Chapter 7: Data

Many careful thinkers, when beginning a new project, seek out multiple archives for research. Katherine Bode, for example, argues for the importance of scholars building their own resources as a means of challenging the prevailing reliance on canonical texts and curated corpora, as solely engaging these risks reinforcing narrow or dominant histories (Bode 2018). A plurality of archives and texts offers an opportunity to think critically about the boundaries and biases of each archive under consideration. Many of the approaches to text mining we have profiled already – especially distinctiveness and distance measures – offer tools for systematically extending the study of the boundaries of the archive, such that a scholar can accumulate critical perspectives as they broaden their access to archives.

When a scholar is pursuing a complex research question, the relevant data must often be located, extracted, and prepared independently before analysis can begin. In historical research or other research problems across the social sciences and humanities, relevant sources may be scattered across different archives and exist only in incomplete or inconsistent formats. While basic guidance can provide some degree of comfort to the analyst who wishes to gain access to the diverse kinds of data available on the web, today's data ecosystem is highly diverse and even fragmented in terms of its infrastructures.

Data diversity brings new technical challenges. The method of extracting and analyzing data vary with how the data is stored and processed. Some archives offer well-documented APIs that facilitate access to records, while others provide only partial documentation or inconsistent metadata. These differences often reflect underlying disparities in the resources allocated to developing, maintaining, and preserving digital archives. Very few archives come packaged as a software repository as `hansardr` is. Therefore most data analysts will eventually want to engage the broader data ecosystem that exists beyond the data prepared for this book.

As Ian Milligan argues in *History in the Age of Abundance* (2019), the explosion of born-digital content has transformed the historian's archive, requiring new tools and methods to access, process, and interpret digital records at scale. In fields ranging from the humanities to computer science (Dobson 2019, Hayler et. al 2016, Hai-Jew 2018), analysts have documented approaches that are useful for aggregating text from the web. Once data has been properly accessed, cleaned, and stored, the analysis exercises from the rest of the book can be applied towards a variety of research questions – but only if the analyst is able to manipulate data.

The skills of accessing and manipulating data from the web are not well distributed. Our own experience suggests that, far from being a familiar task to most college seniors with a degree in data science, downloading and processing data from the web is often beyond the abilities of most students. Even computer scientists graduates skilled in data processing may face trouble when working with unfamiliar file types. Few textbooks offer a practical approach for analysts who wish to meet this growing digital archive, and the existing how-to guides published by historians and archives are, while useful, often disaggregated. Other The piecemeal approach to accessing one digital archive at a time rarely arms the analyst with the skills they need to feel comfortable ingesting large amounts of data.

To access the broader data ecosystem, analysts need to understand at least three modes of accessing data: (a) access to APIs, (b) bulk downloads (common on sites like Hathitrust or Harvard Dataverse), and (c) the topic of web scraping and extracting data through RSS feeds

In this chapter, we show how historians can engage with data housed in online archives–even though the data freely available may be found in disparate and irreconciliable types, many of which are increasingly born-digital.

A highly-skilled data scientist is able to engage these different infrastructures under any circumstance. The

ability to engage these fragmented infrastructures provides the key to becoming flexible enough to adopt the many archives that may been needed for a grand project.

In the age of AI, many analysts compile their own archives or have collected data from primary sources, resulting in a haphazard and sometimes poorly-labeled dataset that needs much cleaning before it is fit for analysis. Even when the data has been carefully treated by librarians or other custodians, the datasets rarely arrive in a neatly curated or structured state.

In result, analysts need a conceptual framework that enables them to identify and collect data for their archive, organize their data, and inform how they address missing data.

To handle the data with grace and ease, analysts also need to be aware of two further concepts: (i) the cleaning of data, (ii) data formats (how data is stored), data types (what kinds of data are being represented), and data structures (how data is organized for use and analysis),

Our hope is to guide readers' imaginations without limiting them. The concepts we introduce in this chapter are intended to be applicable across a wide range of historical research domains–whether in traditional historical scholarship, cultural heritage studies, or fields like the digital humanities or public history.

Through a series of exercises, the following chapter will introduce high-level concepts and best practices that will put the analyst on the way to data fluency.

APIs

Although vast quantities of cultural and historical materials are now available online through APIs (Application Programming Interfaces, or mechanisms that allow one piece of software to request and receive data from another), the ability to actually retrieve and work with these materials remains unevenly distributed. This matters because an increasing share of cultural production is "born digital," meaning the digital record is the original form rather than a later digitized copy (Ries 2022).

In our experience, collecting data from the web is not a routine skill — even among graduating seniors in data-science programs. Computer science students who are otherwise proficient at data wrangling can still struggle when confronted with unfamiliar file formats, download endpoints, or web-based collection workflows.

Part of the problem is that very few instructional materials teach these practices in a systematic way. Several works exist that outline the history of APIs, conceptualize them as a scholarly artifact, or discuss their significance to digital scholarship (Roued-Cunliffe 2017, Wachowiak 2022). However, they do not demonstrate how to use an API.

Most textbooks treat data as if it were already neatly stored on a local machine, while archivists' how-to guides tend to focus on individual platforms one at a time without offering a transferable workflow. This piecemeal approach may leave analysts adept at navigating a single repository but unprepared to ingest larger, more heterogeneous collections of digital sources at scale. Lessons on accessing one digital archive at a time rarely arms the analyst with the skills they need to feel comfortable ingesting large amounts of data from diverse sources (*Programming Historian*, "Lesson Index: APIs"; *Programming Historian*), "Lesson Index: Web Scraping").

Creating a Dataset from an API Endpoint

An API is a set of rules and protocols that allows one piece of software to interact with another. An API endpoint is a specific web address that a program can use to access data stored on another server. Analysts can connect to an API endpoint in two main ways: by sending standard HTTP requests directly from an R script, or by using an R package designed to interact with the API, which simplifies the connection and data retrieval process. This section will demonstrate both approaches.

Direct Connection to an API using an HTTP Request

Increasingly, public institutions and government agencies have adopted open data initiatives that make datasets accessible through APIs. Many cities, such as the City of Dallas and New York City, provide access to a range of datasets—such as 311 service requests, building permits, and police reports—via their open data portals.

APIs are a central mechanism through which analysts can access the vast quantities of data available online. In this sense, API endpoints function not merely as technical tools, but as gateways into contemporary digital archives. Large-scale datasets—ranging from open government records to social media content—are often made accessible through these API endpoints, which return data in structured formats such as CSV, or more commonly, JSON and XML.

Successfully interacting with open APIs—such as those provided by the City of Dallas or New York City—often depends on navigating several technical considerations, including user authentication, rate limits on the number of requests, options for customizing queries, and the specific data formats returned by the API. In what follows, we provide an overview of these considerations.

Sample APIs

Some examples of API endpoints include those offered by Reddit or The Guardian. Reddit, for instance, allows users to retrieve the latest posts in a given subreddit using an endpoint like:

`https://www.reddit.com/r/datascience/new.json`

This URL can be accessed programmatically using R, or the analyst can manually inspect the data by copying and pasting it into a web browser's address bar. When accessed, this URL returns a structured list of recent submissions to the /r/datascience Subreddit in JSON format.

Similarly, *The Guardian*, a global news platform, offers an API that enables users to search for news articles on a given topic, such as education, via a URL like:

`https://content.guardianapis.com/search?q=education`.

Using an API: Setting up an API Key

Using *The Guardian's* API requires additional set up. Accessing many APIs–particularly those provided by commercial services or large-scale platforms like *The Gaurdian* requires user authentication in the form of an API key.

An API key is a unique string of characters that identifies the user making the request. API keys are often used to monitor usage and enforce security. To obtain an API key, users typically need to register with the data provider. For example, to use *The Guardian*'s API, one must register and generate a key at:

`https://open-platform.theguardian.com`.

Once obtained, the key is appended to API requests—for example:

`https://content.guardianapis.com/search?q=education&api-key=your-api-key-here`.

Using an API Key: Authentification

Authentication is the process by which a user verifies their identity to the API, typically by including an API key with each request for data. Importantly, once an API key tethered to a user is submitted, the interaction is no longer anonymous—the exchange of information is now linked to a specific account, allowing the API provider to monitor usage, enforce rate limits, or restrict access if necessary. In this chapter, we will not use an API that requires authentication.

Practical Exercise: The API in Action

To illustrate how an API works, consider this example from the New York City Open Data portal, "NYC OpenData", which provides access to vast public datasets including population data, budgetary data, and data on city infrastructure. For this exercise we will look at just current and historic 311 service requests. The link below takes us to a webpage for the dataset *311 Call Center Inquiry*, which also provides a direct link to the API endpoint.

`https://data.cityofnewyork.us/City-Government/311-Call-Center-Inquiry/wewp-mm3p/about_data`.

Copying and pasting the API endpoint from this page,

`https://data.cityofnewyork.us/resource/wewp-mm3p.json`,

into a web browser will return nested data—typically in JSON format—which can be expanded to reveal the individual fields and their associated values. In this dataset, each entry includes the date and time the request was made, the agency responsible for handling the request, the name of the request, and a brief description. We will now read this data into the R environment for data processing and analysis.

In the following code, we use two new R packages: `httr` for sending HTTP requests, and `jsonlite` for parsing JSON data into a tidy data frame that mirrors the data structure we have been processing throughout the book.

The code sets the user up for a conversation with a server, with the following components of the dialogue: * A "request" is a message sent from a client (in this case, your R script) to a server, asking for specific information or resources—such as a dataset. * The "result" is stored in the "response" object, which represents the message returned by the server. A response includes a status code (e.g., 200 for success, 404 for not found), headers (which contain metadata), and the body (which holds the requested content, such as the JSON data containing the service request data).

