, Rome, London), but there are also words relating to the transmission of news (News, says, gazette, despatch, daily…).

This suggests that Paris is not just an interchangeable city from where news is reported, but has perhaps a more important role, as a key relay place from where news is being sent. News is not just reported from Paris, but it is a place where news is gathered from across Europe to be sent onwards across the Channel. This is reflected in the contextual word embeddings.

12.4 Case study - semantic shifts in the word 'liberal' over time in *The Sun* Newspaper

As a final case study demonstrating how word embeddings might be used, we'll look at how a particular concept shifts in a single newspaper over time. By looking at the most-similar

words to a target word, in this case 'liberal', we can capture changes in the dominant meaning of the word. In Chapter 10, we did a bigram analysis (Section 10.7), which pointed to a change in the way the word liberal was used between 1802 and 1870.

To do so, we'll use another dataset, this time, all issues of *The Sun* newspaper from two years: 1802 (the first full year in the repository data) and 1870 (the last full year). We'll follow the same workflow as above, except create two entirely different set of word vectors for each time period. We can then look at the most similar words in each and make some conclusions. We can also compare the similarity of two words in each time period.

You can do this with any selection of newspapers from any dates. Here, I have already provided the full-text files for the two years of *The Sun*. If you want to use other titles or years, follow the steps outlined in chapters x and y.

Particularly because we are looking at a single title, and, therefore, we might expect it to be more consistent in its editorial and writing practices, looking at shifts might tell us something about how the concept or word use of the word liberal was treated differently in the press over time.

At the same time, there may be many other hidden reasons for the change in the word. Perhaps it is used in a popular advertisement which ran at one time and not another? You should attempt to understand, for example through close reading or secondary sources, why the semantics of a word might look as they do. And of course, this may not reflect anything deeper about how the concept or ideology changed over time. But understanding how a word was represented in a title and how that changed might be a starting-point for further analysis.

As before, we will load in the files, create the token list, and the word embeddings using GloVe. The steps here are exactly as above, just repeated for each year in the data.

```
theSun = list.files(path = "../../../Downloads/TheSun_sample/",
                                    pattern = "csv",
                                    recursive = TRUE,
                                    full.names = TRUE)
```

```
theSunall_files =   lapply(theSun, data.table::fread)

names(theSunall_files) = theSun

theSunall_files_df = data.table::rbindlist(theSunall_files, idcol = 'filename')

theSunall_files_df = theSunall_files_df %>%
  mutate(filename = basename(filename))
```

```
theSunall_files_df = theSunall_files_df %>%
  separate(filename,
           into = c('newspaper_id', 'date'), sep = "_") %>% # separate the filename into t
  mutate(date = str_remove(date, "\\.csv")) %>% # remove .csv from the new data column
  select(newspaper_id, date, art, text) %>%
  mutate(date = ymd(date)) %>% # turn the date column into date format
  mutate(article_code = 1:n()) %>% # give every article a unique code
  select(article_code, everything()) %>%
  left_join(title_names_df, by = 'newspaper_id')# select all columns but with the article
```

```
theSunTokens = theSunall_files_df %>%
  unnest_tokens(word, text, token = 'words')
```

```
tokens_1802 = theSunTokens %>%
  mutate(year = year(date)) %>%
  filter(year == 1802)

words_ls_1802 = list(tokens_1802$word)

it_1802 = itoken(words_ls_1802, progressbar = FALSE)

vocab_1802 = create_vocabulary(it_1802)

vocab_1802 = prune_vocabulary(vocab_1802, term_count_min = 10)
```

```
tokens_1870 = theSunTokens %>% mutate(year = year(date)) %>%
  filter(year == 1870)

words_ls_1870 = list(tokens_1870$word)

it_1870 = itoken(words_ls_1870, progressbar = FALSE)

vocab_1870 = create_vocabulary(it_1870)

vocab_1870 = prune_vocabulary(vocab_1870, term_count_min = 10)
```

```
vectorizer_1802 = vocab_vectorizer(vocab_1802)

# use window of 10 for context words
```

```
tcm_1802 = create_tcm(it_1802, vectorizer_1802, skip_grams_window = 10)

vectorizer_1870 = vocab_vectorizer(vocab_1870)

# use window of 10 for context words
tcm_1870 = create_tcm(it_1870, vectorizer_1870, skip_grams_window = 10)
```

```
glove1802 = GlobalVectors$new(rank = 50, x_max = 10)

wv_main_1802 = glove1802$fit_transform(tcm_1802, n_iter = 10, convergence_tol = 0.01, n_th
```

```
INFO  [11:59:06.316] epoch 1, loss 0.1607
INFO  [11:59:19.276] epoch 2, loss 0.1130
INFO  [11:59:32.279] epoch 3, loss 0.1001
INFO  [11:59:45.254] epoch 4, loss 0.0926
INFO  [11:59:58.214] epoch 5, loss 0.0877
INFO  [12:00:11.265] epoch 6, loss 0.0842
INFO  [12:00:24.200] epoch 7, loss 0.0815
INFO  [12:00:37.161] epoch 8, loss 0.0794
INFO  [12:00:50.158] epoch 9, loss 0.0778
INFO  [12:01:03.130] epoch 10, loss 0.0764
```

```
glove1870 = GlobalVectors$new(rank = 50, x_max = 10)

wv_main_1870 = glove1870$fit_transform(tcm_1870, n_iter = 10, convergence_tol = 0.01, n_th
```

```
INFO  [12:01:29.387] epoch 1, loss 0.1977
INFO  [12:01:55.247] epoch 2, loss 0.1421
INFO  [12:02:22.607] epoch 3, loss 0.1267
INFO  [12:02:57.039] epoch 4, loss 0.1184
INFO  [12:03:27.585] epoch 5, loss 0.1130
INFO  [12:03:54.991] epoch 6, loss 0.1093
INFO  [12:04:21.714] epoch 7, loss 0.1065
INFO  [12:04:48.463] epoch 8, loss 0.1044
INFO  [12:05:15.509] epoch 9, loss 0.1026
INFO  [12:05:42.213] epoch 10, loss 0.1012
```

```
wv_context_1802 = glove1802$components

word_vectors_1802 = wv_main_1802 + t(wv_context_1802)

wv_context_1870 = glove1870$components

word_vectors_1870 = wv_main_1870 + t(wv_context_1870)
```

Finally, we can compare the two sets of embeddings for the word 'liberal'.

```
liberal_1802 = word_vectors_1802["liberal", , drop = FALSE]

cos_sim_liberal_1802 = sim2(x = word_vectors_1802, y = liberal_1802, method = "cosine", no


liberal_1870 = word_vectors_1870["liberal", , drop = FALSE]

cos_sim_liberal_1870 = sim2(x = word_vectors_1870, y = liberal_1870, method = "cosine", no


head(sort(cos_sim_liberal_1802[,1], decreasing = TRUE), 10)
```

```
  liberal    advice      your   tribute allowance  valuable   elegant     added
1.0000000 0.5919485 0.5631463 0.5567651 0.5546792 0.5301071 0.5272170 0.5214882
 grateful   greatly
0.5063169 0.5055405
```

```
head(sort(cos_sim_liberal_1870[,1], decreasing = TRUE), 10)
```

```
    liberal  conservative     candidate         party      election
  1.0000000     0.7603324     0.7526032     0.6420928     0.6296235
   catholic         roman      electors      liberals parliamentary
  0.6097168     0.6083171     0.6025364     0.6018060     0.5941979
```

12.5 Word similarity changes

Let's use the same embeddings for a slightly different perspective on semantic shift. Using the same methods, we can take any two embeddings and calculate a similarity score. Looking at

how this score changed over time can be informative in understanding how a word's meaning (or the relationship between two words) changed over time.

Do this by again calculating the vectors for liberal, plus the word conservative. Do this for both 1802 and 1870.

Use the same method to calculate similarity, but this time instead of calculate one against all, we calculate one against the other. This returns a single similarity score.

```
liberal_1802 = word_vectors_1802["liberal", , drop = FALSE]
conservative_1802 = word_vectors_1802["conservative", , drop = FALSE]

sim2(x = liberal_1802, y = conservative_1802, method = "cosine", norm = "l2")
```

```
        conservative
liberal   -0.1930361
```

```
liberal_1870 = word_vectors_1870["liberal", , drop = FALSE]
conservative_1870 = word_vectors_1870["conservative", , drop = FALSE]

sim2(x = liberal_1870, y = conservative_1870, method = "cosine", norm = "l2")
```

```
        conservative
liberal    0.7603324
```

The words go from being very dissimilar (rarely used in the same context) to being very similar - both are used in terms of party politics by 1870.

A further, easy step, would be to create such a set of embeddings for sets of five years, and look at the change from one period to the next. This would help us to understand exactly when this shift occurred. A similar method has been used by researchers

As with the ngram analysis in a previous chapter, this points to a huge change in the semantic shift of the word liberal, in the newspapers.

Mention newer methods such as BERT which also produce word embeddings. Also mention this new LwM paper: https://muse.jhu.edu/pub/1/article/903976

BERT uses a technology called a transformer, which itself uses what is known as 'attention mechanism'. Rather than simply look at co-occurring words in a given sentence, transformers allow each word to share information with each other, meaning that important and related words within a sentence or chunk of text can be found. This means that the resulting embedding can take into account the specific context and even word order of a given phrase.

Furthermore, BERT generates an embedding not at the corpus-level but at the word-level, meaning that each individual utterance of a word can be represented differently, according to the way it is used in a particular sentence. This is powerful tool which really tease apart the way words, particularly those which might have multiple meanings, semantically shift over time.

As an alternative - try doing the same analysis, but first remove all the articles labelled as advertisements from the previous tutorial.

