Chapter 7: Data

We contend with Ian Milligan that the 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 (2019). This chapter translates that argument into method by offering techniques to meet this growing digitally-born archive.

In this chapter, we focus on foundational concepts behind dataset creation and cleaning, treating the Internet as a broader data ecosystem in which analysts can delight (Dobson 2019; Hayler et al. 2016; Hai-Jew 2018). In doing so, we aim to equip readers with foundational competencies that move beyond the use of one-off tools for isolated case studies. Instead, we advocate for methodological approaches that are transferable across diverse areas of historical inquiry, thereby supporting more sustained and rigorous engagements with the past.

This chapter introduces key technical concepts—such as data formats (how data is stored), data types (what kinds of data are represented), and data structures (how data is organized for use and analysis)—to support readers in working with a wide variety of datasets. These concepts are meant to be broadly applicable across historical research, including traditional historical scholarship, cultural heritage studies, public history, and the digital humanities.

To illustrate this broader applicability, we move beyond the 19th-century Hansard corpus, which has been the primary focus of earlier chapters. The Hansard debates allowed us to demonstrate how sustained engagement with a single dataset can yield multiple, nuanced interpretations (Guldi 2018), especially since the corpus supports both longue durée and close readings.

In contrast, this chapter works with multiple, disparate datasets to show how historians can engage with diverse online archives—many of which are increasingly born-digital—and how analytical approaches may shift depending on the dataset's type, format, and structure.

Our hope is to guide readers' imaginations without limiting them. This is why we have framed this book around "text mining for historical analysis" in a broad sense, while still grounding our examples in specific use cases.

The chapter is organized into three major sections that guide readers through the process of transforming historical text into analyzable data. These sections are:

- Creating Datasets
- Cleaning Datasets
- Interpreting Data Format, Type, and Structure

Creating Datasets introduces the analyst to a larger data ecosystem, beyond the prepackaged `hansardr` data prepared for this book. It covers concepts and technologies such as: (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. We decided to cover these concepts because historians working with digital records must often navigate different and even fragmented infrastructures. 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. Yet, it is ideal to be able to engage these different infrastructures under any circumstance. The ability to engage these fragmented infrastructures, we suggest, may be key to challenging the prevailing reliance on canonical texts and curated corpora, as solely engaging these risks reinforcing narrow or dominant histories. Archives are not themselves transparent sources (Bode 2018).

Bulk downloads from institutional repositories are a common alternative to collecting data through a web service like an API. 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. In this chapter we demonstrate ways to engage nested structures and prepare them for analysis. It is also important to consider that 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.

In cases where support for data collection is limited, web scraping may be necessary to compile datasets—particularly when bulk download options are unavailable or incomplete. In R, this typically involves using packages such as `rvest` or `httr` to extract data from HTML pages (Wickham 2022, 2023). Traditionally, successful web scraping has required a working knowledge of HTML structure, including elements like tags, classes, and IDs. However, with the advent of generative AI tools, engaging with these technical components has become more accessible. For instance, users can now describe their goals in natural language. In this chapter, we will demonstrate this approach. We will also remind analysts about the ethical and legal considerations involved in automated data collection. These include respecting robots directives, which are rules outlined by a website that indicate which parts of the site automated agents (like web scrapers or search engine bots) are allowed or disallowed from accessing. Ignoring these directives may violate a site's terms of service and can result in access being blocked. Finally, we will briefly discuss licensing issues and Terms of Service agreements, noting that this overview is intended as a starting point for understanding the greater data ecosystem, and not a substitute for professional legal advice.

Across these modes of access, we encourage analysts to think critically about how the technical affordances of each approach shape the boundaries of the archive.

The Data Cleaning section addresses the challenges of working with messy or inconsistent data drawn from data archives, emphasizing the interpretive decisions required before meaningful analysis can occur. As established earlier in the book, data processing is not a neutral or purely technical task; it constitutes a form of historical interpretation. Analysts must make informed choices about how to handle Optical Character Recognition (OCR) errors, whether to standardize or retain historical spelling variations, and how to manage ambiguous or missing data, all of which can shape the outcomes and implications of subsequent analysis.

For example, OCR errors are common in digitized print materials, particularly for older typefaces or poor-quality scans. Removing or correcting these errors can improve analytical precision, but it can also obscure the lineage and transmission of the data over time, which in itself tells a history. As W. W. Greg contended in 1932, analysis "may just as rightly be applied to any other point in the transmission of the text." For Greg, our concern should be "the whole history of the text" treating it if though it were a "living organizism which in its descent through the ages, while it departs more and more from the form impressed upon it by its original author, exerts, through its imperfections as much as through its perfections, its own influence." Similarly, choices about whether to standardize spelling across a corpus (like standardizing "honour" and "honor") entail trade-offs between comparative accuracy and preserving historical variation.