Please note the following components of the API call: * The `GET()` function from the `httr` package sends an HTTP `GET` request to the specified API endpoint to retrieve data. * The `status_code()` function checks whether the request was successful. If it was, `content()` extracts the raw JSON text from the response body. * We use the `fromJSON()` from `jsonlite` to convert the data received from the API from JSON format a data frame. * The `flatten = TRUE` argument is necessary for simplifying any nested JSON structures into flat, readable columns suitable for analysis in R.

```r
library("tidyverse")
library("httr")
library("jsonlite")

# Define the JSON API endpoint
url <- "https://data.cityofnewyork.us/resource/wewp-mm3p.json"

# Make GET request to extract JSON content
response <- GET(url)

# If accessing the server is successful it returns the code 200
if (status_code(response) == 200) {
  # Extract and parse JSON content into a data frame
  json_content <- content(response, "text", encoding = "UTF-8")

  service_requests <- fromJSON(json_content, flatten = TRUE) %>%
    as_tibble() } else {
  stop("Failed to retrieve data. Status code: ", status_code(response)) }
```

Because this request was successful, we now have a data frame containing the shelter data:

```r
# Show just the first five columns
head(service_requests[, 1:5])
```

```
## # A tibble: 6 x 5
##   unique_id date                    time        date_time                agency
##   <chr>     <chr>                   <chr>       <chr>                    <chr>
## 1 100013773 2014-03-27T00:00:00.000 8:34:58 PM  2014-03-27T20:34:58.000 HPD
## 2 100013774 2014-03-27T00:00:00.000 9:23:41 AM  2014-03-27T09:23:41.000 DOHMH
## 3 100013775 2014-03-27T00:00:00.000 5:06:52 PM  2014-03-27T17:06:52.000 NYCOURTS
## 4 100013776 2014-03-27T00:00:00.000 12:45:40 PM 2014-03-27T12:45:40.000 CONED
## 5 100013777 2014-03-27T00:00:00.000 12:03:12 PM 2014-03-27T12:03:12.000 ACS
## 6 100013778 2014-03-27T00:00:00.000 10:24:56 AM 2014-03-27T10:24:56.000 DOF
```

Examining the Data Returned by APIs: JSON Records

APIs typically return data in structured formats such as JSON (JavaScript Object Notation), XML (eXtensible Markup Language), or CSV (Comma-Separated Values). JSON is widely used because of its interpretability across programming languages.

The service request API returned data in the JSON format. Using the `fromJSON()` function, and we converted the JSON to a tabular format for readability.

For the sake of understanding the data transformations that occurred, we also kept a copy of the nested JSON data. Many programmers favor JSON because it strikes a strong balance between human readability, machine parsability, and broad compatibility. JSON is the default for most modern APIs.

The window below gives an example of what that JSON data looks. Note that JSON uses key-value pairs, bracketed collections of information called "arrays", and .

Clicking on the `json_content` data in the Global Environment panel will also provide the means to explore the JSON data.


```
## [
##     {
##         "unique_id": "100013773",
##         "date": "2014-03-27T00:00:00.000",
##         "time": "8:34:58 PM",
##         "date_time": "2014-03-27T20:34:58.000",
##         "agency": "HPD",
##         "agency_name": "Department of Housing Preservation and Development",
##         "inquiry_name": "Heat or Hot Water Complaint in Entire Residential Building",
##         "brief_description": "Report a heat or hot water problem affecting an entire apartment build
##         "call_resolution": "Routed to Web Page"
##     },
##     {
##         "unique_id": "100013774",
##         "date": "2014-03-27T00:00:00.000",
##         "time": "9:23:41 AM",
##         "date_time": "2014-03-27T09:23:41.000",
##         "agency": "DOHMH",
##         "agency_name": "Department of Health and Mental Hygiene",
##         "inquiry_name": "Health Insurance Coverage Application Assistance",
##         "brief_description": "Get contact information for an assistor who can meet with you in perso
##         "call_resolution": "Information Provided"
```

```
##     }
## ]
```

When we made this request, we were only able to obtain 1,000 records (which is represented by 1,000 rows, where each row corresponds with a single record).

```
nrow(json_list)
```

```
## [1] 1000
```

However, if we return to the dataset's page —

`https://data.cityofnewyork.us/City-Government/311-Call-Center-Inquiry/wewp-mm3p/about_data`

—

We can see that the site reports the dataset contains millions of records. Technically, nothing went wrong during data extraction. Instead, we were "rate limited." Rate limiting is a common mechanism used by open data APIs to manage server load and prevent abuse. It restricts how many requests a user can make within a given time period. For example, unauthenticated users might be limited to 100 requests per hour, while authenticated users may be allowed significantly more—such as 10,000 requests per hour. This means that even though the data is publicly available, access can still be throttled based on usage patterns and credentials.

Most platforms open government data websites using the Socrata Open Data API (SODA) limit responses to 1,000 records per request by default. If a dataset exceeds this, only the first 1,000 rows are returned unless pagination is used. In the following example we revise the above code to account for pagination. This code runs two cycles, collecting up to 1,000 records per cycle for a total of up to 2,000 records. This code can be modified to extract additional records.

```
url <- "https://data.cityofnewyork.us/resource/wewp-mm3p.json"

# Pagination settings
limit <- 1000
# Will store the output from each iteration
service_request_data <- tibble()

# Run for two iterations: first with offset = 0, then with offset = 1000.
# This means the code will first collect rows 1-1000,
# and then collect rows 1001-2000 on the second iteration.
for (i in 0:1) {
  offset <- i * limit

  # Make paginated request
  response <- GET(url, query = list(`$limit` = limit, `$offset` = offset))

  if (status_code(response) != 200) {
    stop("Request failed. Status code: ", status_code(response)) }

  # Parse the response
  data_subset <- fromJSON(content(response, "text"), flatten = TRUE)

  # Save collected data
  service_requests <- bind_rows(service_requests, as_tibble(data_subset)) }
```

Importantly, even if you use pagination, you may still violate a website's Terms of Service. Some websites explicitly require delays—such as adding a sleep interval between requests—to avoid overloading their servers with requests that exceed typical human usage.

Using R Packages to Connect to an API

The examples above demonstrate basic interactions with APIs that require only a single, fixed URL. However, many APIs–especially those offering more extensive datasets–support more complex queries. These APIs require us to construct URLs, which can be a nontrivial task. This is because URLs often include multiple parameters, such as search terms, filters, authentication keys, and pagination controls, all of which must be formatted correctly to retrieve the desired data. After the data has been via the API, it still needs to be parsed into an R-friendly format.

Numerous R packages have been developed to simply interactions with APIs. These packages encapsulate the API's functionality within R functions and data structures, abstracting away the more complex tasks such as URL construction and parsing JSON responses. For example, the rOpenGov universe–composed of a community that develops open-source R packages to increase access to government data–offers several packages that simplify working with APIs, including `usdoj` and `oldbailey`. This section demonstrates the use of `usdoj` (2023).

`usdoj` provides more intuitive access to data from the U.S. Department of Justice's API by wrapping complex API interactions into simple R functions. It can be installed direclty from the rOpenGov universe. To do this, we first need to tell R to check the rOpenGov repository, and then we can install the package using `install.packages("usdoj")` as usual. By default, R installs packages from CRAN (the Comprehensive R Archive Network), but by adding this additional repository, we can access a wider range of packages.

```
options(repos = c(ropengov = 'https://ropengov.r-universe.dev'))

install.packages("usdoj")

library("usdoj")
```

We can now use `usdoj` functions to collect data from the U.S. Department of Justice. The following code will extract just 10 blog entries from the U.S. Department of Justice. Specifying the search direction as `DESC` instructs `usdoj` to extract the most recent blog entries.

```
blog_entries <- doj_blog_entries(n_results = 10, search_direction = "DESC")

head(blog_entries$teaser, 5)
```

```
## [1] "The Department's 2024 report highlights a range of efforts to encourage compliance with the Fre
## [2] "Enhancements to the Search Tool's artificial intelligence functionality improve public access t
## [3] "Enhancements to the Search Tool's artificial intelligence functionality improve public access t
## [4] "President Trump has been clear that securing the Southwestern Border of the United States is a
## [5] "Agency-wide FOIA data from Fiscal Year 2024 is now available for download on FOIA.gov "
```

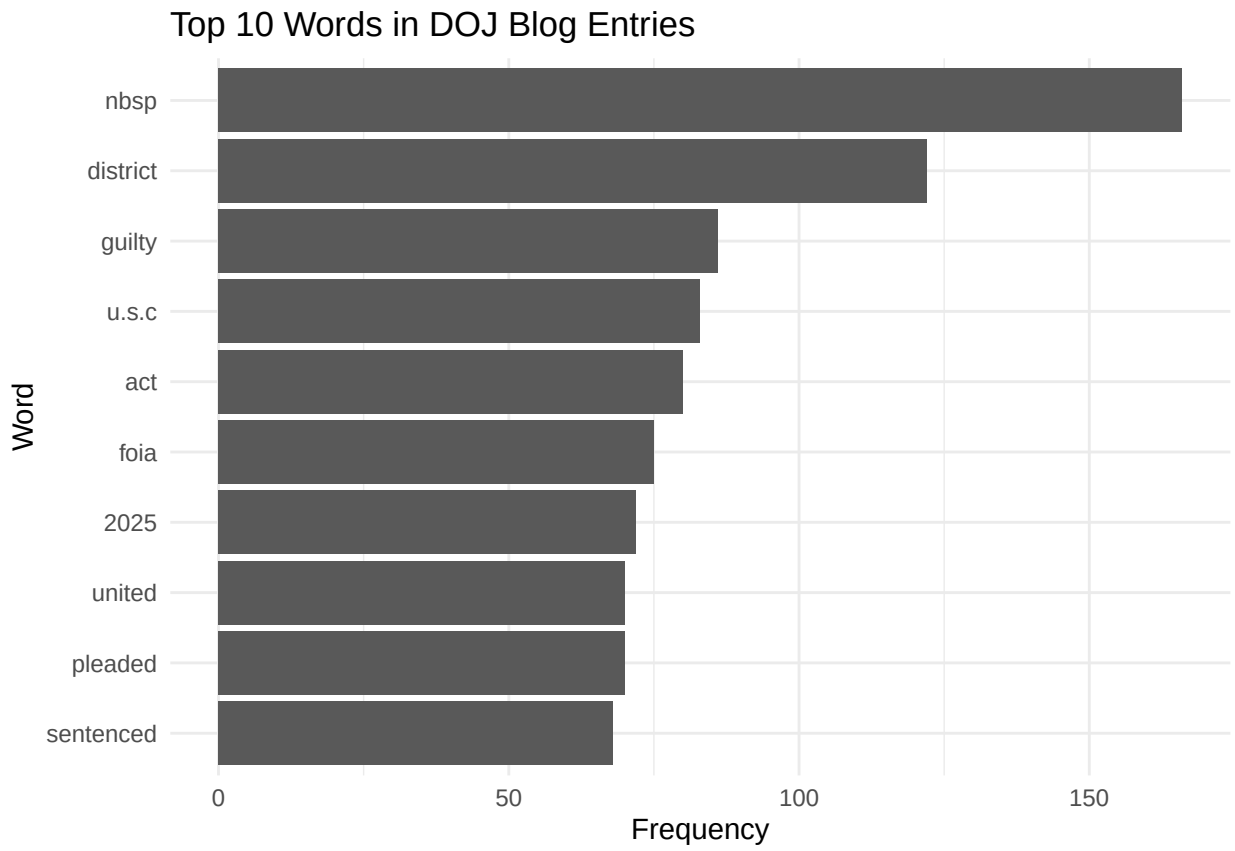
From this point, it is easy to analyze words.

```
library("tidyverse")
library("tidytext")

# Tokenize, remove stop words, and count word frequency
top_words <- blog_entries %>%
```

```r
  select(body) %>%
  unnest_tokens(word, body) %>%
  anti_join(stop_words, by = "word") %>%
  count(word, sort = TRUE) %>%
  slice_max(n, n = 10)

# Plot the top words
ggplot(top_words, aes(x = reorder(word, n), y = n)) +
  geom_bar(stat = "identity") +
  coord_flip() +
  labs(title = "Top 10 Words in DOJ Blog Entries",
       x = "Word",
       y = "Frequency") +
  theme_minimal()
```