Also, download all years of the Sun. Divide into 5-year chunks, compare the vector between them.

13 Machine Learning with Tidymodels

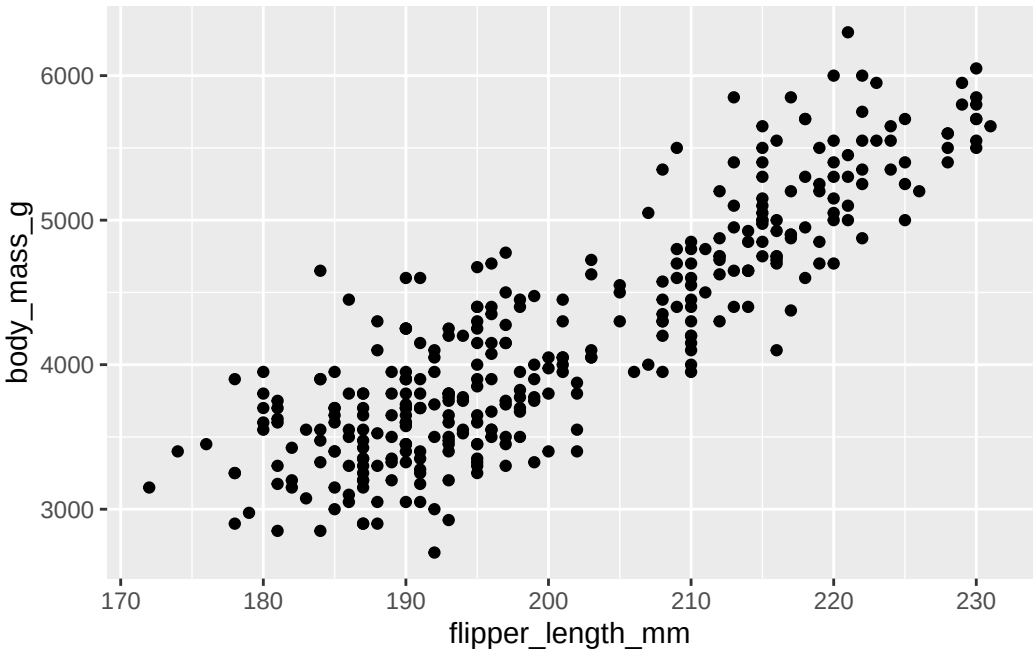
This chapter looks at machine learning using historical newspapers and the 'tidymodels' framework. We'll build a classifier which learns, from some pre-labelled examples, to differentiate between advertisements and news articles. We'll evaluate the model, and fine-tune its parameters. This chapter is heavily indebted to existing tidymodels tutorials on the web, such as the one from Supervised Machine Learning in R by Emil Hvitfeldt and Julia Silge.

13.1 Machine Learning

Machine learning is the name for a group of techniques which take input data of some kind, and learn how to achieve some particular goal. The 'deep learning' used by neural networks and in particular things like ChatGPT are one type, but the field has been around for much longer than that.

Machine learning itself can be divided into subsets: Machine learning done with neural networks, and what we might call 'classical' machine learning, which use algorithms.

A very simple form of machine learning is *linear regression*. Linear regression attempts to find the best fitting line through a dataset. Take this dataset of the flipper size and body mass of a group of observed penguins (from the R package `palmerpenguins`:

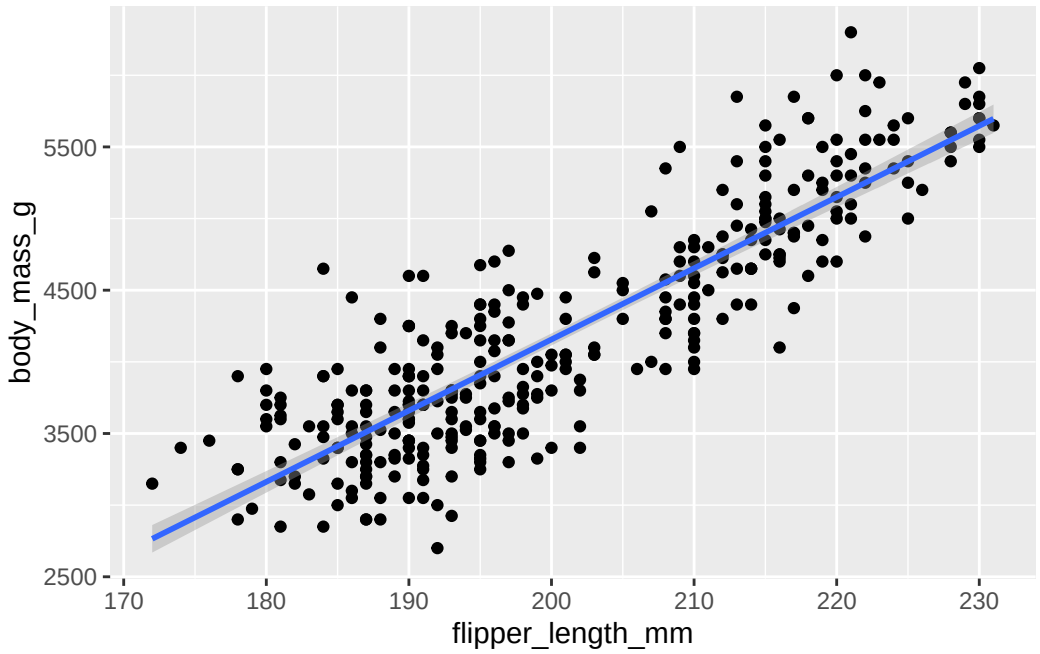


By visually inspecting this, we can guess that there is a statistical relationship between the length and mass: as one value gets higher, the other does too. We can use `geom_smooth()` and a linear model to predict the best line of fit through this dataset:

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

```
Warning: Removed 2 rows containing non-finite values (`stat_smooth()`).
```

```
Warning: Removed 2 rows containing missing values (`geom_point()`).
```



If we measured the slope of this line (using simple geometry), its angle would tell us about the relationship between the two values.In this way, our model helps us to understand some underlying pattern in the dataset - that these two values are highly correlated. We could also use it to *predict* new values: if given the body mass for a new penguin, by using this line, we could predict its most likely flipper length, and in most cases be fairly accurate.

This is a form of simple machine learning. We give an algorithm (here, linear regression) a bunch of input data, in this case the body mass and bill flipper length for a group of penguins, and it provides a model (in this case, a line with a certain slope), which hopefully helps us to explain the existing data and predict unknown parts. Models like this try in some way to minimise a loss function. In this case, finding the line where the error for each point (how far away it is from the line) is as small as possible.

The machine learning here will use basically the same principle. We'll give it data (text, in the form of a mathematical representation of the text, and its assigned category), and an algorithm (in this case, a 'Random Forest' model), and use that to predict the category of unseen texts. The model itself may also tell us something about the existing data.

Much of the AI work at the cutting edge of text analysis today uses neural networks, and in particular the 'transformer' mechanism which allows neural networks to better understand context and word order. But this kind of machine learning still has a very important role to play, it's well-established, relatively easy to run, and the results can be quite good depending on the problem. It is also a good way to get started with understanding the Tidymodels framework.

13.2 The Tidymodels Package

Tidymodels is an R 'meta-package' which allows you to interact with lots of different machine learning packages and engines in a consistent way. Using tidymodels, we can easily compare multiple models together and we can swap in one for another without having to re-do code. We also can make sure that our pre-processing steps are precisely consistent across any number of different models.

Machine learning can be used for a number of different tasks. One key one is text classification.

In this tutorial, we will use tidymodels to classify text into articles and advertisements. It could easily be generalised to any number of categories, for example foreign news, court reporting, and so forth. You can provide your own spreadsheet of training data (as we'll use below), and as long as it is in the same format and has similar information, you should be able to build your own classifier.

Once we have built the model, we will fine-tune it. The random forest algorithm we'll use, like most machine learning algorithms, has a group of parameters which can be adjusted. To find the best values for these, we'll evaluate the same data using many different combinations of parameters, and pick the best one. Using tidymodels, this can all be done consistently. We could even swap out any other model type, and otherwise reuse the exact same workflow.

Finally, we'll put the model to some use: we'll use the model to predict the class of the rest of the articles in the newspaper dataset, and do some analysis on this.

The model is only as good as the training data, and in this case, we don't have many examples, and it won't be terribly accurate in many cases. But it will show how a machine learning model can be operationalised for this task.

13.2.1 Basic Steps

In this chapter we'll create a model which can label newspaper as either articles or advertisements. We'll do some further steps to explore improving the model, and also to look at the most important 'features' used by the model to do its predicting. The steps are:

1. Download a labelled dataset, containing examples of our two classes (articles and advertisements).
2. Create a 'recipe' which is a series of pre-processing steps. This same recipe can be reused in different contexts and for different models.
3. Split the labelled data into testing and training sets. The training data is used to fit the best model. The test set is used at the end, to see how well it performs on unseen data.
4. Run an initial classifier and evaluate the results

5. 'Tune' the model, by re–running the classifier with different parameters, selecting the best one.
6. Run the 'best' model over the full news dataset, predicting whether or not an articles is news or an advertisement.

13.3 Install/load the packages

As a first step, you'll need to load the necessary packages for this tutorial. If you haven't already done so, you can install them first using the below.

If you have installed them, make sure you update to the latest version, as they can change rapidly.

```
install.packages('tidyverse')
install.packages('tidymodels')
install.packages('textrecipes')
install.packages('tidytext')
install.packages('ranger')
install.packages('vip')
```

```
library(tidyverse)
library(tidymodels)
library(textrecipes)
library(tidytext)
library(ranger)
library(vip)
```

13.4 Import data

As a first step, load some pre-labelled data. This contains a number of newspaper articles, and their 'type': whether it is an advertisement or an article.