Ambiguous or incomplete information–such as unclear personal names, uncertain dates, or entirely missing entries–must also be considered during the data cleaning process. Decisions about how to address such gaps are, in their own right, and act of interpretation and have impact on how we engage the historical record.

The final major section, Interpreting Data Format, Type, and Structure, introduces key technical concepts that are often overlooked in traditional humanities research yet are essential for engaging meaningfully with disparate and broad digital data. Humanities scholarship is frequently constrained by limited engagement with the technical dimensions of data work. Yet, having a conceptual understanding of these technical foundations–such as understanding data formats, data types, and data structures–makes it more feasible for researchers to work with large-scale, textual datasets hosted anywhere, of any kind. These technical

dimensions of data shape research questions and analytical methods.

Data formats such as plain text, XML, JSON, and CSV each come with affordances and limitations. XML, for instance, is useful when working with deeply nested or hierarchical information, such as parliamentary proceedings or encoded manuscripts with layered metadata. JSON is often used for web-based data or APIs and can represent complex nested structures efficiently. CSV files, by contrast, are flat and best suited for spreadsheet-style data, making them easy to use with statistical tools but less effective for representing rich metadata or relationships between documents.

Equally important is an awareness of data types—such as textual, numerical, categorical, and temporal—and how each is handled within a coding environment. While the paradigm of text mining set forth in *Text Mining for Historical Analysis* largely automates the process of recognizing and processing different data types, this level of automation is not consistent across all datasets or programming languages. For example, importing a dataset might result in interpreting all date fields as character strings unless directly converted, and categorical variables–such as political party affiliation, region, or document type–may need to be manually encoded as characters to be used appropriately in analysis. Attempting to perform operations on data of unexpected data types can result in inaccurate results or errors in the programming logic.

Data structure refers to the way information is organized and stored within a programming environment, which in turn shapes how it can be accessed, processed, and analyzed. In *Text Mining for Historical Analysis*, we primarily worked with data frames–specifically, tibbles, a tidyverse variant of the data frame optimized for compatibility with other tidyverse tools (Wickham 2019). We also created character strings, or sequences of text that represent words, sentences, or larger textual units. However, many other data structures may be used for text analysis. The `quanteda` package, as one example, provides specialized data structures such as the "corpus object" and the "document-feature matrix" (Benoit 2018) These structures are optimized for large-scale text analysis but are not always interoperable with other packages or approaches. Recognizing the qualities of different data structures is important for selecting appropriate techniques and ensuring analytical flexibility.

As a final but not exhaustive example of a data structure, some data may be stored as lists, which are like highly flexible containers capable of holding elements of varying types and lengths, including vectors, data frames, and even other lists. In earlier chapters, we used lists for relatively simple tasks, such as storing sets of keywords, but we did not examine their structure or capabilities in much detail. This chapter turns more directly to that task, demonstrating how lists can be organized, indexed, and integrated into more complex analytical workflows with the aim of providing historians with concepts needed to engage more fully with the interpretive dimensions of computational text analysis.

Taken together, these sections provide perspective to work with historical data in a digital age.

Creating Datasets

Throughout *Text Mining for Historical Analysis*, we have relied on a curated dataset developed specifically for this book, `hansardr` (Buongiorno 2021). However, in many research contexts, data will not be pre-curated or neatly structured in advance. This is especially true for analysts pursuing original research questions, where relevant data must often be located, extracted, and prepared independently before analysis can begin. Such is frequently the case for analysts who have compiled their own archives or have collected data from primary sources. 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. Such a conceptual framework may be especially important for historical research, where relevant sources may be scattered across different archives and exist only in incomplete or inconsistent formats. While historians have long engaged in the manual creation of datasets, this process can be time-consuming and resource-intensive, often requiring substantial investments of both labor and funding. In such cases, automation—or partial automation—may be a practical and necessary strategy for assembling and managing research data at a large scale.

This section demonstrates three ways of creating a dataset:

- By accessing an API endpoint
- By performing bulk downloads
- By scraping websites

Creating a Dataset from an API Endpoint

An API, or Application Programming Interface, 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

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.

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`.

However, using this 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`.

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.

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. Below we provide an overview of these considerations.

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.