APIs for Scholarly Research

Here we present a list of APIs for scholarly researcher derived from the University of Virginia Library with description provided by the authors of this book (University of Virginia).

Source	Description
AM Digital	Adam Matthew Digital is a major publisher of digitized primary source archives used in humanities and social science research. The company partners with libraries, archives, and museums to digitize rare historical materials—such as manuscripts, letters, maps, colonial records, newspapers, and ephemera—and curates them into themed online collections for scholarly use.
Coherent Digital	Coherent Digital is a U.S.-based digital publisher and aggregator that specializes in building digital collections of "hidden, lost, and endangered materials" — grey literature, non-traditional sources, web-only or at-risk content, and materials from underrepresented communities (especially in the Global South).
Dewey Data	Dewey Data is a platform designed to provide access to curated datasets from third-party premium providers that have traditionally been difficult for scholars to obtain. It aggregates people, place, and company data—such as foot-traffic, real estate, spending, web analytics, and points of interest. By negotiating licensing and integration on behalf of researchers, the platform makes commercially controlled datasets more accessible for scholarly work under academic-friendly terms.
Edinburgh University Press	Edinburgh University Press supports text and data mining through a licensed crawler-based system that provides institutional researchers with access to full-text XML of books, chapters, and journal articles for non-commercial academic use. It does not offer a public-facing API but rather a web crawler that once enabled, researchers can systematically harvest the XML corpus, making the platform suitable for large-scale linguistic, historical, or computational text analysis while maintaining compliance with publisher licensing terms.
Elsevier	Elsevier provides a suite of research APIs that give programmatic access to citation data, metadata, abstracts, and in some cases full text from its major platforms, including Scopus, ScienceDirect, Reaxys, PharmaPendium, and Engineering Village. These APIs support a wide range of scholarly workflows, from bibliometrics and literature reviews to chemical data retrieval and research evaluation. Most APIs are available at no cost for non-commercial academic users, provided their institution already subscribes to the underlying products.
HathiTrust	HathiTrust is a large-scale digital library that provides access to millions of digitized books and archival materials from research libraries, with strong coverage of public-domain works and U.S. government documents. For text and data mining, it offers the HathiTrust Research Center (HTRC), which provides computational access to full text through curated datasets and a Data Capsule environment that allows large-scale analysis while protecting copyrighted material. Researchers can download public-domain corpora directly or use controlled infrastructure to analyze in-copyright texts, making HathiTrust one of the most important platforms for scholarly text mining in the humanities and social sciences.

Source	Description
IEEE Xplore	Institute of Electrical and Electronics Engineers (IEEE) offers the IEEE Xplore API, a developer toolkit that enables programmatic access to its digital library of engineering, computer science, and technology content. Through this API users can query metadata, abstracts and, in certain cases, full-text articles. Access requires registration and an API key. It supports automated workflows like automated literature discovery and citation analysis.
New York Times API	The New York Times API is a set of publicly documented web APIs that allow developers and researchers to access structured data from the NYT, including articles, metadata, archives, books, reviews, and topical tags. It provides access to the Article Search API (historical and current articles with metadata), Books API (bestsellers and reviews), Archive API (full monthly issue metadata back to 1851), Movie Reviews API, and Top Stories API. Most endpoints return JSON and are designed for metadata-level access rather than full article text, though headlines, abstracts, keywords, and URLs are included. The API is widely used in digital humanities, journalism studies, and computational media research for building corpora, trend analysis, and topic modeling against news metadata, while full-text mining typically requires separate licensed access through ProQuest or NYT institutional agreements.
Taylor & Francis	Taylor & Francis hosts a very large corpus of scholarly journals and books across the humanities and social sciences as well as STEM fields, with particularly deep coverage in cultural studies, history, education, media studies, and the social sciences. It also includes both contemporary and historical materials and supports programmatic large-scale analysis of full text, which makes it useful for trend analysis, discourse studies, topic modeling, and longitudinal claims about scholarly language or concepts. Because its platform encompasses many influential journals across multiple disciplines.
Springer / BMC	Springer offers a public RESTful API (via the BMC platform) that provides structured access to open-access journal content in JSON or PAM formats, making it easy to integrate into computational workflows in R. The platform also maintains rich article-level metadata and real-time usage and citation metrics, which are especially useful for bibliometrics, scholarly communication research, and longitudinal studies of academic impact.

Source	Description
Library of Congress	The Library of Congress is one of the most valuable resources for large-scale text and data mining because it serves as both a national library and a long-term digital preservation infrastructure with extensive open collections spanning newspapers, manuscripts, maps, recordings, photographs, and government documents. Many of its holdings are digitized and made available through platforms like Congress.gov, Chronicling America, and the LoC Digital Collections, with rich metadata and machine-readable formats that support computational analysis. Its breadth across U.S. history, law, politics, culture, and media — combined with stable hosting and long-term stewardship — makes it a foundational corpus for humanities, social science, and civic-data research, especially for longitudinal or historically grounded projects.
ERIC	ERIC (the Education Resources Information Center) is a major open-access index of research in education, sponsored by the U.S. Department of Education, and it is one of the strongest resources for text and data mining in the learning sciences, education policy, and pedagogy. It offers structured metadata for journal articles, reports, dissertations, curricula, and grey literature from government and nonprofit education organizations, much of it available in full text. Because ERIC aggregates both peer-reviewed research and practitioner-oriented materials, it supports both scholarly analysis and policy-oriented studies, making it ideal for large-scale text analysis of educational trends, terminology shifts, interventions, and discourse over time.
Chronicling America	Chronicling America is a major open-access newspaper portal operated by the Library of Congress in partnership with the National Endowment for the Humanities, and it is one of the most important historical text corpora available for large-scale humanities research. It provides digitized U.S. newspapers spanning 1777–1963 along with rich bibliographic metadata and OCR text, making it well-suited for computational analysis of language, place-based reporting, public discourse, and cultural change over time. Because the platform is built for long-term preservation and public access, it is widely used for projects in digital history, media studies, and historical linguistics, particularly those requiring geographically and temporally broad sources.
Unpaywall	Unpaywall is a powerful resource for large-scale scholarly research because it provides open, programmatic access to metadata about which versions of over 54 million academic articles are freely available on the web. It maintains a massive database of OA copies harvested from repositories, preprint servers, and publisher pages, and its API allows researchers to look up open versions of paywalled articles using DOIs. Rather than hosting the articles itself, Unpaywall acts as a discovery layer for legal open-access versions, which makes it especially useful for building large corpora of scholarly texts without needing direct publisher agreements or subscriptions.

Source	Description
JSTOR	JSTOR is one of the largest digital libraries of scholarly journals, books, and primary sources spanning the humanities and social sciences, with particular strength in older journal runs (deep backfiles) and historically oriented scholarship. For computational text analysis, JSTOR provides Data for Research (DfR), a service that allows researchers to obtain structured metadata, n-grams, and full-text datasets derived from its corpus. It enables bulk export of data in machine-readable formats such as JSON or XML. Because JSTOR's core holdings are largely in-copyright, access to full text is controlled: researchers typically begin with n-gram or metadata-level exports, then may apply for privileged access upon IRB-style justification or institutional agreement. This model makes JSTOR valuable for linguistic, conceptual, and discourse analysis over time, especially in fields like history, literature, cultural studies, anthropology, philosophy, feminist theory, political thought, and other disciplines where journals preserved by JSTOR often represent the canonical scholarly record across the 19th–21st centuries.

RSS Feeds

RSS feeds are a simpler alternative to APIs for data extraction. For example, BBC News offers an RSS feed that returns content in XML format, which can be parsed to extract relevant information.