If we take a look at the dataframe once it is loaded, you'll see it's quite simple structure: it's got a 'filename' column, the full text of the article store in 'text', and the type stored in a column called 'type'.

```
advertisements_labelled = read_csv('advertisements_labelled.csv')

advertisements_labelled = advertisements_labelled %>% filter(!is.na(text))
```

```
advertisements_labelled %>% head(5) %>% kableExtra::kbl()
```

filename	art	text
newspapers//0002194/1855/0328/0002194_18550328.csv	107	4 ,;,-A4J - BriTtr EMIT MO NT BLANC a
newspapers//0002194/1855/1228/0002194_18551228.csv	126	THE ROTAL PICTURES AT OSBORNE.
newspapers//0002194/1855/0529/0002194_18550529.csv	67	WOODIN'S ENTERTAINMENT. Mr. Woo
newspapers//0002194/1855/1023/0002194_18551023.csv	106	No. 19,675.] PUBLICATIONS. NOTICE.—
newspapers//0002194/1855/0501/0002194_18550501.csv	181	THE,A.I RE ituY AL ILAYMA.BahT. It of

13.5

13.6 Set up the Machine Learning Model

In this step, we begin preparing the data for machine learning. We set a seed for reproducibility to ensure consistent results when randomization is involved. The target variable 'type' in the advertisements data is converted to a factor as it represents categorical classes ('advertisement' and 'article').

The data is then split into training and testing sets using the `initial_split` function. This separation is crucial for evaluating the performance of the machine learning model and preventing overfitting.

```
set.seed(9999)
advertisements_labelled = advertisements_labelled %>%
  mutate(type = factor(type))

advertisements_split <- initial_split(advertisements_labelled, strata = type)

advertisements_train <- training(advertisements_split)
advertisements_test <- testing(advertisements_split)
```

13.7 Create Recipe for Text Data

To prepare the text data for modeling, we create a recipe using the `recipe` function. The `textrecipes` package provides essential tools for text preprocessing and feature extraction in machine learning. In this recipe, we tokenize the text, remove stop words, and apply a

term frequency transformation to represent the text data as numerical features. These transformations convert the raw text data into a format suitable for machine learning algorithms, enabling them to process and understand textual information.

```
advertisement_rec <-
  recipe(type ~ text, data = advertisements_train)

advertisement_rec <- advertisement_rec %>%
  step_tokenize(text, token = "words") %>%
  step_tokenfilter(text, max_tokens = 1000, min_times = 5 )  %>%
  step_tf(text)
```

Next, we set up a `workflow()` object, which will store the recipe and later the model instructions, and make it easier to reuse.

```
advertisement_wf <- workflow() %>%
  add_recipe(advertisement_rec)
```

Create the model. In this case, we'll use a random forest model, from the package `ranger`. Setting the `importance = "impurity"` parameter means we'll be able to see what words the model used to make its decisions.

```
rf_spec <- rand_forest("classification") %>%
  set_engine("ranger", importance = "impurity")

rf_spec
```

```
Random Forest Model Specification (classification)

Engine-Specific Arguments:
  importance = impurity

Computational engine: ranger
```

Take the workflow object we made above, and add the model to it. After this, use `fit` to run the model, specifying it should use the `advertisements_train` dataset we created above.

Finally, we use `predict` on this fitted model, specifying `advertisements_test` as the dataset. Add the true labels from `advertisements_test` as a new column, and use `accuracy()` to compare the true label with the prediction to get an accuracy score.

```
 advertisement_wf %>%
  add_model(rf_spec) %>%
  fit(data = advertisements_train)%>%
  predict(new_data = advertisements_test) %>%
  mutate(truth = advertisements_test$type) %>%
  accuracy(truth, .pred_class)
```

```
# A tibble: 1 x 3
  .metric  .estimator .estimate
  <chr>    <chr>          <dbl>
1 accuracy binary         0.811
```

13.8 Perform Cross-Validation

To evaluate the model further, we'll use cross-validation. Cross-validation is a crucial step in model evaluation. It helps assess the model's generalization performance on unseen data and reduces the risk of overfitting.

In this step, we set up cross-validation folds using the `vfold_cv` function, which creates multiple training and testing sets from the training data. The model will be trained and evaluated on each fold separately, providing a more robust estimate of its performance.

```
set.seed(234)
advertisements_folds <- vfold_cv(advertisements_train)
```

13.9 Train the Random Forest Model with Cross-Validation

Now, we train the Random Forest model using cross-validation. The `fit_resamples` function fits the model to each fold created during cross-validation, allowing us to evaluate its performance across different subsets of the training data. The `control_resamples` function is used to control various settings during the resampling process.

```
rf_wf <- workflow() %>%
  add_recipe(advertisement_rec) %>%
  add_model(rf_spec)

rf_wf
```

```
== Workflow ====================================================================
Preprocessor: Recipe
Model: rand_forest()

-- Preprocessor ----------------------------------------------------------------
3 Recipe Steps

* step_tokenize()
* step_tokenfilter()
* step_tf()

-- Model -----------------------------------------------------------------------
Random Forest Model Specification (classification)

Engine-Specific Arguments:
  importance = impurity

Computational engine: ranger
```

```
rf_rs <- fit_resamples(
  rf_wf,
  advertisements_folds,
  control = control_resamples(save_pred = TRUE)
)
```

13.10 Evaluate the Model's Performance

In this step, we collect the evaluation metrics and predictions from the cross-validation process. The collected metrics will help us assess the model's performance, while the predictions on each fold will be used for further analysis and comparison. By evaluating the model on multiple subsets of the data, we can gain insights into its robustness and reliability.

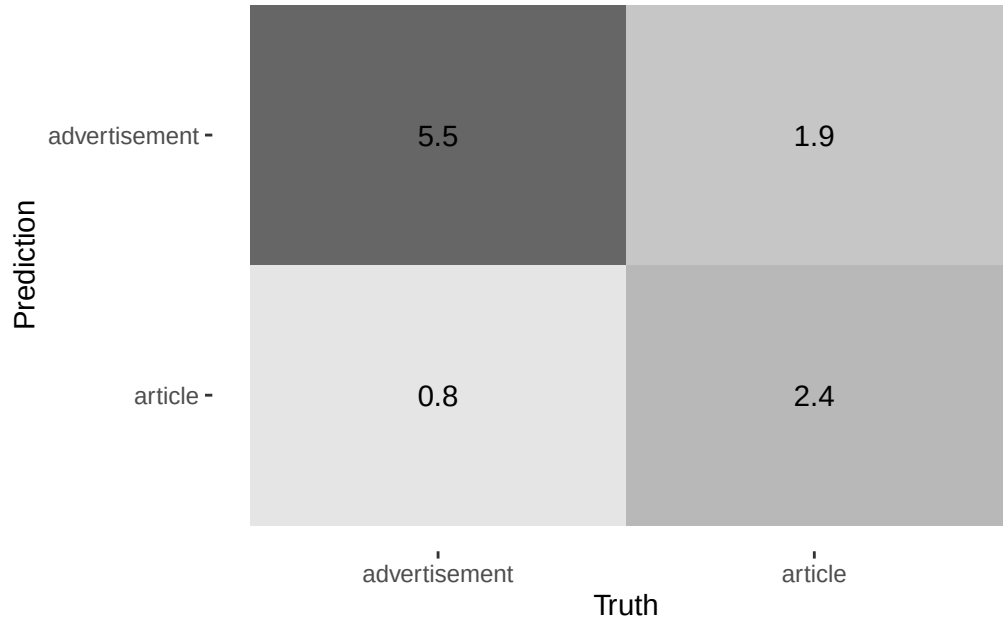
```
rf_rs_metrics <- collect_metrics(rf_rs)
rf_rs_predictions <- collect_predictions(rf_rs)
```

13.11 Visualize the Confusion Matrix

The confusion matrix is a useful visualization for evaluating the performance of a classification model. It shows the number of true positives, true negatives, false positives, and false negatives.

The `autoplot` function from the `yardstick` package allows us to visualize the confusion matrix as a heatmap. This visualization aids in understanding the model's classification accuracy and any potential misclassifications.

```
conf_mat_resampled(rf_rs, tidy = FALSE) %>%
  autoplot(type = "heatmap")
```



We can see that there are very few true advertisements which are misclassified as articles - but there are some some articles misclassified as advertisements.