To illustrate how an API works, consider this example from the City of Dallas Open Data portal, which provides access to data from the Dallas Animal Shelter for the 2019–2020 fiscal year:

`https://www.dallasopendata.com/Services/Dallas-Animal-Shelter-Data-Fiscal-Year-2019-2020/7h2m-3um5/abou`

Copying and pasting the API endpoint from this page,

`https://www.dallasopendata.com/resource/7h2m-3um5.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. The dataset includes information about shelter operations, such as animal intakes, adoptions, transfers to rescue organizations, and the daily care provided by shelter staff. 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 `GET()` function from the `httr` package sends an HTTP `GET` request to the specified API endpoint to retrieve data. 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 animal shelter data). The `status_code()` function checks whether the request was successful. If it was, `content()` extracts the raw JSON text from the response body. This text is then passed to `fromJSON()` from `jsonlite`, which parses the JSON into 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
# load libraries for data wrangling (tidyverse), making http requests (httr),
# and parsing json content into data frames (jsonlite)
library(tidyverse)
library(httr)
library(jsonlite)

# define the api endpoint that provides animal shelter data in json format
url <- "https://www.dallasopendata.com/resource/7h2m-3um5.json"  # api url

# make a get request to the api endpoint and store the response
response <- GET(url)  # send http get request

# check if the response from the server was successful (status code 200)
# if successful, extract the json content as a text string
# parse the json text into a flat data frame (tibble) for easier analysis
# if the request failed, halt execution and show the returned status code
if (status_code(response) == 200) {
  json_content <- content(response, "text", encoding = "UTF-8")  # get content

  animal_shelter_data <- fromJSON(json_content, flatten = TRUE) %>%  # parse json
    as_tibble()  # convert to tibble
  } else {
  stop("Failed to retrieve data. Status code: ", status_code(response)) } # error handling
```

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

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

```
## # A tibble: 6 x 5
##   animal_id animal_type animal_breed  kennel_number kennel_status
##   <chr>     <chr>       <chr>         <chr>         <chr>
## 1 A1093136  CAT         DOMESTIC SH   K01           IMPOUNDED
## 2 A1046046  DOG         PIT BULL      AD 085        UNAVAILABLE
## 3 A1098758  BIRD        HAWK          RECEIVING     UNAVAILABLE
## 4 A1061310  DOG         LABRADOR RETR LFD 163       UNAVAILABLE
## 5 A1091970  CAT         DOMESTIC SH   CC 25         UNAVAILABLE
## 6 A1091007  DOG         CHIHUAHUA SH  INJD 015      UNAVAILABLE
```

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. This API returned data in JSON, 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. Below is an example of what that JSON data looks like, as shown below. Clicking on the `json_content` data i n the Global Environment panel will also provide the means to explore the JSON data.


```
## [
##     {
##         "animal_id": "A1093136",
##         "animal_type": "CAT",
##         "animal_breed": "DOMESTIC SH",
##         "kennel_number": "K01",
##         "kennel_status": "IMPOUNDED",
##         "activity_sequence": "1",
##         "source_id": "P0915082",
##         "census_tract": "005200",
##         "council_district": "1",
##         "intake_type": "STRAY",
##         "intake_subtype": "AT LARGE",
##         "intake_total": "1",
##         "reason": "OTHRINTAKS",
##         "staff_id": "CBU",
##         "intake_date": "2020-01-12T00:00:00.000",
##         "intake_time": "11:37:00",
##         "due_out": "2020-01-12T00:00:00.000",
##         "intake_condition": "APP SICK",
##         "hold_request": "MEDICAL",
##         "outcome_type": "FOSTER",
##         "outcome_subtype": "TREATMENT",
##         "outcome_date": "2020-01-12T00:00:00.000",
##         "outcome_time": "14:46:00",
##         "impound_number": "K20-494003",
##         "outcome_condition": "APP SICK",
##         "chip_status": "SCAN NO CHIP",
##         "animal_origin": "OVER THE COUNTER",
##         "month": "JAN.2020",
##         "year": "FY2020"
##     },
```

```
##     {
##         "animal_id": "A1046046",
##         "animal_type": "DOG",
##         "animal_breed": "PIT BULL",
##         "kennel_number": "AD 085",
##         "kennel_status": "UNAVAILABLE",
##         "activity_sequence": "1",
##         "source_id": "P0740141",
##         "census_tract": "012900",
##         "council_district": "9",
##         "intake_type": "STRAY",
##         "intake_subtype": "CONFINED",
##         "intake_total": "1",
##         "reason": "OTHRINTAKS",
##         "staff_id": "AR1758",
##         "intake_date": "2019-11-28T00:00:00.000",
##         "intake_time": "12:52:00",
##         "due_out": "2019-12-05T00:00:00.000",
##         "intake_condition": "APP WNL",
##         "hold_request": "ADOP RESCU",
##         "outcome_type": "ADOPTION",
##         "outcome_subtype": "WALK IN",
##         "outcome_date": "2019-12-19T00:00:00.000",
##         "outcome_time": "14:56:00",
##         "impound_number": "K19-489088",
##         "outcome_condition": "APP WNL",
##         "chip_status": "SCAN CHIP",
##         "animal_origin": "FIELD",
##         "month": "NOV.2019",
##         "year": "FY2020",
##         "activity_number": "A19-203263",
##         "receipt_number": "R19-561166",
##         "additional_information": "ADOPTION"
##     }
## ]
```

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).