In the following example, only the titles of the news articles and their associated hyperlinks are retrieved from the feed.

```
library("rvest")
library("tidyverse")
library("xml2")

rss_url <- "https://feeds.bbci.co.uk/news/rss.xml"

rss_page <- read_xml(rss_url)

articles <- tibble(
  title = rss_page %>%
    xml_find_all("//item/title") %>%
    xml_text(),
  link = rss_page %>%
    xml_find_all("//item/link") %>%
    xml_text())

print(head(articles, 5))
```

```
## # A tibble: 5 x 2
##   title                                                                link
##   <chr>                                                                <chr>
## 1 Illegal migration tearing UK apart, Mahmood says                     http~
## 2 Marjorie Taylor Greene doubles down on Epstein files amid fallout with ~ http~
```

```
## 3 Arctic blast to move across UK as flood clean-up continues               http~
## 4 My mum was a 17-year-old free spirit, so she was locked up and put in a~ http~
## 5 Five young people dead following two-car crash                           http~
```

Bulk Downloads

Bulk downloads from institutional repositories are a common alternative to collecting data through a web service like an API. When working with large datasets, analysts may find bulk download options-—when available-—to be more efficient than relying solely on an API, which can take time to collect the data.

While this option might seem straightforward, it is often overlooked because inconsistencies in how repositories are structured or labeled can make bulk download options difficult to locate.

Making use of bulk downloads requires attention to file formats, file size, and licensing restrictions. Data may be packaged in tabular formats like CSV or TSV, which mirror the data frames we have been processing throughout this book. Other common formats include JSON and XML, which store data in hierarchical, nested structures often used for representing complex relationships or metadata.

Bulk downloads also include particular challenges for the data analyst. Bulk downloads often include the entire dataset in a single file. Opening such a large file can place significant demands on one's computer resources. In some cases, the file size may exceed the available memory, making it impossible to load the dataset into R (or another coding environment) without additional preprocessing or segmentation. We engage this issue below.

A notable limitation on bulk downloads is that the data is "static" and does not update dynamically, as is typically the case with API-based access. In other words, analysts may not receive the most current or up-to-date information if the dataset is still considered "live", such as is the case with accessing police reports in Dallas. This concern is generally less applicable for historical datasets that are fixed to a past time period, unless records are still being uncovered and added to the collection.

Examples of Bulk Download Websites

A representative example of a bulk download resource is provided by Old Bailey Online, which states:

"If you require the complete data (or the API is not suitable for your needs), it is available in XML format only from the University of Sheffield's data repository (ORDA): http://dx.doi.org/10.15131/shef.data.4775434." (Hitchcock et. al)

The dataset referenced includes 2,163 editions of the Proceedings and 475 Ordinary's Accounts, all encoded in XML.

Other popular sites for bulk downloads include: * the HathiTrust, a large-scale collaborative repository of digitized texts from academic, historic, and research institutions. Hathitrust offers materials such as historical magazines and newspapers, government documents, and literary works. * Harvard Dataverse, an open-access data repository designed to share, cite, and preserve research data across disciplines. It hosts datasets from a wide range of academic fields, including political science surveys, public health data, economics experiments, and qualitative social science studies. * Kaggle, a data science platform perhaps best known for its competitions, also hosts a wide array of community-contributed datasets, ranging from open science research (such as climate or genomics data) to pop culture topics (like movie ratings, video game sales, or lyrics datasets).

Working With Bulk Download Data

Importantly, while bulk data downloads seem straightforward, they can be difficult to work with in practice. Many open data portals offer complete dataset exports in formats such as CSV or JSON. The primary advantage of this approach is the ability to download substantial volumes of data in a single step.

Size of the download is another concern. A dataset may appear manageable in size because it is compressed–for example, as a .zip file–but unzipping it can produce gigabytes of text that overwhelm the memory or storage capacity of a typical laptop.Even when extraction is successful, trying to load the entire dataset into R with a single command like `read_csv()` or `fromJSON()` can cause the session to crash or become unresponsive.

The `hansardr` package was built with this challenge in mind: the full Hansard corpus was broken into smaller partitions to make it possible to load and analyze a large data set without consuming too many resources. But for other datasets, the analyst may need to "chunk" the data manually to make it manageable on a smaller machine.

"Chunking" data is the key approach for an analyst who needs to break a large dataset down into smaller, more workable sections.

Reading in a Large Dataset in Chunks

In cases where large dataset has not been partitioned, it is often necessary to process the data in smaller chunks, such as reading one line at a time or loading a subset of rows using functions like `read_csv_chunked()` from the `tidyverse`'s `readr` or `fread(nrows = ...)` from the `"data.table"()` package.

Below is an example of code for reading a large dataset in 1000 row chunks. For this sake of this practice, we will not download a large dataset onto our computers. We are simply offering a code example.

```
library("data.table")

# Read the first 1,000 rows
chunk1 <- fread("hansard_large.csv", nrows = 1000, skip = 0)

# Read the second 1,000 rows
chunk2 <- fread("hansard_large.csv", nrows = 1000, skip = 1000)
```

This is the same process but automated, so the user will read up to 100,000 rows of the data.

```
# Set parameters for reading data
file_path <- "hansard_large.csv"
chunk_size <- 1000
max_rows <- 100000

# Loop over the file in chunks
for (start_row in seq(0, max_rows, by = chunk_size)) {
  chunk <- fread(file_path, nrows = chunk_size, skip = start_row)

  # Do something to the data.
  # For example, the analyst could count the number of words
  # in the Hansard debates, save just the word count, but release the
  # Hansard debates to save memory.
}
```

Downloading and Scraping PDFs

When scholars and libraries work with data, frequently what they make available is not a structured file at all, but a series of PDFs – portable document format files that, once opened, have the appearance of a printed book. PDFs are often favored by scholars who appreciate the graphic format of print.

For the purposes of learning how to read in a pdf to R, let's explore how the process might work using a relatively unproblematic pdf composed of machine-readable text.

Please note that most pdfs that analysts might find on the web are not so well formatted, and they will have formatting or cleaning issues that the analyst must deal with – and which we will turn to in a later section.

Practicing with PDFs: Egil's Saga

It is possible to download a PDF file directly to one's computer and extract its text content for text mining. The following example demonstrates how this process can be automated.

We will practice with an 1893 translation of *Egil's Saga,* or the *Skalla-Grímssonar*, one of the classic Icelandic sagas, thought to be written in the 13th century, likely by Snorri Sturluson. It blends historical narrative, legend, and poetry, focusing on Egill Skallagrímsson, a fierce warrior and gifted poet. The copy of the text that we are using has been previously OCR'd from an original book, rendered as text, cleaned, and output as a machine-readable text with very little noise – making it an ideal subject on which to practice reading in data from a PDF.

Please note: The code that follows handles downloading and text extraction, but it does not include functionality for deleting the file afterward. This omission is intentional: to prevent any risk of accidentally deleting important or unrelated files, we have left the task of file deletion to the user's discretion. If the script is executed, the reader should manually delete the downloaded PDF file from their computer once it is no longer needed.

```
library("httr")
library("fs")
library("pdftools")
```

```
## Using poppler version 22.02.0
```

```
# Define the URL with the PDF to be downloaded
url <- "https://sagadb.org/files/pdf/egils_saga.en.pdf"

# Define the directory it will be downloaded to on one's computer
# The file will be downloaded to folder named "tmha_data"
destfile <- path("tmha_data", "egils_saga.en.pdf")

# Create directory if it doesn't exist
dir_create(path_dir(destfile))

# Download the PDF file
GET(url, write_disk(destfile, overwrite = TRUE))
```

```
## Response [https://sagadb.org/files/pdf/egils_saga.en.pdf]
##   Date: 2025-11-16 20:25
##   Status: 200
##   Content-Type: application/pdf
##   Size: 492 kB
## <ON DISK>  /home/stephbuongiorno/Repos/text-mining-for-historical-analysis/chapter_9_data/tmha_data/
```

```
# Extract text from the PDF
pdf_text_raw <- pdf_text(destfile)
```

```r
# Convert into a tibble with one row per page
tidy_egils_saga <- tibble(page = seq_along(pdf_text_raw),
                          text = pdf_text_raw)
```

The result is a data frame (specifically, a tibble) named `tidy_egils_saga`, which has two columns: one column contains the page numbers, and the other contains the corresponding body text from each page.

```r
tidy_egils_saga %>%
  slice(50:60)
```

```
## # A tibble: 11 x 2
##     page text
##    <int> <chr>
##  1    50 "They were of his company this winter, and sate next to the two brothe~
##  2    51 "It chanced one evening, when the king had gone to rest, as had also T~
##  3    52 "Thorolf said: 'Herein ye have so wrought, methinks, that it will not ~
##  4    53 "Alfred the Great had deprived all tributary kings of name and power; ~
##  5    54 "shameful act to harry before the battle was ended. Accordingly king O~
##  6    55 "his men to king Athelstan.\n\nThen the messengers ride all together, ~
##  7    56 "Egil was armed in the same way as Thorolf. He was girded with the swo~
##  8    57 "earls Hring and Adils were fallen, and a multitude of his men likewis~
##  9    58 "hewed on either hand of him, felling many men. Thorfid bore the stand~
## 10    59 "king. Egil sat down there, and cast his shield before his feet. He ha~
## 11    60 "Then those men were healed whose wounds left hope of life. Egil abode~
```