13.12 Tune the Random Forest Model

In machine learning, hyperparameter tuning is essential for optimizing model performance. In this step, we define a tuning grid using the `rand_forest` function. The grid specifies different combinations of hyperparameters, such as the number of variables randomly sampled for splitting (`mtry`) and the minimum number of samples per leaf node (`min_n`). We aim to find the best combination of hyperparameters that yields the highest performance.

```
tune_spec <- rand_forest(
  mtry = tune(),
  trees = 1000,
  min_n = tune()
```

```
) %>%
  set_mode("classification") %>%
  set_engine("ranger", importance = "impurity")
```

```
tune_wf <- workflow() %>%
  add_recipe(advertisement_rec) %>%
  add_model(tune_spec)
```

13.13 Tune the Random Forest Model with Cross-Validation

Now, we perform hyperparameter tuning using cross-validation. The `tune_grid` function uses the tuning grid specified earlier and fits the model on each fold of the data to identify the optimal hyperparameters. This process helps us identify the best hyperparameters for the Random Forest model, leading to improved performance and better generalization.

```
set.seed(555)
trees_folds = vfold_cv(advertisements_train)
```

```
doParallel::registerDoParallel()

set.seed(666)
tune_res = tune_grid(
  tune_wf,
  resamples = trees_folds,
  grid = 20
)
```

```
i Creating pre-processing data to finalize unknown parameter: mtry
```

```
tune_res
```

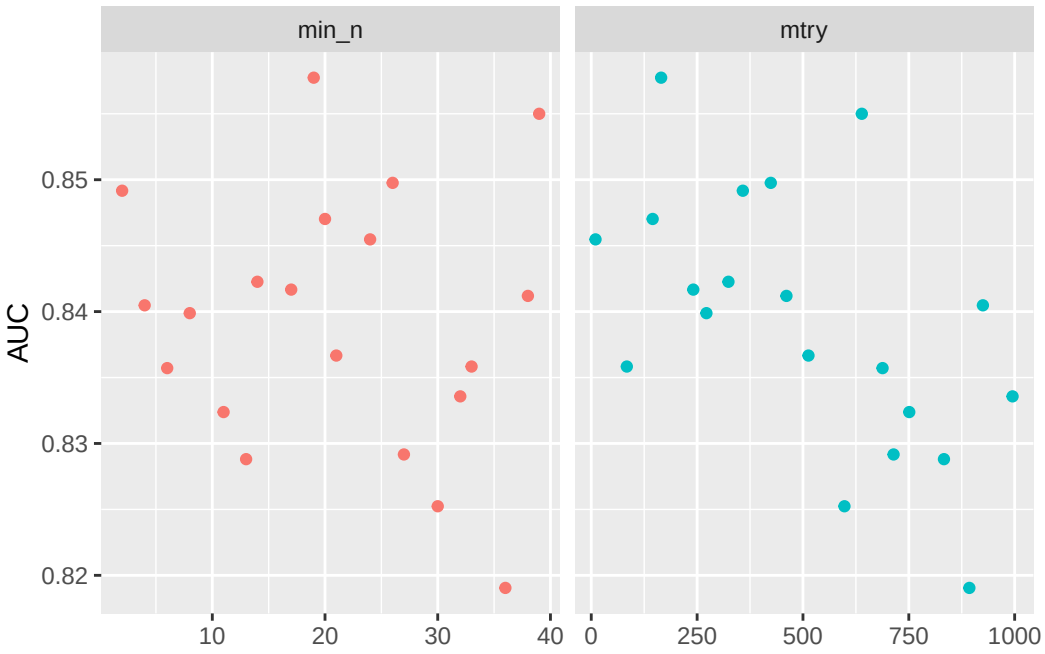
```
# Tuning results
# 10-fold cross-validation
# A tibble: 10 x 4
   splits          id     .metrics          .notes
   <list>          <chr>  <list>            <list>
 1 <split [95/11]> Fold01 <tibble [40 x 6]> <tibble [0 x 3]>
```

```
 2 <split [95/11]> Fold02 <tibble [40 x 6]> <tibble [0 x 3]>
 3 <split [95/11]> Fold03 <tibble [40 x 6]> <tibble [0 x 3]>
 4 <split [95/11]> Fold04 <tibble [40 x 6]> <tibble [0 x 3]>
 5 <split [95/11]> Fold05 <tibble [40 x 6]> <tibble [0 x 3]>
 6 <split [95/11]> Fold06 <tibble [40 x 6]> <tibble [0 x 3]>
 7 <split [96/10]> Fold07 <tibble [40 x 6]> <tibble [0 x 3]>
 8 <split [96/10]> Fold08 <tibble [40 x 6]> <tibble [0 x 3]>
 9 <split [96/10]> Fold09 <tibble [40 x 6]> <tibble [0 x 3]>
10 <split [96/10]> Fold10 <tibble [40 x 6]> <tibble [0 x 3]>
```

13.14 Visualize the Tuning Results

To visualize the tuning results, we plot the average Area Under the Receiver Operating Characteristic Curve (ROC AUC) against different values of `mtry` and `min_n`. ROC AUC is a common metric for assessing the model's ability to discriminate between classes. The plot provides insights into how different hyperparameter values affect the model's performance.

```
tune_res %>%
  collect_metrics() %>%
  filter(.metric == "roc_auc") %>%
  select(mean, min_n, mtry) %>%
  pivot_longer(min_n:mtry,
    values_to = "value",
    names_to = "parameter"
  ) %>%
  ggplot(aes(value, mean, color = parameter)) +
  geom_point(show.legend = FALSE) +
  facet_wrap(~parameter, scales = "free_x") +
  labs(x = NULL, y = "AUC")
```



It's a little difficult to interpret, but it looks like the highest values are between about 40 and 250 for the `mtry` value, and around 10 and 30 for the `min_n` value. We can do another grid search, this time just looking between these values.

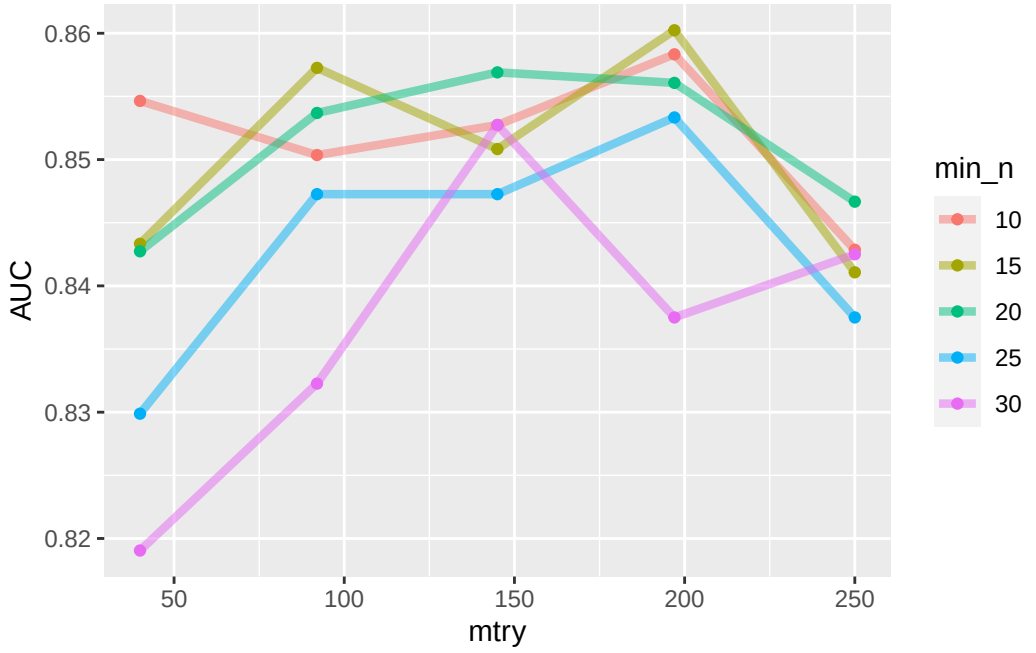
```
rf_grid <- grid_regular(
  mtry(range = c(40, 250)),
  min_n(range = c(10,30)),
  levels = 5
)
```

```
set.seed(999)
regular_res <- tune_grid(
  tune_wf,
  resamples = trees_folds,
  grid = rf_grid
)
```

```
regular_res %>%
  collect_metrics() %>%
  filter(.metric == "roc_auc") %>%
  mutate(min_n = factor(min_n)) %>%
  ggplot(aes(mtry, mean, color = min_n)) +
```

```
  geom_line(alpha = 0.5, size = 1.5) +
  geom_point() +
  labs(y = "AUC")
```

```
Warning: Using `size` aesthetic for lines was deprecated in ggplot2 3.4.0.
i Please use `linewidth` instead.
```




```
best_auc <- select_best(regular_res, "roc_auc")

final_rf <- finalize_model(
  tune_spec,
  best_auc
)
```

Lastly, evaluate the accuracy using the same method as before:

```
advertisement_wf %>%
  add_model(final_rf) %>%
  fit(data = advertisements_train)%>%
  predict(new_data = advertisements_test) %>%
  mutate(truth = advertisements_test$type) %>%
```

```
    accuracy(truth, .pred_class)
```

```
# A tibble: 1 x 3
  .metric  .estimator .estimate
  <chr>    <chr>          <dbl>
1 accuracy binary         0.865
```

As you can see, we have improved the model's performance, from about .80 to about .86. There are other ways you could try to improve the performance further. With text, the pre-processing steps can often make a huge difference. You could experiment with the text 'recipe', for instance adjusting `step_tokenfilter` to include or remove more tokens. You could also use `step_word_embeddings`, using the output of the word embeddings in Chapter 12.