```
# return the number of rows (records) in the json list
nrow(json_list)
```

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

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

https://www.dallasopendata.com/Services/Dallas-Animal-Shelter-Data-Fiscal-Year-2019-2020/7h2m-3um5/about_data —

We can see that the site reports the dataset contains tens of thousands 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://www.dallasopendata.com/resource/7h2m-3um5.json"

# pagination settings
limit <- 1000
# will store the output from each iteration
animal_shelter_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

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

if (status_code(response) != 200) { # check if request failed
  stop("Request failed. Status code: ", status_code(response)) } # stop execution with error

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

  animal_shelter_data <- bind_rows(animal_shelter_data, as_tibble(data_subset)) } # save collected data
```

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.

```
## Warning: package 'usdoj' was built under R version 4.5.1
```

```r
# set custom repo for ropengov packages
options(repos = c(ropengov = 'https://ropengov.r-universe.dev'))

# install the usdoj package from ropengov repository
install.packages("usdoj")

# load the usdoj package into the session
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.

```r
# return 10 U.S. department of justice blog entries
blog_entries <- doj_blog_entries(n_results = 10, search_direction = "DESC")
```

```r
head(blog_entries$teaser, 5)
```

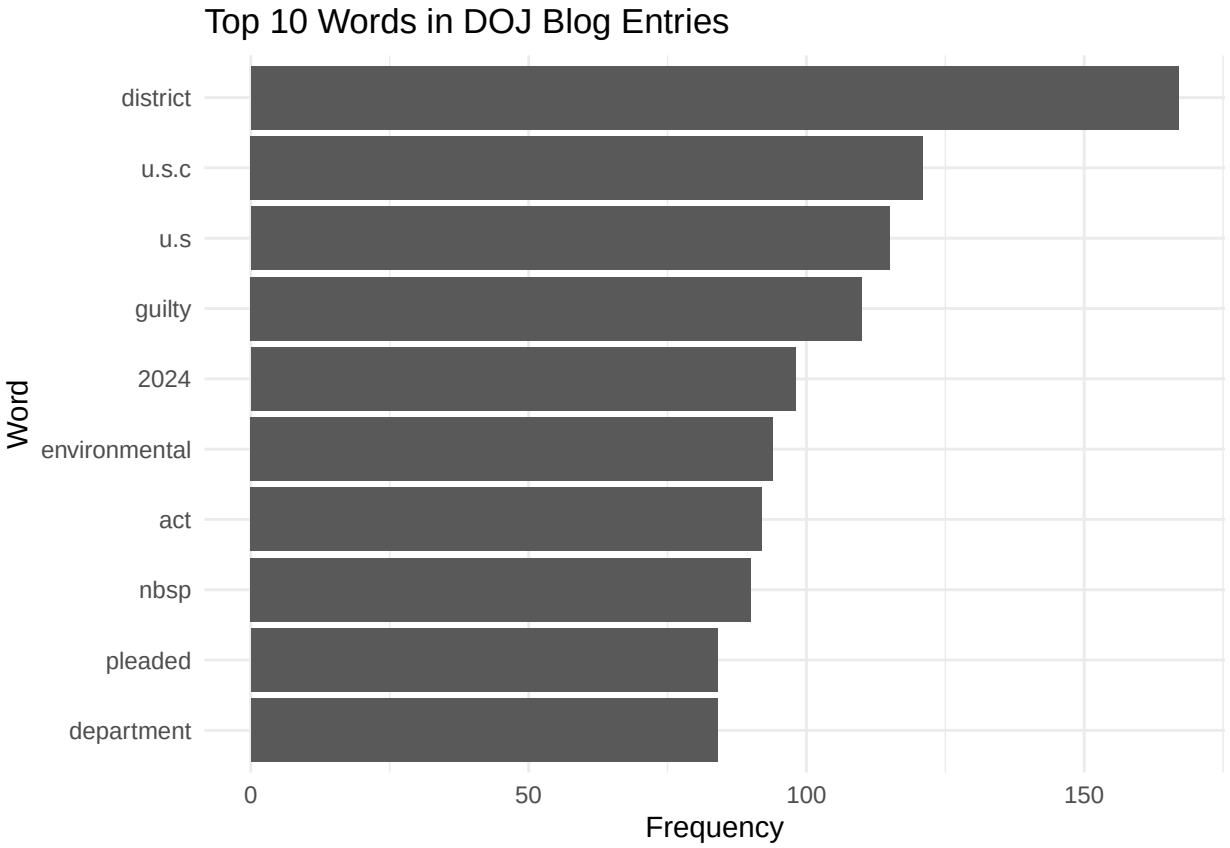
```
## [1] "December's cases include violations of the Clean Air Act; Conspiracy; Tampering with a Monitori
## [2] "In 2024, the United States Trustee Program appointed 35 private trustees to serve in cases unde
## [3] "Our Combating Redlining Initiative, launched in 2021, has made transformative progress in
## [4] "December's cases include violations of the Clean Air Act; Conspiracy; Tampering with a Monitori
## [5] "The U.S. Trustee Program has revamped its Section 341 Meeting of Creditors webpage and released
```

From this point, it is easy to analyze words.

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

# tokenize, remove stop words, and count word frequency
top_words <- blog_entries %>%
  select(body) %>% # keep only body text
  unnest_tokens(word, body) %>% # split into individual words
  anti_join(stop_words, by = "word") %>% # remove stop words
  count(word, sort = TRUE) %>% # count word frequency
  slice_max(n, n = 10) # keep top 10 words

# plot the top words
ggplot(top_words, aes(x = reorder(word, n), y = n)) + # reorder for plotting
  geom_bar(stat = "identity") + # draw bars
  coord_flip() + # flip axes for readability
  labs(title = "Top 10 Words in DOJ Blog Entries", # add title
       x = "Word", # label the x axis
       y = "Frequency") + # label the y axis
  theme_minimal() # apply clean theme
```