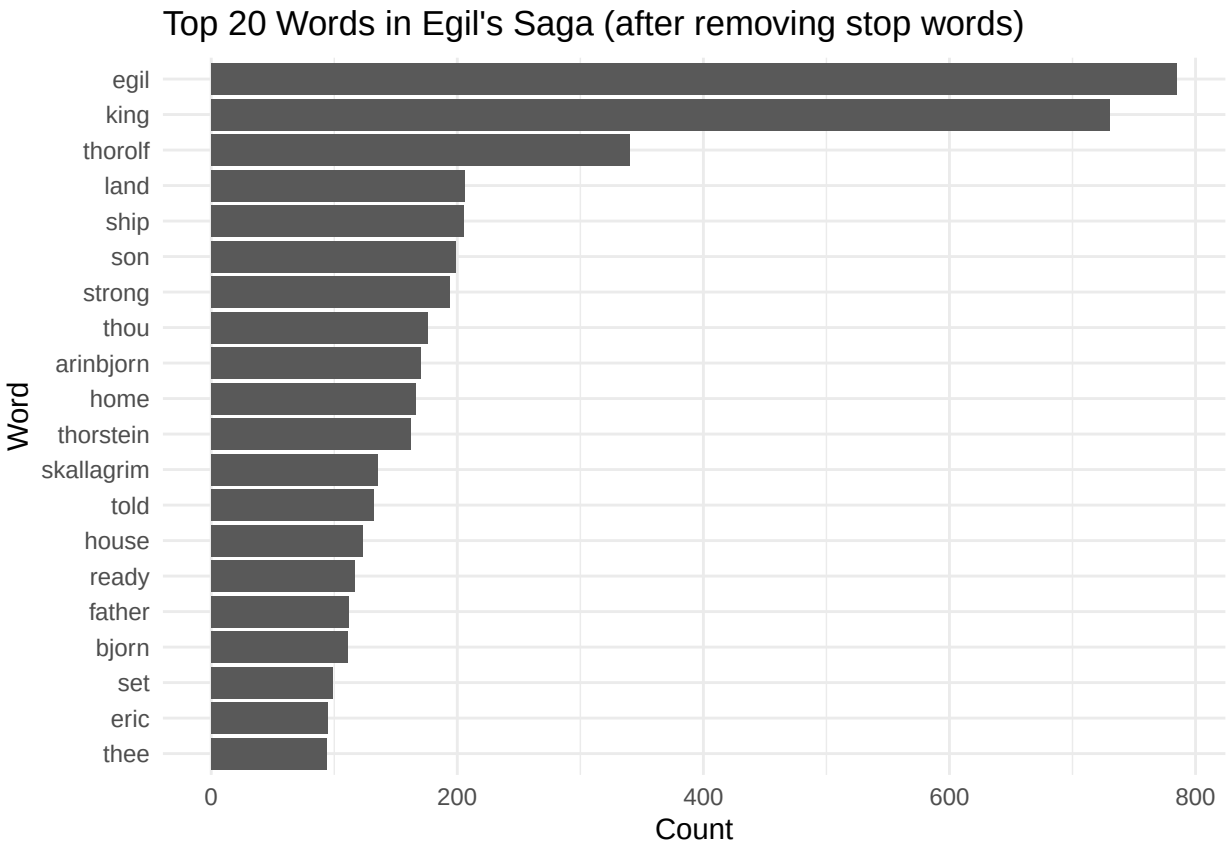
With just a few lines of code to locate, download, and scrape the PDF file, it is now easy to count the top words in *Egil's Saga*.

```r
library("tidyverse")
library("tidytext")

data(stop_words)

# Unnest tokens, remove stop words, count words, and select the top 20
top_words <- tidy_egils_saga %>%
  unnest_tokens(word, text) %>%
  anti_join(stop_words, by = "word") %>%
  count(word, sort = TRUE) %>%
  slice_max(n, n = 20)

# Visualize top 20 words
top_words %>%
  ggplot(aes(x = reorder(word, n), y = n)) +
  geom_col() +
  coord_flip() +
  labs(title = "Top 20 Words in Egil's Saga (after removing stop words)",
       x = "Word",
       y = "Count") +
  theme_minimal()
```



For transparency and book keeping, the analyst may want to add additional metadata—such as the book's title (Egil's Saga), the author (Snorri Sturluson), and the publication date of the English translation (1893).

```
tidy_egils_saga_with_metadata <- tidy_egils_saga %>%
  mutate(author = "Snorri Sturluson",
         book = "Egil's Saga",
         date = "1893")
```

This metadata helps an analyst keep track of key information about the book. However, because each row in the author, book, and date columns contains the same values, the metadata is redundantly stored in the data frame.

```
library("gt")

gt(head(tidy_egils_saga_with_metadata))
```

Getting PDFs Ready for Computer Readability: Optical Character Recognition (OCR)

The foregoing exmaple used a deliberately simple-to-read PDF to demonstrate a process. However, in real life, the PDFs collected by analysts or digitalized by museums, archives, corporations, or government institutions may or may not have been prepared for computer readability.

One rule of thumb for deciding whether a PDF is machine readable is to open the document in a computer application like Adobe or Preview, designed for examining PDFs. If the text on the page of the PDF can

be highlighted by the user of an app, then the text is ready for digital processing, i.e. it has already been "OCR'd," or provided to an algorithm called "optical character recognition," which uses visual pattern detection to recognize the text on the page.

In the past, analysts typically paid for proprietary software when they wanted to apply OCR to PDF files, and the OCR was of varying quality, with OCR packages from twenty years ago producing many errors, and more recent OCR showing a marked improvement and occasionally the ability to identify handwritten scripts from the past, especially when the OCR program had been trained on many examples of the script translated by patient scholars.

The results are promising. For printed Ottoman Turkish, scholars using platforms like Transkribus have trained Handwritten Text Recognition (HTR) models on ~386 pages of late-Ottoman periodicals, achieving a character error rate of ~7.2 % as of June 2023 (Digital Orientalist 2023).Printed Devanagari (Hindi, Sanskrit, etc.) has been widely supported through engines like Tesseract, E-Aksharayan, and customized models (e.g. C-DAC). Chinese handwritten character recognition is a long-standing research domain. Techniques such as DenseRAN (radical-aware networks) and recurrent neural net architectures (LSTM/GRU) have achieved high accuracy in offline recognition contexts—even on large character sets up to 30,000 characters, as shown in industry systems (e.g. Apple's Scribble) and academic datasets like ICDAR-2013 with strong performance (Zhang et al. 2016).

Today, OCR can be performed using AI tools, typically at a fraction of the cost. Although best practices in text recognition are beyond the scope of this book, we have heard reports of scholars having success with using AI to recognize hand-written text in Latin and Spanish using out-of-the-box LLMs such as ChatGPT.

A cursory review of recent articles suggests that the best results for handwritten records involve extensive training sets, translation, and sampling. Breakthrough success has been reported for using AI trained on Chinese characters to recognize Chinese texts (Chen et. al 2025). A Stanford team is applying deep ML and custom training to recognize handwritten text from records in Ottoman, Classical Arabic, and Ottoman Turkish ("Arabic Script OCR/HTR," accessed 2025). We read of good results on Ancient Indian records from Brahmi, Grantha, Devanagari and Tamil manuscripts, ancient Chinese records, handwritten East Syriac, and Latin registers from the Vatican (Yadav 2025, Majeed and Hossani 2024, Zhang 2024, "In Codice Ratio"). The reports suggest that text mining for historical analysis will soon be applied to a widening collection of scholarly texts from different times and places.

Once a collection of PDFs has been OCR'd, it is possible to use the resulting PDF files to create a data frame for text mining.

Web Scraping

Before the widespread availability of APIs, data collection from websites was commonly performed through web scraping, which involves the extraction of information from the underlying HTML structure of web pages. In many cases, web scraping is no longer necessary. Numerous websites now provide structured data access through APIs. But in certain circumstances, web scraping can still help the analyst to prepare information from websites as a dataset.

In R, the `rvest` package facilitates web scraping by parsing HTML content into a structured representation, commonly referred to as the HTML tree.

An additional precaution is to add code to slow down the web scraping process. To avoid overloading the server hosting the webpage, requests are typically rate-limited during this process.

Practicing Web Scraping: Retrieving the Code for *Text Mining for Historical Analysis*

The following example illustrates the initial step in this process: retrieving the HTML content of the GitHub repository page for this book followed by a snippet of the page's HTML.

```
url <- "https://github.com/stephbuon/text-mining-for-historical-analysis"

page <- read_html(url)
```

```
## <!DOCTYPE html>
## <html lang="en" data-color-mode="auto" data-light-theme="light" data-dark-theme="dark" data-a11y-...
## <head>
## <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
## <meta charset="utf-8">
```

Understanding HTML Code and Finding the Content

To make practical use of webscraping, the analyst first needs to become aware of the components of the HTML code that structures a website. On a webpage, each element of HTML code gives the computer specific information about what to display, for instance headers, paragraphs, and links. This information is given in a specific structure in the HTML document, which has a tree-like branching structure.

There are two main features of HTML code: * CSS selectors, derived from web design conventions, identify elements based on tags (e.g., "`p`" for paragraphs), classes (e.g., "`.title`"), IDs (e.g., "`#main`"), or structural relationships (e.g., "`div > p`" for paragraph tags directly nested in a `div`). * In contrast, XPath offers a more expressive syntax for specifying paths through the HTML tree, enabling selection based on element attributes, positions, or nested hierarchies (e.g., "`//div[@class='content']/p`" selects all paragraph tags within div elements that have the class content).

As an example, in the following code the CSS selector `p` would select both paragraph elements inside the `<div>`.

```
<div class="article">
  <h2>A Heading</h2>

  <div class="text-block">
    <p>First paragraph.</p>
    <p>Second paragraph.</p>
  </div>

  <ul class="links">
    <li><a href="#">Link one</a></li>
    <li><a href="#">Link two</a></li>
  </ul>
</div>
```

To extract the content itself, the analyst uses these selectors to locate relevant elements in the HTML tree and then retrieve their text. In practice, this involves three steps: downloading the page's HTML, parsing it into a structured document, and applying CSS or XPath queries to extract the desired nodes. For example, using the `rvest` package in R, the following code retrieves all paragraph text from a webpage using the `p` tag as the selector:

```
# Step 1: Read the webpage
url <- "https://example.com"
page <- read_html(url)

# Step 2: Select and extract all <p> tags using a CSS selector
paragraphs <- page %>%
  html_elements("p") %>%      # CSS selector
  html_text(trim = TRUE)      # extract clean text

print(paragraphs)
```

```
## [1] "This domain is for use in documentation examples without needing permission. Avoid use in opera
## [2] "Learn more"
```

Data Cleaning

Historic documents pose unique challenges for text mining due to inconsistencies in spelling and the physical degradation of materials over time. Additionally, digitization often introduces errors, such as mistakes from Optical Character Recognition (OCR), which complicate analysis.

As analysts aim to uncover language patterns and insights within historical texts, data cleaning becomes an important step in preparing these documents for analysis. Data cleaning is the process of identifying and correcting errors, inconsistencies, and inaccuracies in data to ensure its quality and reliability.