13.15 Build the Final Random Forest Model

In this step, we build the final Random Forest model using the best hyperparameters obtained from tuning. The model is then fitted on the entire training dataset to capture the relationships between features and target classes optimally. This final model is the one that we will use for making predictions on new data.

```
final_wf <- workflow() %>%
  add_recipe(advertisement_rec) %>%
  add_model(final_rf)

final_res <- final_wf %>%
  last_fit(advertisements_split)

final_res %>%
  collect_metrics()
```

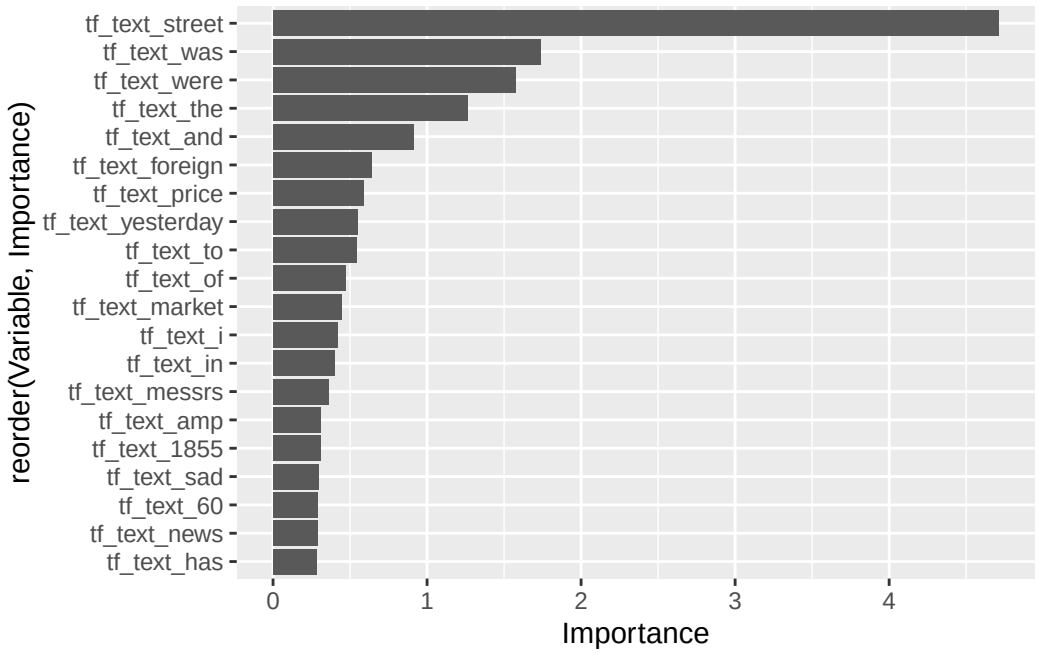
```
# A tibble: 2 x 4
  .metric  .estimator .estimate .config
  <chr>    <chr>          <dbl> <chr>
1 accuracy binary         0.865 Preprocessor1_Model1
2 roc_auc  binary         0.888 Preprocessor1_Model1
```

13.16 What features is the model using?

To understand how the model is using the text to make decisions, we can look at the most important features (in this case, text frequency counts) used by the random forest algorithm. We'll use the package `vip` to extract the most important features, using the function `extract_fit_parsnip()`, and plotting it using `ggplot2`.

For the random forest method, we can only see the overall top features, and not which were more important for the prediction of the different categories. The most important feature is the frequency of the word 'street': at a guess, this is used more often in advertisements, which very often contain an address to a business or service. Interestingly, the second most important feature is the word 'was'. Is this perhaps because news articles are more likely to use the past tense than advertisements? Interestingly, the frequency of certain function words such as `the`, `and`, `to`, `of` (which would often be filtered out as 'stop words') also seems to be important to the model. One explanation is that the use of these words is quite different in prose and in the kind of text used in advertisements.

```
complaints_imp <- extract_fit_parsnip(final_res$.workflow[[1]]) %>%
  vi(lambda = choose_acc$penalty)

complaints_imp %>%
  mutate(Variable = str_remove(Variable, "tfidf_text_")) %>%
  head(20) %>%
  ggplot() +
  geom_col(aes(x = reorder(Variable,Importance), y= Importance)) + coord_flip()
```



13.17 Using the Model

Once we are happy with the final model, we can use it to label unseen data as either articles or advertisements. For this, we'll need a newspaper corpus. As in previous chapters, either construct your own corpus by following Chapter 8 and Chapter 9, or download and open the ready-made .zip file with all issues from 1855. Next, get these articles into the correct format. See Chapter 10 for an explanation of this code:

```
news_sample_dataframe = list.files(path = "newspaper_text/",
                                   pattern = "csv",
                                   recursive = TRUE,
                                   full.names = TRUE)


all_files = lapply(news_sample_dataframe, data.table::fread)

names(all_files) = news_sample_dataframe

all_files_df = data.table::rbindlist(all_files, idcol = 'filename')
```

```
title_names_df = tibble(newspaper_id = c('0002090', '0002194', '0002244', '0002642', '0002

news_df = all_files_df %>%
  mutate(filename = basename(filename))


news_df = news_df %>%
  separate(filename,
           into = c('newspaper_id', 'date'), sep = "_") %>% # separate the filename into t
  mutate(date = str_remove(date, "\\.csv")) %>% # remove .csv from the new data column
  select(newspaper_id, date, art, text) %>%
  mutate(date = ymd(date)) %>% # turn the date column into date format
  mutate(article_code = 1:n()) %>% # give every article a unique code
  select(article_code, everything()) %>% # select all columns but with the article code fi
  left_join(title_names_df, by = 'newspaper_id') # join the titles
```

13.18 Make Predictions on Newspaper Articles

With the final Random Forest model trained, we proceed to make predictions on the newspaper articles' text data. This involves applying the text preprocessing steps (e.g., tokenization, TF-IDF) used during training to transform the new data into the same format. The model then predicts whether each article is an advertisement or not.

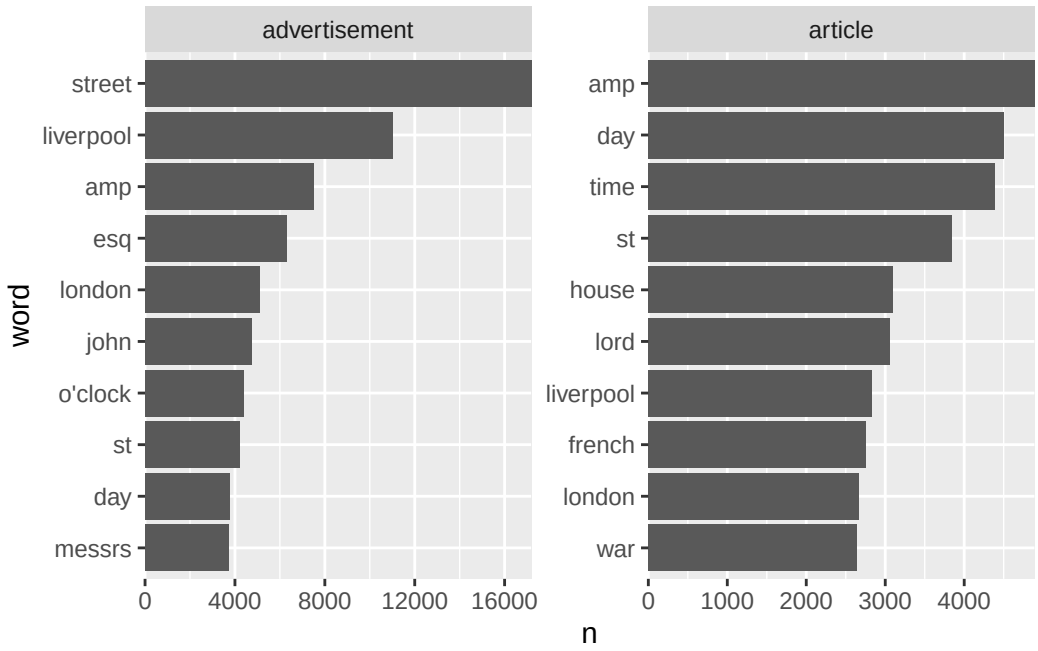
```
new_ads_to_check = final_wf %>%
  fit(data = advertisements_train)%>%
  predict(new_data = news_df)

all_files_df = all_files_df %>%
  mutate(prediction =new_ads_to_check$.pred_class)
```

13.19 Analyze the Top Words in Advertisements

After making predictions, we perform text analysis to identify the top words associated with advertisements. We tokenize the text data, count the occurrences of each word for each prediction, and select the most frequent words in advertisements. This analysis allows us to gain insights into the language patterns characteristic of advertisements.

```
data('stop_words')


all_files_df %>%
  head(10000) %>%
  unnest_tokens(word, text) %>%
  filter(!str_detect(word, "[0-9]")) %>%
  anti_join(stop_words) %>%
  count(prediction, word) %>%
  group_by(prediction) %>%
  slice_max(order_by = n, n = 10) %>% ungroup() %>%
    mutate(prediction = as.factor(prediction),
           word = reorder_within(word, n, prediction)) %>%
  ggplot() + geom_col(aes(word, n)) +
  facet_wrap(~prediction, scales = 'free') +
    scale_x_reordered()+
    scale_y_continuous(expand = c(0,0))+ coord_flip()
```

```
Joining with `by = join_by(word)`
```



Advertisements often contain an address, including the word 'street', unlike articles. This matches the findings from the top features, above. Advertisements also often specify an exact

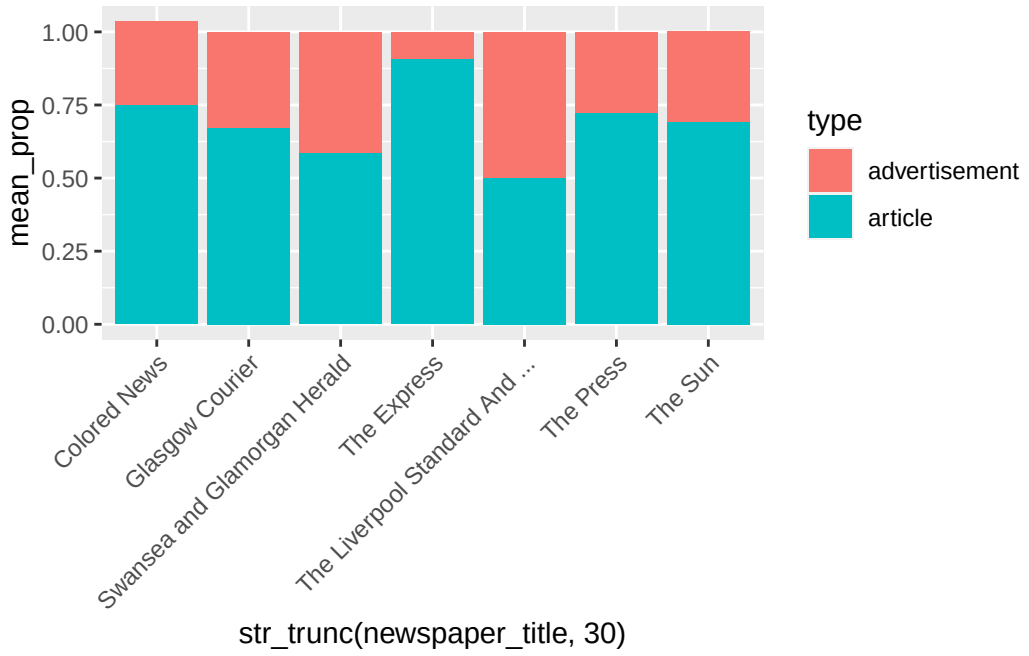
time and date, resulting in higher counts of the word 'o'clock'.