Bulk Downloads

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. 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. However, a notable limitation 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.

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 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 is 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).

Importantly, while bulk data downloads seem straightforward, they can be difficult to work with in practice. 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.

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.

```
library(data.table)

# 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)) { # loop through file in chunks
  chunk <- fread(file_path, nrows = chunk_size, skip = start_row) # read chunk

  # 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

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. However, it is important to note that while the code handles downloading and text extraction, 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)
```

```
## Warning: package 'pdftools' was built under R version 4.5.1
```

```
## Using poppler version 25.05.0
```

```
library(tidyverse)

# 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: 2025-07-26 00:32
##   Status: 200
##   Content-Type: application/pdf
##   Size: 492 kB
## <ON DISK>  C:\Users\steph\OneDrive\Desktop\Text Mining for Historical Analysis\chapter_9_data\tmha_da
```

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

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

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.

```
head(tidy_egils_saga)
```

```
## # A tibble: 6 x 2
##    page text
##   <int> <chr>
## 1     1 "Egil's Saga\n"
## 2     2 "Egil's Saga\n1893 translation into English by W. C. Green from the ori~
## 3     3 "Chapter 3 - The beginning of the rule of Harold Fairhair.\nHarold, son~
## 4     4 "Vemund Audbjorn's brother still retained the Firthfolk, being made kin~
## 5     5 "The messengers went away, and when they came to the king told him all ~
## 6     6 "nor rival thy betters; thou wilt not, I am sure, yield to others overm~
```

The PDF contains *Egil's Saga*, a famous work of medieval Icelandic literature. It is believed to have been written in the 13th century, most likely by the historian and poet Snorri Sturluson. The saga recounts the life of Egil Skallagrímsson, a 10th-century Icelandic warrior, poet, and farmer. Spanning multiple generations, the narrative begins with Egil's ancestors and extends through his descendants, situating his personal story within a broader historical and familial context. Central themes include feuds, honor, and legal disputes, along with encounters between Egil and various Norwegian kings.