This section provides analysts with the conceptual foundation and practical tools necessary for effective data cleaning. We argue that data cleaning should not be treated as a purely technical task, but rather as an interpretive and methodological decision: analysts must consider whether data should be cleaned for a given analysis, and if so, determine which forms of cleaning are appropriate in light of the research goals. Our aim is to demonstrate the techniques for cleaning data while remaining critical and aware of the editorial decisions involved in this process.

Identifying Messy Data

Defining "messy" is not a straightforward task. Rather, determining what counts as messy is subjective and should be guided by a clear rationale tied to the goals of a given analysis. Throughout *Text Mining for Historical Analysis*, we have made baseline assumptions about what constitutes "messy" in ways we believe support certain forms of computational inquiry. For example, we provide a version of Hansard–via `hansardr`–that has disambiguated speaker names. This column is meant to address inconsistencies and ambiguities in the original records, as well as remove special symbols or other artifacts that may interfere with our research goals, like debate text appended to a speakers name.

We offer this "cleaned" speaker data because we believe it can help structure the data in a way that allows speech to be attributed to a single member of Parliament. This is particularly useful for analyses that rely on mapping discourse to individual actors. Similarly, we have removed stop words and eliminated symbols such as brackets or asterisks when they appear in discourse. In doing this, we have made certain assumptions

about which textual features are considered meaningful for our purposes and which can be treated as noise–assumptions that reflect specific analytical priorities and inevitably shape the kinds of historical arguments that can be made.

Our approach to identifying and managing "messy" data is not shared unilaterally across all domains. Certainly, for some researchers the "mess" we have identified might be considered a rich feature of the historical record that should be considered during an analysis. For instance, Ryan Cordell (2017), has argued that messy data is not a problem to be solved so much as a condition to preserved and engaged critically. He notes that digital artifacts are often treated as though they are meant to serve as faithful replicas of the original texts, as if they are most valuable if they contain the exact words without any OCR errors. Yet, as he argues, when "we treat the digitized object primarily as a surrogate for its analog original, we jettison the most compelling qualities of both media. The unique use of the digital medium, broadly considered, is the capacity to computationally trace patterns across corpora of various sizes" (193). Matthew Kirschenbaum (2008) calls for providing an "account of electronic texts as artifacts—mechanisms—subject to material and historical forms of understanding" (ENTER). In this way, Kirschenbaum challenges the idea that digital texts are immaterial or purely abstract representations. Instead, when we understand electronic texts as artifacts—objects that are produced, stored, and transmitted through specific material mechanisms (like hard drives, servers, or software systems) we can critically engage with the artifact's existence in the world.

We do not disagree with these points. Texts–especially historical ones–are mediated through layers of transmission, digitization, and encoding. Depending upon one's approach to analysis, to clean too aggressively is to erase and to normalize to heavily is to flatten. Indeed, a historian's job is not just to extract patterns but to understand data provenance. Here, however, we have defined "messy" in a way that we think is meaningful for computational historical analysis, which engages data through text mining. For our purposes here, common signs of messy data include:

- Special characters (e.g. an asterisk in a speaker's name)
- Duplicate rows
- Misspellings
- Numeric values stored as text/character data types
- White space
- Missing values
- Illogical or unintended text concatenations resulting from formatting

From the Hansard data, we provide the following example of an illogical text concatenation: the word "said" is included as part of the speaker's name. As a result, "The Bishop of Exeter" and "The Bishop of Exeter said" appear as two separate speakers in the dataset. If we were to group the data by speaker and count the most frequent words, this error would lead to treating a single individual as two distinct parliamentarians.

```
library("hansardr")

data("speaker_metadata_1840")
```

```
speaker_metadata_1840 %>%
  filter(sentence_id == "S3V0051P0_728")
```

```
## # A tibble: 1 x 6
##   sentence_id   speaker        suggested_speaker ambiguous fuzzy_matched ignored
##   <chr>         <chr>          <chr>                 <int>         <int>   <int>
## 1 S3V0051P0_728 The Bishop of~ ""                        0             0       0
```

In this example, an apostrophe is included at the beginning of the name "Mr. Speaker" and a hyphen appears before the name "Lord Ashley."

```
speaker_metadata_1840 %>%
  filter(sentence_id == "S3V0060P0_2194")
```

```
## # A tibble: 1 x 6
##   sentence_id    speaker      suggested_speaker  ambiguous fuzzy_matched ignored
##   <chr>          <chr>        <chr>                  <int>         <int>   <int>
## 1 S3V0060P0_2194 'Mr. Speaker charles_shaw-lefe~         0             0       0
```

```
speaker_metadata_1840 %>%
  filter(sentence_id == "S3V0051P0_18193")
```

```
## # A tibble: 1 x 6
##   sentence_id     speaker      suggested_speaker ambiguous fuzzy_matched ignored
##   <chr>           <chr>        <chr>                 <int>         <int>   <int>
## 1 S3V0051P0_18193 -Lord Ashley ""                        0             0       0
```

These are just a few examples from the many typographical anomalies existing within the digitized version of the Hansard corpus.

Data Cleaning Using a Large Langauge Model

The problem of cleaning data is a long one. A common approach to addressing the unwanted symbols, formatting issues, and inconsistencies is through the use of regular expressions (often shortened to "regex"). We have already demonstrated the regular expressions can be used to target specific, unwanted data. Here we briefly demonstrate a very simple use of a regular expression to clean the "'" from the name "Mr. Speaker."

```
speaker_metadata_1840 %>%
  filter(sentence_id == "S3V0060P0_2194")
```

```
## # A tibble: 1 x 6
##   sentence_id    speaker      suggested_speaker  ambiguous fuzzy_matched ignored
##   <chr>          <chr>        <chr>                  <int>         <int>   <int>
## 1 S3V0060P0_2194 'Mr. Speaker charles_shaw-lefe~         0             0       0
```

```
speaker_metadata_1840 %>%
  filter(sentence_id == "S3V0060P0_2194") %>%
  mutate(speaker = str_replace_all(speaker, "'", ""))
```

```
## # A tibble: 1 x 6
##   sentence_id    speaker     suggested_speaker   ambiguous fuzzy_matched ignored
##   <chr>          <chr>       <chr>                   <int>         <int>   <int>
## 1 S3V0060P0_2194 Mr. Speaker charles_shaw-lefev~         0             0       0
```

Many guides exist for defining and using regular expressions. For this reason, we will not include one here and instead suggest reading

https://cran.r-project.org/web/packages/stringr/vignettes/regular-expressions.html

for a comprehensive encyclopedia of regular expressions in R.

While effective, writing our own regular expression-based approaches to data cleaning can become increasingly intricate and difficult to scale, such as demonstrated by the following code that relies on `stringr` for handling strings.

```r
# Messy string
messy_string <- paste0("<p>We!!!   can clean   messy--",
                       "<b> text using </b>.. ,    regular expressions </p>")

# Clean it with a complex regex pipeline that
# removes all HTML tags
# removes all non-word and non-space characters
# replaces multiple spaces with a single space
# trims leading/trainign spaces
clean_string <- messy_string |>
  str_remove_all("<[^>]+>") |>
  str_replace_all("[^\\w\\s]", "") |>
  str_replace_all("\\s+", " ") |>
  str_trim()

print(clean_string)
```

```
## [1] "We can clean messy text using regular expressions"
```

Writing complex regular expressions can be time-consuming and difficult to manage, making alternative approaches increasingly appealing. Recently, large language models (LLMs) have been explored as tools for data cleaning. Unlike traditional programming methods, LLMs are not bound by rigid symbolic logic. They can instead interpret natural language instructions. This means we can simply describe how we want a messy string cleaned, and the model can perform the task without requiring detailed code.

However, relying on LLMs to directly clean text introduces certain risks. Unlike traditional programmatic methods, which transform text by manipulating the original content, LLMs generate entirely new text based on probabilistic models. Because the output often closely resembles the input, it is easy to mistake this generated text for a faithful reproduction—when in fact it is newly generated text, not a transference of the original text. This generation makes LLMs more prone to introducing errors or "hallucinations"–that is, producing text that is semantically plausible but factually incorrect. Such risks are heightened when (a) the input text is significantly different from the data the model was trained on–as is often the case with historical or domain-specific texts–or (b) when the model is given an overly large or complex input to process at once.

For this reason, we have found it most effective to use the LLM to identify issues in the text and generate code—such as R scripts or regular expressions—that can then be executed programmatically to clean the data. This approach lets us have our cake and eat it too: we benefit from the LLM's ability to quickly construct complex cleaning logic, while maintaining the safety, transparency, and reproducibility of programmatic data processing. Rather than asking the LLM to rewrite the passage directly and copying its regenerated output, this approach allows us to preserve the original text while applying transparent, repeatable transformations.

For example, we provided this prompt and string to OpenAI's GPT-4:

> Write regular expression(s) to clean this text. Return your response as R code. "
>
> We!!! can clean messy– text using .. , regular expressions
>
> "

GPT-4 returned this code:

```r
library("stringr")

# Remove HTML tags
```

```r
messy_string <- gsub("<[^>]+>", "", messy_string)

# Replace multiple punctuation marks with a space
# (put hyphen first or last to avoid an error where the code
# goes out of range
messy_string <- gsub("[-!.,]+", " ", messy_string)

# Collapse multiple spaces and trim
text <- str_squish(messy_string)

print(text)
```

```
## [1] "We can clean messy text using regular expressions"
```

Since a single regular expression is often insufficient to handle all types of messy input, it may be desirable to apply this approach iteratively, for example, by using a `for` loop that calls the API for a LLM and dynamically generates regular expressions tailored to the specific cleaning needs of the given text.