13.20 Analyze the Proportion of Advertisements in Each Newspaper Issue

Finally, we analyze the proportion of advertisements in each newspaper issue. We group the articles by their `newspaper_id` and `date`, calculate the proportion of advertisements in each issue, and then compute the mean proportion for each newspaper. This analysis helps us understand the prevalence of advertisements in different newspapers over time.

```
news_df$type = new_ads_to_check$.pred_class

news_df =news_df  %>% mutate(count = str_count(text))

news_df =news_df  %>% mutate(issue_code = paste0(newspaper_id, "_", date))

news_df %>%
  group_by(newspaper_title, issue_code, type) %>%
  summarise(n = sum(count)) %>%
  mutate(prop = n/sum(n)) %>%
  group_by(newspaper_title, type) %>%
  summarise(mean_prop = mean(prop)) %>%
  ggplot() +
  geom_col(aes(x = str_trunc(newspaper_title,30), y = mean_prop, fill = type)) +
  theme(axis.text.x = element_text(angle = 45, hjust = 1, vjust = 1))
```

```
`summarise()` has grouped output by 'newspaper_title', 'issue_code'. You can
override using the `.groups` argument.
`summarise()` has grouped output by 'newspaper_title'. You can override using
the `.groups` argument.
```



Interestingly, some of the regional titles (Liverpool, Glasgow, Swansea and Glamorgan) seem to have much higher proportions of advertisements - provided the labelling by the machine learning model is largely correct, of course.

13.21 Recommended Reading

Supervised Machine Learning for Text Analysis in R

Broersma, Marcel, and Frank Harbers. "Exploring Machine Learning to Study the Long-Term Transformation of News." In *Journalism History and Digital Archives*, edited by Henrik Bødker, 1st ed., 38–52. Routledge, 2020. https://doi.org/10.4324/9781003098843-4.

14 Text Reuse

Nineteenth-century newspapers shared text all the time. Sometimes this took the form of credited reports from other titles. For much of the century, newspapers paid the post office to give them a copy of all other titles. Official reused dispatches were not the only way text was reused: advertisements, of course, were placed in multiple titles at the same time, and editors were happy to use snippets, jokes, and so forth

Detecting the extent of this reuse is a great use of digital tools. R has a library, *textreuse*, which allows you to do this reasonably simply. It was intended to be used for plagiarism detection and to find duplicated documents, but it can also be repurposed to find shared articles.

Many interesting projects using newspaper data have used text reuse detection. The *Oceanic Exchanges* project is a multi-partner project using various methods to detect this overlap. This methods paper is really interesting, and used a similar starting point, though it then does an extra step of calculating 'local alignment' with each candidate pair, to improve the accuracy. (Smith, Cordell, and Mullen 2015)

Melodee Beals's *Scissors and Paste* project, at Loughborough and also part of *Oceanic Exchanges*, also looks at text reuse in 19th century British newspapers. Another project, looking at Finnish newspapers, used a technique usually used to detect protein strains to find clusters of text reuse on particularly inaccurate OCR. (**inproceedings-salmi?**; **inproceedings-blast?**)

14.1 Find reused text with the package `textreuse`

There are many methods by which reused text can be found. Usually these involve finding overlapping chunks of text. It is complicated when the OCR is not perfect - it's very unlikely that long strings of exactly the same text will be found, because of small differences. One package to do this in R is the `textreuse` package.

In this chapter, we'll demonstrate how to use this package and view and interpret the results from a sample of newspaper issues covering a single month.

14.2 Background

For this tutorial, we'll use a subset of the 1855 newspaper text data, taken from the Shared Research Repository. In 1855, despite the invention and spread of the telegraph, much of the circulation of news from London to regional hubs relied on the copies of newspapers or reports travelling by railway. News from overseas might arrive first in London where it was printed in papers there, and then sent onwards via the next train to Liverpool, Edinburgh, and so forth. This was particularly true of non-urgent or timely reports.

Computationally detecting text reuse will show some of this pattern, and is a way of understanding more about the circulation of news. By looking at the source newspapers for text printed in regional papers, we can get an idea of the patterns of news flow from one place to another. The results will show a variety of reuse: not just reused articles, but repeated advertisements, and quotations from third parties.

14.3 Method

The `textreuse` package expects a corpus in the form of a folder of text documents. To get the data in this format, the first step is to take the dataframe of articles we have been using throughout the book, filter to a single month, and export the text each article to a separate text document, in a new folder.

Following this, we'll use functions from the textreuse package to generate a datasetset of pairs of articles and a similarity score.

The final step is to merge this set of similarity scores with the text and metadata of the newspaper articles, so we can explore, read, and visualise the results.

14.3.1 Load necessary libraries

As in previous chapters, the first step is to load the libraries used throughout the tutorial If they are not installed, you can do so using the following:

```
install.packages('tidyverse')
install.packages('textreuse')
install.packages('data.table')
install.packages('ggtext')
```

```
library(tidyverse)
library(textreuse)
library(data.table)
```

```
library(ggtext)
```

14.3.2 Load the dataframe and preprocess

In the *extract text* chapter @ref(label), you created a dataframe, with one row per article. The first step is to reload that dataframe into memory, and do some minor preprocessing.

14.3.2.1 Combine Newspaper Articles into One Dataframe

As in previous chapters, either construct your own corpus by following Chapter 8 and Chapter 9, or download and open the ready-made .zip file with all issues from 1855. Next, get these articles into the correct format. See Chapter 10 for an explanation of this code:

```
news_sample_dataframe = list.files(path = "newspaper_text/",
                                   pattern = "csv",
                                   recursive = TRUE,
                                   full.names = TRUE)


all_files = lapply(news_sample_dataframe, data.table::fread)

names(all_files) = news_sample_dataframe

all_files_df = data.table::rbindlist(all_files, idcol = 'filename')


title_names_df = tibble(newspaper_id = c('0002090', '0002194', '0002244', '0002642', '0002

news_df = all_files_df %>%
  mutate(filename = basename(filename))


news_df = news_df %>%
  separate(filename,
           into = c('newspaper_id', 'date'), sep = "_") %>% # separate the filename into t
  mutate(date = str_remove(date, "\\.csv")) %>% # remove .csv from the new data column
  select(newspaper_id, date, art, text) %>%
  mutate(date = ymd(date)) %>% # turn the date column into date format
  mutate(article_code = 1:n()) %>% # give every article a unique code
  select(article_code, everything()) %>% # select all columns but with the article code fi
```

```
  left_join(title_names_df, by = 'newspaper_id') # join the titles
```

14.3.3 Add a unique article code to be used in the text files

Make a more useful code to use as an article ID. First use `str_pad()` to add leading zeros up to a maximum of three digits.

```
news_df$article_code = str_pad(news_df$article_code,
                                                width = 3,
                                                pad = '0')
```

Use `paste0()` to add the prefix 'article' to this number.

```
news_df$article_code = paste0('article_',
                                               news_df$article_code)
```

14.3.4 Filter to a single month

Let's limit the newspapers to articles from a single month. When text is being reused in newspapers, it would be reasonable to assume that it is more likely that the reusing text will do so shortly after the source was originally published. Comparing all the articles for a single month may be a good way to pick up some interesting reuse.

Use `filter()` to restrict the data to a single month, and save this as a new object:

```
sample_for_text_reuse = news_df %>%
  mutate(month = as.character(cut(date, 'month'))) %>%
  filter(month == '1855-09-01')
```

14.3.5 Export each article as a separate text file

The code below loops over each row in the news_sample_dataframe, writes the fifth cell (which is where the text of the article is stored) to a text file, using a function from a library called data.table called fwrite(), stores it in a folder called textfiles/, and makes a filename from the article code concatenated with '.txt'.

Before you do this, create an empty folder first, in the project directory, called `textfiles`

Once this has finished running, you should have a folder in the project folder called textfiles, with a text document for each article within it.

```
for(i in 1:nrow(sample_for_text_reuse)){


  filename = paste0("textfiles/", sample_for_text_reuse[i,1],".txt")

  writeLines(sample_for_text_reuse[i,5],con = filename)
}
```

14.4 Load the files as a TextReuseCorpus

After this, we'll import the files in this `textfiles` folder, and store them as an object from the textreuse package: a TextReuseCorpus.

14.4.1 Generate a minhash

The `textreuse` package uses minhashes to more efficiently store and compare the text documents (more information here). Before we generate the TextReuseCorpus, use the minhash_generator to specify the number of minhashes you want to represent each document. Set a random seed to make it reproducible:

```
minhash <- minhash_generator(n = 200, seed = 1234)
```

14.4.2 Create the TextReuseCorpus

Next, create the TextReuseCorpus

The function `TextReuseCorpus()` expects a number of parameters:

dir = is the directory where all the text files are stored.

tokenizer is the function which tokenises the text. Here we've used tokenize_ngrams, but you could use characters, or build your own.

n sets the length of the ngrams for the tokenizer. `n = 2` means the documents will be split into bigrams (sequences of two tokens).

minhash_func = is the parameters set using `minhash_generator()` above.

keep_tokens = Whether or not you keep the actual tokens, or just the hashes. There's no real point keeping the tokens as we use the hashes to make the comparisons.