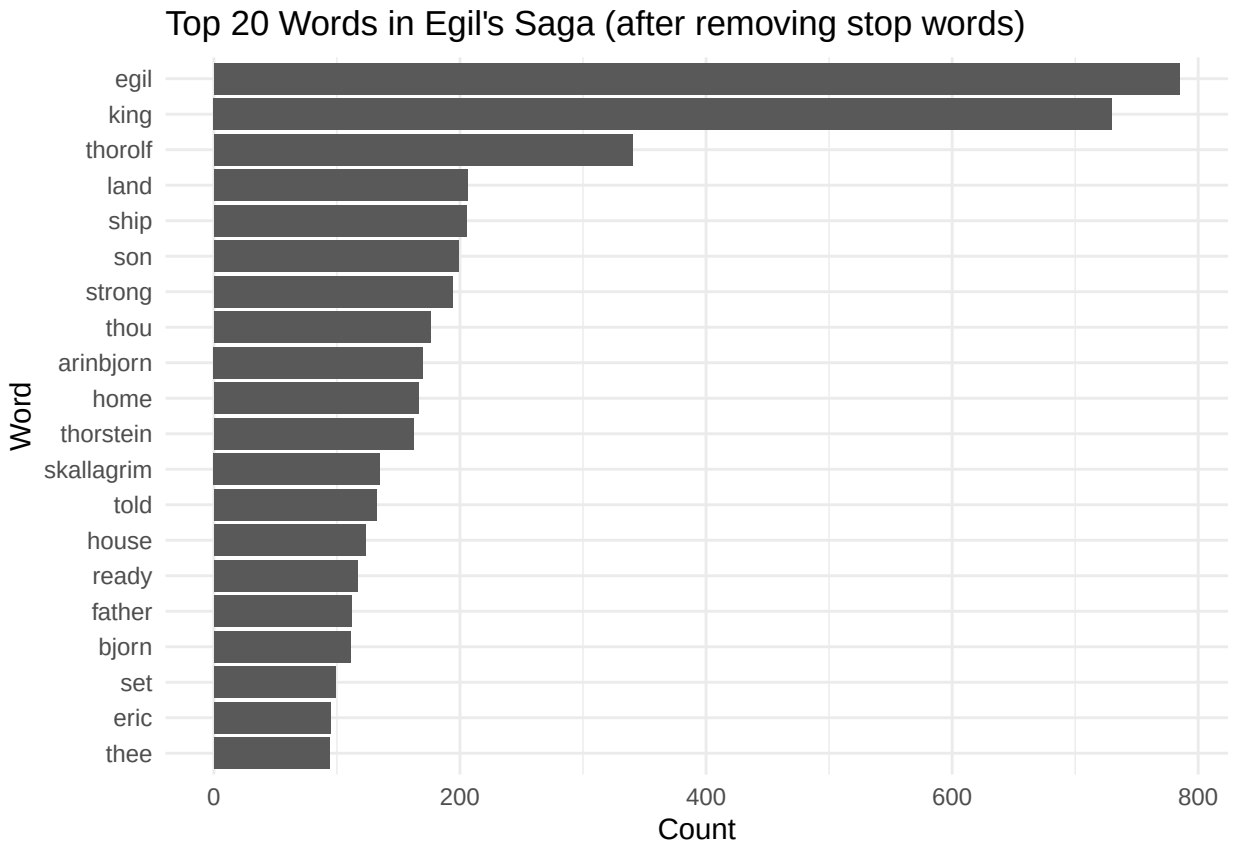
With just a few lines of code to locate, download, and scrape the PDF file, it is now easy to count the top words in *Egil's Saga*.

```
library(tidyverse)
library(tidytext)

# load the build-in stop words from tidytext
data(stop_words)

# unnest tokens, remove stop words, count words, and select the top 20
top_words <- tidy_egils_saga %>% # create a new dataset
  unnest_tokens(word, text) %>% # tokenize text
  anti_join(stop_words, by = "word") %>% # remove stop words
  count(word, sort = TRUE) %>% # count word frequency
  slice_max(n, n = 20) # keep top 20 words

# visualize top 20 words
ggplot(top_words,
       aes(x = reorder(word, n), y = n)) + # reorder words by count
  geom_col() + # draw bars
  coord_flip() + # flip axes
  labs(title = "Top 20 Words in Egil's Saga (after removing stop words)",  # add labels
       x = "Word", # label x axis
       y = "Count") + # label y axis
  theme_minimal() # apply minimal theme
```



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 %>% # create a new dataset
  mutate(author = "Snorri Sturluson", # add author name
         book = "Egil's Saga", # add book title
         date = "1893") # add publication date
```

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.

```
head(tidy_egils_saga_with_metadata)
```

```
## # A tibble: 6 x 5
##    page text                                                 author book  date
##   <int> <chr>                                                <chr>  <chr> <chr>
## 1     1 "Egil's Saga\n"                                      Snorr~ Egil~ 1893
## 2     2 "Egil's Saga\n1893 translation into English by W. C.~ Snorr~ Egil~ 1893
## 3     3 "Chapter 3 - The beginning of the rule of Harold Fai~ Snorr~ Egil~ 1893
## 4     4 "Vemund Audbjorn's brother still retained the Firthf~ Snorr~ Egil~ 1893
## 5     5 "The messengers went away, and when they came to the~ Snorr~ Egil~ 1893
## 6     6 "nor rival thy betters; thou wilt not, I am sure, yi~ Snorr~ Egil~ 1893
```

Web Scraping and RSS Feeds

Before the widespread availability of APIs and RSS feeds, data collection from websites was commonly performed through web scraping, which involves the extraction of information from the underlying HTML structure of web pages. To avoid overloading the server hosting the webpage, requests were typically rate-limited during this process.

In R, the rvest package facilitates this method by parsing HTML content into a structured representation, commonly referred to as the HTML tree. This tree reflects the nested hierarchy of HTML elements on a webpage, where each element—such as headers, paragraphs, and links—occupies a specific node within the document's structure.

Extraction of content from this structure can be accomplished using CSS selectors or XPath queries. 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).

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.

```
# load libraries
library(rvest) # for web scraping
```

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

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

```
library(dplyr) # for data manipulation

url <- "https://github.com/stephbuon/text-mining-for-historical-analysis"  # target url
page <- read_html(url)  # read html content of the page
```

```
## <!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">
```

However, in many cases, web scraping is no longer necessary. Numerous websites now provide structured data access through APIs or RSS feeds, which are more simpler alternatives 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.