Below, we provide an example of what this might look like in pseudo-code, or code that resembles real code in order to communicate an idea, but is not actually intended to run. To use similar code, the reader will need to obtain and configure an API key for access to a LLM.

```
library("stringr")

max_iterations <- 5

for (i in 1:max_iterations) {
  for (row in 1:nrow(df)) {

    text <- df$text[row]

    llm_suggested_regex <- call_llm_api(
      prompt = paste("Suggest a regular expression to further clean this text:\n", text))

    text <- str_replace_all(text, llm_suggested_regex, " ")

    # Update the data frame with the cleaned text
    df$text[row] <- text } }
```

Interpreting Data Format, Type, and Structure

Digital methods are now deeply embedded in many areas of humanities research. As a result, it is increasingly necessary to move beyond a surface-level familiarity with the technical aspects of computational text analysis. We argue it is necessary to approach digital media not merely as "tools," but as complex systems shaped by specific technical features. This shift requires a transition toward computational thinking as an approach that emphasizes the conceptual understanding of how data is structured, processed, and interpreted. We argue that adopting computational thinking is essential for engaging more critically with digital sources, and for integrating these methods into the interpretive practices of the humanities.

This section introduces core technical concepts—data format, type, and structure—that shape how textual data is processed and analyzed. In many computational disciplines, these concepts are treated as foundational and are typically addressed early, as they serve as the building blocks to more advanced forms of analysis.

In this book, however, we intentionally set these concepts aside because our approach to text mining in R largely handles these data complexities. Most of our work involved character data (e.g., words in a debate) and integer data (e.g., frequency counts of top words), which were stored in data frames compatible with the (primarily) `tidyverse` tools we selected. Our deliberate approach allowed us to move straight into historical analysis without worrying about data complexities. Here, we pause to examine these underlying structures more deliberately, as a deeper understanding of them can inform strategies for different types of analyses. This section thus reframes some technical fluency as a critical component of interpretive practice in computational text analysis.

Data Format

At this point in the chapter, we have already examined and engaged with several different data formats. Instead of reiterating the same exercises here, we will provide a summation of data formats.

Data formats refer to the way information is stored and represented. It has an impact on how it is accessed by both analysts and machines, and distinct affordances and trade-offs, depending on the structure and goals of historical analysis.

- CSV/TSV (Comma- or Tab-Separated Values) – Both CSV and TSV are flat tabular formats in which each line represents a row and values are separated by a delimiter–commas for CSV, tabs for TSV. Both formats are well-suited for structured, spreadsheet-like data, but they are less effective for representing complex or hierarchical relationships.
- JSON (JavaScript Object Notation)– A hierarchical format often used in APIs and web services. It allows for nested values, making it useful for representing complex, structured data such as Reddit posts or metadata-rich archives. JSON data often requires flattening before analysis in tools like R.
- XML (eXtensible Markup Language) – A tree-structured markup language designed to store richly encoded text and metadata. XML is widely used in scholarly editions and historical datasets (e.g., Old Bailey) and supports deeply nested content like legal documents or manuscripts. It is verbose but highly descriptive. XML is more verbose than JSON, and its verbosity enables more granular control over how texts are described and interpreted—especially when fine distinctions between text types or historical layers matter.
- HTML (HyperText Markup Language) – The foundational format of web pages. When scraping websites, analysts parse HTML to extract specific content.
- PDF (Portable Document Format) – A fixed-layout format intended for visual presentation rather than structured analysis. A fixed layout means that the position of text, images, and other elements is preserved exactly as intended across devices and platforms, replicating the appearance of a printed page. This makes PDFs ideal for human reading and archival purposes but challenging for computational analysis. Extracting text from PDFs—especially those with poor OCR (Optical Character Recognition) quality—can be error-prone and often requires additional cleaning. Still, they remain a common format for digitized historical texts.
- R data files (.rds, .RData) – These are used to store R objects, such as data frames, in a way that preserves their structure and attributes. These are an efficient means of storage and retrieval within the R environment. When using packages like `hansardr`, the analyst loads .RData files.

Each of these formats has implications for how data can be queried, cleaned, stored, and interpreted and understanding these distinctions equips researchers to navigate the data ecosystem more effectively.

Data Type

"Data type" refers to the kind of value a variable can hold in R, such as numbers, text, or logical values. A data type refers to the kind of individual data value — its atomic unit. It defines a given piece of data is – whether it can be added or subtracted, whether it has a TRUE/FALSE value, and so on.

Data types determine how an object should be interpreted and used by R. Data types may differ from how values appear to human analysts.

Consider, for example, the following code, where the value "42" is enclosed in quotation marks, which tells R to treat it as a "string" or "character," not as a number (a "numeric" object).

```
my_number = "42"
typeof(my_number)
```

```
## [1] "character"
```

Because the the variable my_number is a character-type object, not a number-type object, we cannot add it to another number without producing an error.

```
my_number + 2
```

```
## Error in my_number + 2: non-numeric argument to binary operator
```

However, we can use the base R function `as.numeric()` to tell R to treat my_number henceforth as a number, not a string.

```
my_number <- as.numeric(my_number)
my_number + 2
```

```
## [1] 44
```

The functions `as.character()`, `as.date()`, and `as.numeric()` are all useful for converting datatypes from one type to the next. When working with `hansardr`, some of us have struggled with deep confusion in our inability to work with the year, until we realized that at some point the year had been converted to a string object, not a numeric or date object.

Analysts can check the type of data stored in a column using the `typeof()` function. For example, the following code returns "character" as the type for the "text" column in Hansard:

```
library("hansardr")

data("hansard_1800")

typeof("hansard_1800$text")
```

```
## [1] "character"
```

Throughout this book, we have primarily worked with character and integer data. For other types of historical analyses, however, we may encounter data types or values we have not addressed in detail. These are some common data types in R:

Data Type	Description	Example Code	`typeof()` Output
`numeric`	Decimal numbers (default type for numbers with decimals)	`x <- 3.14`	`"double"`

Data Type	Description	Example Code	`typeof()` Output
`integer`	Whole numbers	`y <- 42`	`"integer"`
`character`	Text or string values	`z <- "Mr. Gladstone"`	`"character"`
`logical`	Boolean TRUE/FALSE values	`a <- TRUE`	`"logical"`
`factor`	Categorical data stored as integers with labels	`f <- factor(c("low", "high"))`	`"integer"`

The meta-awareness that an analyst needs to become fluent with data types is not that different from an appreciation of the work of different media in representation. Consider René Magritte's famous 1929 painting *The Treachery of Images*, which depicts a pipe with the caption "Ceci n'est pas une pipe"—French for "This is not a pipe."

When observing this image, the viewer sees a pipe, despite the instructions emblazoned below the image. If an analyst stares at this image and puzzles for hours, they may still see just a pipe.

The caption serves to remind us that a representation is an object, different in its nature from the original object represented; the pipe that Magritte held in his hand and painted is different from the picture of the pipe he made.

Just so, a human analyst may stare at their code for hours while asking themselves why two seemingly identical objects are behaving differently once they run their code. The painting draws our attention away from its content (a pipe) back to the pieces making up the content (oil and canvas). Programming, too, asks us to practice shifting our attention away from the content of our constructions ("42") and onto the components that make up our constructions (the rules that govern data types within a programing language).

When Data Type is Missing: NA and NULL Values

Not all values have a data type. Analysts delving deeper into programming may encounter values like `NA` and `NULL` often appear in datasets with missing records. Engaging datasets with these values can be confusing because they analyst cannot treat them the same as data that have a type, like character or integer data.

In R, `NA` (Not Available) represents missing or undefined data. It used when a value is expected but not present, much like a blank cell in a spreadsheet.

In contrast, `NULL` indicates the absence of an object altogether, meaning the data structure or value does not exist at all.

Here is just one example of how NA's are treated differently than character values. This is an illustrative, fake historical snippet of a dataset of individuals recorded in an 1850 census table or notarial register, where some people have a listed occupation while others have a blank entry (NA). You want to count how many people were "laborers." If you treat NA as a string instead of a missing value, your counts will be wrong, because NA is the absence of information, not the word "laborer" misspelled or omitted.

The "occupation" column (which records the species' conservation status—e.g., endangered, threatened, or of least concern) contains character strings as well as NA values. As we will demonstrate, missing values (NA) behave yield a different result than text when undergoing string operations.

```
library("tidyverse")

census <- tibble(
  name = c("Mary Jenkins",
           "Samuel Price",
           "Hannah Lee",
           "Charles Morton",
           "Unnamed Child"),
  occupation = c("washerwoman",
                 NA,
                 "seamstress",
                 "laborer",
                 NA),
  age = c(32,
          44,
          29,
          51,
          1))
```

```
census
```

```
## # A tibble: 5 x 3
##   name           occupation    age
##   <chr>          <chr>       <dbl>
## 1 Mary Jenkins   washerwoman    32
## 2 Samuel Price   <NA>           44
## 3 Hannah Lee     seamstress     29
## 4 Charles Morton laborer        51
## 5 Unnamed Child  <NA>            1
```

From printing just the first rows, multiple NAs appear.

We can now process the dataset and observe what happens when we try to implement functions that expect a string on these rows. Here, we use str_detect() to identify whether or not a value is equal to the word "domesticated." We return our results in a new column named "is_washerwoman."

```
census %>%
  mutate(is_washerwoman = str_detect(occupation, "washerwoman"))
```

```
## # A tibble: 5 x 4
##   name           occupation    age is_washerwoman
##   <chr>          <chr>       <dbl> <lgl>
## 1 Mary Jenkins   washerwoman    32 TRUE
## 2 Samuel Price   <NA>           44 NA
```

```
## 3 Hannah Lee     seamstress     29 FALSE
## 4 Charles Morton laborer        51 FALSE
## 5 Unnamed Child  <NA>            1 NA
```

`str_detect()` returned `TRUE` if the value in the row is equal to "domesticated" and `FALSE` if it is not. However, multiple rows are left with `NA`, even though the values of those rows were not equal to "domesticated." This is because `NA` is a different data type than a string, and so `str_detect()`, a function that anticipates a string, did not assign a true or false value.