This function can take a long time to run with a large number of documents.

```
reusecorpus <- TextReuseCorpus(dir = "textfiles/",
                               tokenizer = tokenize_ngrams,
                               n = 2,
                           minhash_func = minhash,
                           keep_tokens = FALSE,
                           progress = FALSE)
```

Now each document is represented by a series of hashes, which are substitutes for small sequences of text. For example, this is the first ten minhashes for the first article:

```
head(minhashes(reusecorpus[[1]]),10)
```

```
 [1] -1509664909 -1764812607 -1837533555 -1776955188 -1785365283 -1551982906
 [7] -1409803751 -2137549872 -1559991213 -2094002709
```

At this point, you could compare any single document's sequences of hashes to any other, and get its Jacquard Similarity score, which counts the number of shared hashes in the documents. The more shared hashes, the higher the similarity.

To do this over the entire corpus more efficiently, a *local Sensitivity Hashing* algorithm is used to solve this problem. This groups the representations together, and finds pairs of documents that should be compared for similarity. More details can be found here:

> LSH breaks the minhashes into a series of bands comprised of rows. For example, 200 minhashes might broken into 50 bands of 4 rows each. Each band is hashed to a bucket. If two documents have the exact same minhashes in a band, they will be hashed to the same bucket, and so will be considered candidate pairs. Each pair of documents has as many chances to be considered a candidate as their are bands, and the fewer rows there are in each band, the more likely it is that each document will match another.

The first step in this is to create the buckets. You can try other values for the bands.

```
buckets <- lsh(reusecorpus, bands = 50, progress = FALSE)
```

```
Warning: `gather_()` was deprecated in tidyr 1.2.0.
i Please use `gather()` instead.
i The deprecated feature was likely used in the textreuse package.
  Please report the issue at <https://github.com/ropensci/textreuse/issues>.
```

Next, use `lsh_candidates()` to compare each bucket, and generate a list of candidates.

```
candidates <- lsh_candidates(buckets)
```

Next, go back to the full corpus, and calculate the similarity score for these pairs, using `lsh_compare()`. The first argument is the candidates, the second is the full corpus, the third is the method (other similarity functions could be used). The output (the first 10 lines are below this code) is a dataframe with three columns, one for each pair of matched documents: the first two columns store the unique IDs for the documents, and the last the similarity score.

```
jacsimilarity_both = lsh_compare(candidates,
                                 reusecorpus,
                                 jaccard_similarity,
                                 progress = FALSE) %>%
  arrange(desc(score))

jacsimilarity_both %>% head(10) %>% kableExtra::kbl()
```

a	b	score
article_3092	article_3249	1
article_3092	article_3291	1
article_3092	article_3321	1
article_3092	article_3369	1
article_3092	article_3393	1
article_3092	article_3407	1
article_3092	article_3533	1
article_3092	article_3666	1
article_3092	article_3706	1
article_3092	article_3711	1

In order to see what this means in practice, there are a few more data wrangling steps to carry out.

Firstly, we'll merge the matches dataframe to the original sample of articles, using the article code. This means we can see the text and the metadata (the date and newspaper source) for each pair of articles. We can use this information to clean the data and get some more meaningful pairs.

The method compares all articles against all other articles, meaning articles from the same newspapers are compared to each other. It also doesn't have a minimum length, so sometimes very short articles (often just a few letters or punctuation marks, because of OCR and segmentation errors) will be matched with a very high score. We then select just the relevant columns, and filter out those where both documents are from the same newspaper.

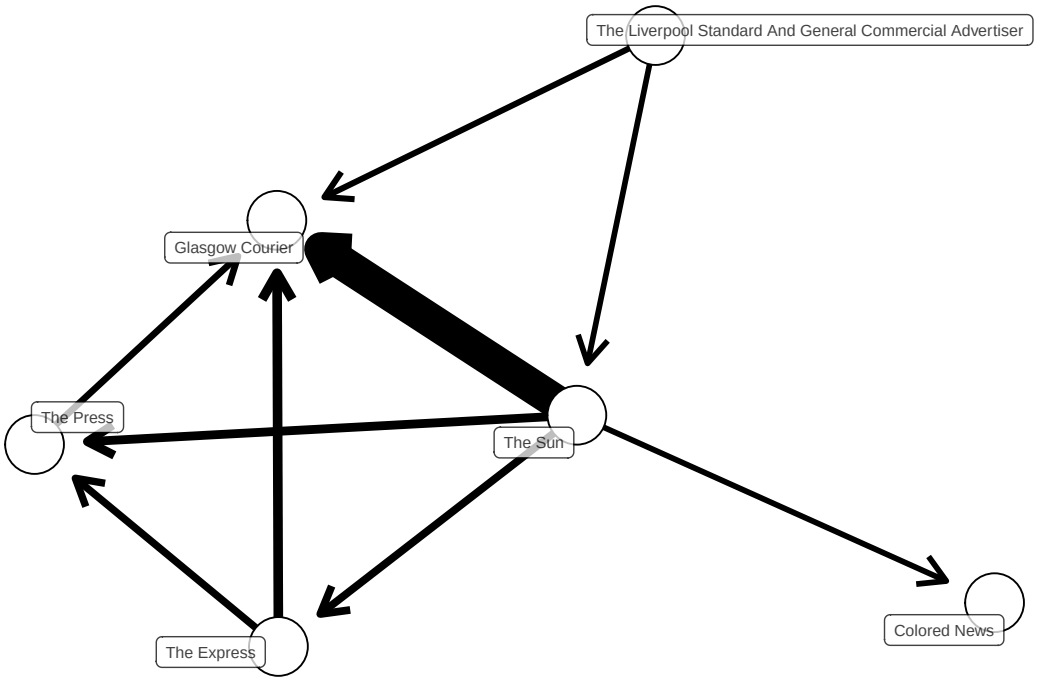
```
matchedtexts = jacsimilarity_both %>%
  left_join(sample_for_text_reuse, by = c('a' = 'article_code')) %>%
  left_join(sample_for_text_reuse, by = c('b' = 'article_code')) %>%
  select(a,b,score, text.x, text.y, newspaper_id.x, newspaper_id.y, date.x, date.y, newspa
  filter(!is.na(newspaper_id.x)) %>%
  mutate(count = str_count(text.x)) %>%
  filter(count>500) %>%
  mutate(text.x = paste0("<b>",newspaper_title.x, "</b><br>", text.x))%>%
  mutate(text.y = paste0("<b>",newspaper_title.y, "</b><br>", text.y))
```

The resulting file is a dataframe of every matched pair, including the titles and dates.

14.5 Analysis of the results

As a first step in understanding what exactly has been found, we could take a 'bird's eye' view of the patterns of reuse. If we filter to just include matches with high overlap, and then organise the data so that we only look at matches where one is published at least a day later than the other. With this, we can assume that one text is the 'source' text for an article published by the other.

This can be visualised using network analysis software. We'll count the total reuse from one title to another. Each title will be treated as a 'node' in a network, and the amount of reuse, going from older to newer, can be visualised as a link, with an arrow pointing from the amount of reuse from the source to the 'target'. The thickness of the line will be mapped to the volume of reused text. We'll also filter so that only reuse with a similarity score of at least .3 is considered.



Looking at the visualisation, one particularly clear pattern seems to be the reusing of text from *The Sun* in the *Glasgow Courier*.

In order to compare the two versions, we'll isolate a single pair, and use `ggplot2` to render it in a nicely-formatted text box.

```
data = matchedtexts %>%
  slice(1) %>%
  pivot_longer(cols = c(text.x, text.y)) %>%
  mutate(value = str_trunc(value, 1000))
```

This code, using the package `ggtext`, will display the two articles side-by-side:

```
# Create the plot
ggplot(data, aes(x = name, y = 1, label = value)) +
  geom_textbox(width = 0.4,
               box.padding = unit(0.5, "lines"), halign = 0.5, size = 3) +
  coord_cartesian(ylim = c(0, 2)) +
  theme_void()
```

```
Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
conversion failure on 'WORKS-ELM-BANK' in 'mbcsToSbcs': dot substituted for
<e2>
```

```
Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
conversion failure on 'WORKS-ELM-BANK' in 'mbcsToSbcs': dot substituted for
<80>

Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
conversion failure on 'WORKS-ELM-BANK' in 'mbcsToSbcs': dot substituted for
<94>

Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
conversion failure on 'WORKS-ELM-BANK' in 'mbcsToSbcs': dot substituted for
<e2>

Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
conversion failure on 'WORKS-ELM-BANK' in 'mbcsToSbcs': dot substituted for
<80>

Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
conversion failure on 'WORKS-ELM-BANK' in 'mbcsToSbcs': dot substituted for
<94>

Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
conversion failure on 'WORKS-ELM-BANK' in 'mbcsToSbcs': dot substituted for
<e2>

Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
conversion failure on 'WORKS-ELM-BANK' in 'mbcsToSbcs': dot substituted for
<80>

Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
conversion failure on 'WORKS-ELM-BANK' in 'mbcsToSbcs': dot substituted for
<94>
```

The Liverpool Standard And General Commercial Advertiser
GAS FITTINGS to his well−selected STOCK of CHANDELIERS, in CRYSTAL ORMOLU, and BRONZE, HALL LAMPS, BRACKETS, PENDANTS, &c., in the greatest possible variety, all of which, on inspection, will be found to consist of the newest designs of the day, and at such prices as will insure a large sale. The LAYING of PIPES, &c., in all its branches, by experienced Workmen, and properly qualified Fitters sent to any part of the Country.
WORKS...ELM−BANK FOUNDRY, GLASGOW.