```
# load libraries in R
library(rvest) # for scraping and html/xml parsing
library(tidyverse) # for data manipulation
```

```r
library(xml2) # for working with xml documents

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

rss_page <- read_xml(rss_url) # read and parse the rss xml

articles <- tibble(title = rss_page %>% # create a tibble with article info
                     xml_find_all("//item/title") %>% # extract titles
                     xml_text(),
                   link = rss_page %>%
                     xml_find_all("//item/link") %>% # extract links
                     xml_text())

print(head(articles, 5)) # show first 5 articles
```

```
## # A tibble: 5 x 2
##   title                                                                link
##   <chr>                                                                <chr>
## 1 More than 200 MPs call for Starmer to recognise Palestinian state    http~
## 2 Almost a third of people in Gaza not eating for days, UN food programme~ http~
## 3 'I'm so tired': Mother of starving Gazan baby speaks to BBC          http~
## 4 Gregg Wallace 'sorry' but says he's 'not a groper'                   http~
## 5 Trump praises PM as he arrives in Scotland for four-day trip         http~
```

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". 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 digital history, 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.

```
# load hansardr
library(hansardr)

# load the speaker metadata for just the 1840s
data("speaker_metadata_1840")
```

```
# view just the data for the specified sentence_id
speaker_metadata_1840 %>%
  filter(sentence_id == "S3V0051P0_728")
```

```
##     sentence_id                   speaker suggested_speaker ambiguous
## 1 S3V0051P0_728 The Bishop of Exeter said                           0
##   fuzzy_matched ignored
## 1             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."

```
# view just the data for the specified sentence_id
speaker_metadata_1840 %>%
  filter(sentence_id == "S3V0060P0_2194")
```

```
##      sentence_id      speaker         suggested_speaker ambiguous fuzzy_matched
## 1 S3V0060P0_2194 'Mr. Speaker charles_shaw-lefevre_3007         0             0
##   ignored
## 1       0
```

```
# view just the data for the specified sentence_id
speaker_metadata_1840 %>%
  filter(sentence_id == "S3V0051P0_18193")
```

```
##       sentence_id      speaker suggested_speaker ambiguous fuzzy_matched
## 1 S3V0051P0_18193 -Lord Ashley                            0             0
##   ignored
## 1       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."

```
# view just the data for the specified sentence_id
speaker_metadata_1840 %>%
  filter(sentence_id == "S3V0060P0_2194")
```

```
##      sentence_id      speaker         suggested_speaker ambiguous fuzzy_matched
## 1 S3V0060P0_2194 'Mr. Speaker charles_shaw-lefevre_3007         0             0
##   ignored
## 1       0
```

```
# view just the data for the specified sentence_id
speaker_metadata_1840 %>%
  filter(sentence_id == "S3V0060P0_2194") %>%
  mutate(speaker = str_replace_all(speaker, "'", ""))  # remove apostrophes from speaker names
```

```
##      sentence_id     speaker         suggested_speaker ambiguous fuzzy_matched
## 1 S3V0060P0_2194 Mr. Speaker charles_shaw-lefevre_3007         0             0
##   ignored
## 1       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.

```
library(stringr)

# Messy string
messy_string <- " <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("<[^>]+>") %>%  # remove html tags
  str_replace_all("[^\\w\\s]", "") %>% # remove punctuation/symbols
  str_replace_all("\\s+", " ") %>%  # collapse multiple spaces
  str_trim() # trim leading/trailing spaces

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:

```
library(stringr)

# Remove HTML tags
messy_string <- gsub("<[^>]+>", "", messy_string)

# Replace multiple punctuation marks with a space (put hyphen first or last to avoid range error)
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.

```
# For wrangling strings
library(stringr)

# loop through each row of the data frame multiple times (up to max_iterations)
# on each iteration, send the current text to an llm to get a suggested regex pattern
# apply that regex to further clean the text by replacing matches with a space
# update the text in the data frame with the cleaned result after each pass
max_iterations <- 5 # number of cleaning passes

for (i in 1:max_iterations) { # loop over iterations
  for (row in 1:nrow(df)) { # loop over each row of the data frame

    text <- df$text[row] # extract the current text

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

    text <- str_replace_all(text, llm_suggested_regex, " ") # apply regex to clean

    df$text[row] <- text } }# update data frame with cleaned 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.

Therefore, we offer a conceptual understanding of how data is structured, processed, and interpreted.

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. Data types determine how the data is stored and what operations can be performed on it. Developing a deeper understanding of data types requires a meta-awareness of the mode in which a value is represented. For example, in the following code, the value "42" is considered a string (or character) because it is enclosed in quotation marks, not a number Data types are a representation of the value that determine how it will be interpreted and used by R. Data types may differ from how values appear to human analysts.