We must use a different approach to identifying `NA` values:

```
census %>%
  mutate(is_missing = is.na(occupation))
```

```
## # A tibble: 5 x 4
##   name           occupation    age is_missing
##   <chr>          <chr>       <dbl> <lgl>
## 1 Mary Jenkins   washerwoman    32 FALSE
## 2 Samuel Price   <NA>           44 TRUE
## 3 Hannah Lee     seamstress     29 FALSE
## 4 Charles Morton laborer        51 FALSE
## 5 Unnamed Child  <NA>            1 TRUE
```

If we wish for the `NA` values to be treated like strings–which could be the case if we are analyzing historical records where some individuals' occupations are missing, and we want to treat "unknown" as a searchable category–we could replace them with the word "missing" or with an empty string (`""`). The following code provides an example for how this can be accomplished:

```
library("tidyverse")

# Replace NA in ""conservation" with the string "missing"
census <- census %>%
  mutate(occupation = replace_na(occupation, "missing"))

# View the updated data
census %>%
  slice(1:10)
```

```
## # A tibble: 5 x 3
##   name           occupation    age
##   <chr>          <chr>       <dbl>
## 1 Mary Jenkins   washerwoman    32
## 2 Samuel Price   missing        44
## 3 Hannah Lee     seamstress     29
## 4 Charles Morton laborer        51
## 5 Unnamed Child  missing         1
```

Mixed Methods: Lists as Data Type

Lists are an important data type in R that also function as a data structure (defined in the following section). Lists are more complex than other types because they can include multiple data types within them.

Here, we define a list of continent names followed by their approximate population in millions.

```
continents_population <- list('South America', 644.54, 'Africa', 1305, 'Europe',
                              745.64, 'Asia', 4587)
```

```
typeof(continents_population)
```

```
## [1] "list"
```

`continents_population` is the list type. But each item inside the list also has its own type. To see this, we can loop through the list and print each item's type like so:

```
for (item in continents_population) {
  print(typeof(item))}
```

```
## [1] "character"
## [1] "double"
## [1] "character"
## [1] "double"
## [1] "character"
## [1] "double"
## [1] "character"
## [1] "double"
```

Lists can contain varying types, which makes them well-suited for representing complex data. They can provide a more memory-efficient alternative to data frames, particularly in applications such as text mining large datasets. However, the flexibility of lists introduces additional complexity in terms of data types. If these complexities are not carefully managed, code may execute without error but yield unintended results due to discrepancies in type.

Data Structure

A data structure is a way of organizing and storing data so it can be accessed and modified. If a data type defines what a given piece of data is, a data structure is a container — it organizes multiple values (often of different types) in a structured way. It defines how data is stored, related, and accessed.

Throughout this book, we have consistently employed data frames, and more specifically tibbles, due to their easy integration with the `tidyverse` and their predictable behavior across a wide range of operations beyond the `tidyverse`. We have used intuitive naming conventions and pipeable workflows – all of which are customary for the main software package we have used, the `tidyverse`.

The deliberate choice of using the `tidyverse` methods – and more specifically the dityverse data structure of tibbles =– for teaching minimizes the complications often introduced by less interoperable or idiosyncratic data structures.

But when an analyst begins text mining in earnest, they may soon venture into engaging other software packages in R that may be useful for text mining, but which use different data structures that can be confusing. As with data types, data structures determine the ways in which data can be accessed, transformed, and analyzed in R – so non-tidy formats may appear confusing and difficult-to-access to the analyst familiar only with the tools from the `tidyverse`.

For instance, the `quanteda` software package exports many outputs as S3 objects with custom print methods and slot-like components. The tm and topicmodels package export their outputs as S4 objects, some of which require knowledge of package-specific functions to extract results.

Common R Data Structures in Text Mining

Structure	Dimensions	Homogeneous	Key Use for Text Mining
Matrix	2D	Yes	Numeric structure for word embeddings or term-document co
List	1D	No	Flexible storage for mixed or uniform data—e.g., texts, dates
Data Frame	2D	No	Standard tabular format for digitized records or structured te
Tibble	2D	No	Tidyverse-friendly data frames optimized for large data and p
S3 Object	Varies	No	Used in packages like `quanteda`, `stm` with custom print/acces
S4 Object	Varies	No	Formal, slot-based system used in `tm`, `topicmodels`, with acc

Some common data structures in R that we have used for text mining are represented by the following table.

The solution to working with new data structures is to learn how to translate a new data structure into a format the analyst is familiar with.

So the goal is to: extract what you need → eeshape it → coerce it into a tibble

Working with the Output of Other R-Based Software: `Quanteda`

However, there are cases where a data frame may not be the most appropriate or efficient data structure. For example, when working with the `quanteda` package (such as when we performed Key Words in Context (KWIC), we rely on its custom-built data structure known as a `quanteda` "tokens object." The tokens object was specifically designed to optimize performance when handling large-scale text data within the `quanteda` ecosystem. In fact, it would be very difficult to perform some of the analyses we did without invoking the `quanteda` ecosystem given how efficient `quanteda` is at processing.

```
library("quanteda")
```

```
## Package version: 4.3.1
## Unicode version: 14.0
## ICU version: 70.1
```

```
## Parallel computing: disabled
```

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

```
library("tidyverse")
library("hansardr")

data("hansard_1800")

texts <- hansard_1800$text

# Create a tokens object from the text column
tokens_hansard <- tokens(texts)

# Perform a KWIC search on the tokens object
kwic_result <- kwic(tokens_hansard, pattern = "corn", window = 5)
```

docname	from	to	pre	keyword	post
text560	34	34	to the practice of hoarding	corn	in this of scarcity ,
text1124	23	23	paid to foreign countries for	corn	and other articles of provision
text1283	58	58	Mutiny Bills , the Seed	Corn	Exportation , the Greenland Whal
text2284	11	11	and encouraging the importation of	corn	the act for regulating the
text2358	33	33	all the last regulations respecting	corn	, permitting the importation ,
text3349	3	3	The Irish	Corn	, Potatoe , and Provision

Impressively, the results from running `KWIC()` on an entire decade of the Hansard debates are now ready to analyze.

```
gt(head(kwic_result))
```

However, attempting to use a `tidyverse` function on the `quanteda` tokens object returns an error.

```
slice(tokens_hansard, 1:5)
```

```
## Error in UseMethod("slice"): no applicable method for 'slice' applied to an object of class "tokens"
```

While `quanteda`'s design improves efficiency, it also introduces certain limitations. In particular, a `quanteda` tokens object cannot be passed directly to most base R or `tidyverse` functions. Instead, handling a `quanteda` obejct largely restricts us to using `quanteda` functions–or other functions explicitly designed to interface with this structure–unless we first convert the corpus back into a data frame.

To use `tidyverse` functions on the data we processed using `quanteda`, we must convert the data (in this case, the tokens object) to a tibble or data frame.

```
tokens_tibble <- tibble(doc_id = rep(names(tokens_hansard), lengths(tokens_hansard)),
                        token = unlist(tokens_hansard, use.names = FALSE))
```

Now we can use `tidyverse` functions freely.

```
slice(tokens_tibble, 1:5)
```

```
## # A tibble: 5 x 2
##   doc_id token
##   <chr>  <chr>
## 1 text1  moved
## 2 text1  that
## 3 text1  Lord
## 4 text1  Walsingham
## 5 text1  be
```

Working With Packages Beyond `Quanteda`

Wrangling diverse data structures into the tidy format can be frustrating, but learning the proprietary functions for each individual software package can take a time. Each analyst will have to balance their own level of interest and comfort as they navigate between different formats of data. In general, however, the

Quanteda (tokens, dfm, etc.)

Task	What to do
Convert dfm to tibble	‘dfm %>% convert(to = ”data.frame”) %>% as_tibble()‘
Get token list	‘tokens %>% as.list()‘
Combine with metadata	‘docvars(dfm)‘ gives doc-level covariates
Tidy version	‘Use quanteda.textmodels::convert()‘ or ‘tidytext::tidy()‘ if applicable

tm (Corpus, DocumentTermMatrix)

Task	What to do
Convert Corpus	‘sapply(corpus, as.character)‘ or ‘content(corpus)‘
DTM to tidy format	‘tidy(DTM)‘ from ‘tidytext‘
Combine with meta	‘meta(corpus)‘ or ‘tm_map()‘

strategy for the analyst who has learned tidy methods of working with data remains the same: wrangle the data into the tidy format, then work with it according to methods you already understand.

Here is a guide for extracting data from the most common text-based software packages in R and their outputs.

Helpful Tidyverse-Compatible Tools	
Tool/Package	Use
tidytext	Makes many text objects tidy (DTMs, LDA, etc.)
textrecipes	Tidy preprocessing (e.g., tokenization, tf-idf)
tidylo	Additional tidy tools for text and lexicons
textdata	Provides external lexicons and datasets
unnest_tokens()	Tokenization (splitting text into units, e.g., words or n-grams)
broom / broom.mixed	Tidy output from models (including ‘topicmodels‘)

Conclusion

No single book can prepare the analyst for all possible variations of data they may encounter, but the major kinds of data used as of the writing of this book are encompassed by the the methods we have set out here – using API calls and RSS feeds, bulk downloads, and pdf’s; cleaning data, and paying attention to data type and structure. With practice, a data analyst may soon acquire the fluency with these methods needed to wrangle data from across the web and to engage with multiple software packages, each of which have their strengths and weaknesses.

text2vec (itoken, dtm, etc.)

Task	What to do
Convert DTM	‘as.matrix(dtm) %>% as_tibble()‘ or ‘Matrix::summary(dtm)‘
Vocabulary to tibble	‘vocab$term_stats %>% as_tibble()‘
Embeddings to tibble	‘as_tibble(embeddings)‘

topicmodels / stm (LDA/STMs)

Task	What to do
Tidy topics + terms	‘tidy(model, matrix = ”beta”)‘ (from ‘broom‘)
Tidy docs + topics	‘tidy(model, matrix = ”gamma”)‘
STM effects	‘Use estimateEffect()‘ then extract manually

Sparse matrices (e.g., dgCMatrix)

Task	What to do
Convert to dense	‘as.matrix(sparse_mat)‘
Extract non-zeros	‘Matrix::summary(sparse_mat)‘
Tidy-friendly format	‘Use reshape2::melt()‘ or ‘pivot_longer()‘