The Liverpool Standard And General Commercial Advertiser
GAS FITTINGS to his well−selected STOCK of CHANDELIERS, in CRYSTAL ORMOLU, and BRONZE, HALL LAMPS, BRACKETS, PENDANTS, &c., in the greatest possible variety, all of which, on inspection, will be found to consist of the newest designs of the day, and at such prices as will insure a large sale. The LAYING of PIPES, &c., in all its branches, by experienced Workmen, and properly qualified Fitters sent to any part of the Country.
WORKS−ELM−BANK FOUNDRY, GLASGOW.

We can see that these two texts are indeed the same, with some small differences attributable to OCR errors.

14.5.1 Check Local Alignment

To check the specific overlap of two documents, use another function from textreuse to check the 'local alignment'. This is like comparing two documents in Microsoft Word: it finds the bit of the text with the most overlap, and it points out where in this overlap there are different words, replacing them with ######

First turn the text in each cell into a string:

```
a = paste(matchedtexts$text.x[1], sep="", collapse="")

b =  paste(matchedtexts$text.y[1], sep="", collapse="")
```

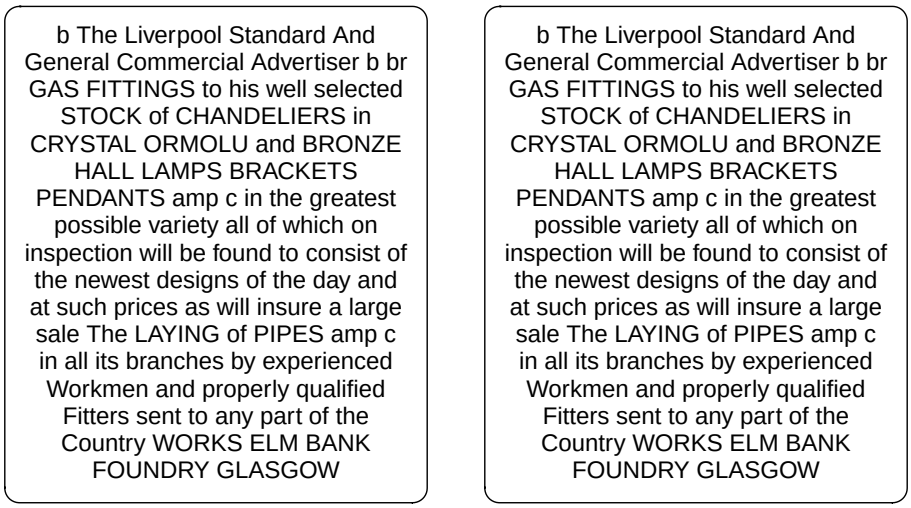
Call the `align_local()` function, giving it the two strings to compare.

```
aligned = align_local(a, b)
```

As before, we can use `ggplot2` to render the result so that it is easier to read:

```
aligned_df = tibble(texts = c(aligned$a_edits, aligned$b_edits)) %>%
  mutate(texts = str_trunc(texts, 1000)) %>%
  mutate(name = c('The Sun', 'Glasgow Courier'))
```

```
# Create the plot
ggplot(aligned_df, aes(x = name, y = 1, label = texts)) +
  geom_textbox(width = 0.4,
               box.padding = unit(0.5, "lines"), halign = 0.5, size = 3) +
  coord_cartesian(ylim = c(0, 2)) +
  theme_void()
```



14.6 Next steps

Text reuse is complex, because ultimately it is difficult to know whether is is 'genuine' text reuse, or something else such as reuse from a third party, boilerplate text, commonplaces, or famous quotations. In the above example, it's likely that the true pattern is that *The Sun* published this news in London along with many other titles at the same time, and the *Glasgow Courier* may have picked it up from any of them. Perhaps even more so than with other forms of text analysis, you'll want to really understand the sources and do close reading to fully understand the results.

You should experiment with the parameters, mostly the `n` parameter in the minihash generator, the `n` in the tokenizer, and the `bands` in the candidates function.

14.7 Further reading

David A. Smith and others, Computational Methods for Uncovering Reprinted Texts in Antebellum Newspapers, American Literary History, Volume 27, Issue 3, Fall 2015, Pages E1–E15, https://doi.org/10.1093/alh/ajv029

Ryan Cordell, Reprinting, Circulation, and the Network Author in Antebellum Newspapers, American Literary History, Volume 27, Issue 3, Fall 2015, Pages 417–445, https://doi.org/10.1093/alh/ajv028

15 Final Thoughts

I hope this book has shown you that it is possible to work with newspaper data to get meaningful and interesting insights, even for beginners. Working with 'real-world' data, with its OCR errors and the many gaps in the collection, can be a difficult but ultimately fruitful way of applying digital methods to historical datasets. Techniques such as these can be applied to other types of historical data, for example books or letters.

For a next step, I would recommend finding some specific question of interest which you think could be tackled with the collections made freely available and download your own corpus related to this question. Then, see how the different methods might help to answer it.

The use of large language models in the historical domain will ingest texts such as this at a huge scale, and understanding how these texts look is valuable in getting to know how these models work and where their limitations are. It's worth bearing in mind the specific material history of the collection, where it has come from, and how it has been digitised. As the volume of digitised material increases and becomes more accessible, this will be an increasingly important point.

15.1 Further Reading

There is a huge volume of literature on R, text analysis and newspaper digitisation. This is a small collection of recommended reading.

A useful list of coding resources: https://scottbot.net/teaching-yourself-to-code-in-dh/

A book on R specifically for digital humanities: http://dh-r.lincolnmullen.com

Geocomputation with R - a fantastic introduction to advanced mapping and spatial analysis. https://bookdown.org/robinlovelace/geocompr/

Use R to write blog posts: https://bookdown.org/yihui/blogdown/

R-Studio cheatsheets, which are really useful to print out and keep while you're coding, particularly at the beginning: https://rstudio.com/resources/cheatsheets/

Text mining with R - lots of the examples in this book are based on lessons from here: https://www.tidytextmining.com

The best introduction to R and the Tidyverse: https://r4ds.had.co.nz

A recent report on newspaper digitisation and metadata standards: https://melissaterras.files.wordpress.com/202
3.pdf

16 Final Thoughts

Techniques that can be done elsewhere

Newspapers have very particular makeup, particular type of text (temporality etc)

Hopefully will get more useful as more texts become available.

References

Ahnert, Ruth, Emma Griffin, Mia Ridge, and Giorgia Tolfo. 2023. "Collaborative Historical Research in the Age of Big Data," January. https://doi.org/10.1017/9781009175548.

Beelen, Kaspar, Jon Lawrence, Daniel C S Wilson, and David Beavan. 2022. "Bias and Representativeness in Digitized Newspaper Collections: Introducing the Environmental Scan." *Digital Scholarship in the Humanities* 38 (1): 1–22. https://doi.org/10.1093/llc/fqac037.

Fyfe, Paul. 2016. "An Archaeology of Victorian Newspapers." *Victorian Periodicals Review* 49 (4): 546–77. https://doi.org/10.1353/vpr.2016.0039.

Harris, P. R. 1998. *A History of the British Museum Library, 1753-1973*. London: British Library.

King, Ed. 2007. "Digitisation of British Newspapers 1800-1900." https://www.gale.com/intl/essays/ed-king-digitisation-of-british-newspapers-1800-1900.

———. 2008. "British Library Digitisation: Access and Copyright."

Mussell, James. 2014. "Elemental Forms: Elemental Forms: The Newspaper as Popular Genre in the Nineteenth Century." *Media History* 20 (1): 4–20. https://doi.org/10.1080/13688804.2014.880264.

Prescott, Andrew. 2018. "Travelling Chronicles: News and Newspapers from the Early Modern Period to the Eighteenth Century." In, edited by Siv Gøril Brandtzæg, Paul Goring, and Christine Watson, 51–71. Leiden, The Netherlands: Brill.

Ryan, Yann, and Luke McKernan. 2021. "Converting the British Library's Catalogue of British and Irish Newspapers into a Public Domain Dataset: Processes and Applications." *Journal of Open Humanities Data* 7. https://doi.org/10.5334/johd.23.

Shaw, Jane. 2005. "10 Billion Words: The British Library British Newspapers 1800-1900 Project: Some Guidelines for Large-Scale Newspaper Digitisation." https://archive.ifla.org/IV/ifla71/papers/154e-Shaw.pdf.

———. 2007. "Selection of Newspapers." *British Library Newspapers*. https://www.gale.com/intl/essays/jane-shaw-selection-of-newspapers.

Smith, David A., Ryan Cordell, and Abby Mullen. 2015. "Computational Methods for Uncovering Reprinted Texts in Antebellum Newspapers." *American Literary History* 27 (3): E1–15. https://doi.org/10.1093/alh/ajv029.

Smits, Thomas. 2016. "Making the News National: Using Digitized Newspapers to Study the Distribution of the Queen's Speech by W. H. Smith & Son, 1846–1858." *Victorian Periodicals Review* 49 (4): 598–625. https://doi.org/10.1353/vpr.2016.0041.

Tolfo, Giorgia, Olivia Vane, Kaspar Beelen, Kasra Hosseini, Jon Lawrence, David Beavan, and Katherine McDonough. 2022. "Hunting for Treasure: Living with Machines and the

British Library Newspaper Collection." In, 23–46. De Gruyter. https://doi.org/10.1515/9783110729214-002.

Wevers, Melvin. 2019. "Using Word Embeddings to Examine Gender Bias in Dutch Newspapers, 1950-1990." https://doi.org/10.48550/ARXIV.1907.08922.

Wevers, Melvin, and Marijn Koolen. 2020. "Digital Begriffsgeschichte: Tracing Semantic Change Using Word Embeddings." *Historical Methods: A Journal of Quantitative and Interdisciplinary History* 53 (4): 226–43. https://doi.org/10.1080/01615440.2020.1760157.