```
# see the data type
typeof("42")
```

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

The idea that a meta-awareness is needed to interpret the representation of an object is not a new concept. It is illustrated in 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."

Yet, when observing this image, one might see a pipe. If an analyst stares at this image and puzzles for hours, they may still see just a pipe. But technically, as the quote on the bottom serves to remind us, representations are not the same as the thing they depict to an analyst. On a technical level, this is not a pipe. It is a painting of a pipe. Or mode technically correct here: it is a digital image of a painting of a pipe. Or, if you are reading the hard copy version of *Text Mining for Historical Analysis*, this is really a printed image of a digital image of a painting of a pipe.

Similarly, a human analyst observing this image might perceive a quote with five distinct words ("this", "is", "not", "a", "pipe"). However, R only interprets one item because the phrase is actually part of the image.

We argue that the same kind of meta-awareness is just as important when working with digital forms, such as data types in programming languages. This is because many analysts have found themselves in the same puzzlement as Magritte's spectators, but while staring at their code for hours while asking themselves why two seemingly identical objects are behaving differently once they run their code.

Magritte's painting offers a framework for understanding this problem. 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 and unto the pieces that make up our constructions.

The two major components that make up our constructions are "data types" (the subject of this section) and "data structures" (the subject of the following section).

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:

```
# load the hansardr library
library("hansardr")

# load the debates for just the decade 1800
data("hansard_1800")

# view the type of the text column
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"`
`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"`

Types impact how R interprets code. For example, the `+` operator performs numeric addition when used with numbers:

```
# Returns 4
2 + 2
```

```
## [1] 4
```

However, if one tries to use `+` with character strings, R will return an error because it does not support addition for character types:

```
"2" + "2"
```

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

Not all values have a data type, however. This is an important note for analysts delving deeper into programming. For example, values like `NA` and `NULL` often appear in datasets with missing records. 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. 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.

Here is just one example of how `NA`'s are treated differently than character values. Our example uses the `msleep` data from the tidyverse, which contains information about the sleep habits of various mammal species. But we will look at just a few columns. The "conservation" 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.

```
# load tidyverse for data wrangling
library(tidyverse)

data("msleep") # load built-in msleep dataset

msleep_subset <- msleep %>% # create a new dataset
  select(name, conservation, sleep_total) %>% # keep only relevant columns
  slice(1:10) # take the first 10 rows

print(msleep_subset) # print the result=
```

```
## # A tibble: 10 x 3
##    name                        conservation sleep_total
##    <chr>                       <chr>              <dbl>
##  1 Cheetah                     lc                  12.1
##  2 Owl monkey                  <NA>                17
##  3 Mountain beaver             nt                  14.4
##  4 Greater short-tailed shrew lc                   14.9
##  5 Cow                         domesticated         4
##  6 Three-toed sloth            <NA>                14.4
##  7 Northern fur seal           vu                   8.7
##  8 Vesper mouse                <NA>                 7
##  9 Dog                         domesticated        10.1
## 10 Roe deer                    lc                   3
```

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_carnivore."

```
# add a new column indicating whether the animal is domesticated, based on the 'conservation' status
msleep_subset %>%
  mutate(is_carnivore = str_detect(conservation, "domesticated"))  # check for 'domesticated'
```

```
## # A tibble: 10 x 4
##    name                        conservation sleep_total is_carnivore
##    <chr>                       <chr>              <dbl> <lgl>
##  1 Cheetah                     lc                  12.1 FALSE
##  2 Owl monkey                  <NA>                17   NA
##  3 Mountain beaver             nt                  14.4 FALSE
##  4 Greater short-tailed shrew lc                   14.9 FALSE
##  5 Cow                         domesticated         4   TRUE
##  6 Three-toed sloth            <NA>                14.4 NA
```

```
##  7 Northern fur seal          vu                   8.7 FALSE
##  8 Vesper mouse               <NA>                 7   NA
##  9 Dog                        domesticated        10.1 TRUE
## 10 Roe deer                   lc                   3   FALSE
```

`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:

```
# add a new column indicating whether the 'conservation' value is missing (NA)
msleep_subset %>%
  mutate(is_missing = is.na(conservation))  # flag missing conservation values
```

```
## # A tibble: 10 x 4
##    name                        conservation sleep_total is_missing
##    <chr>                       <chr>              <dbl> <lgl>
##  1 Cheetah                     lc                  12.1 FALSE
##  2 Owl monkey                  <NA>                17   TRUE
##  3 Mountain beaver             nt                  14.4 FALSE
##  4 Greater short-tailed shrew lc                   14.9 FALSE
##  5 Cow                         domesticated         4   FALSE
##  6 Three-toed sloth            <NA>                14.4 TRUE
##  7 Northern fur seal           vu                   8.7 FALSE
##  8 Vesper mouse                <NA>                 7   TRUE
##  9 Dog                         domesticated        10.1 FALSE
## 10 Roe deer                    lc                   3   FALSE
```

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:

```
# load tidyverse for data wrangling
library(tidyverse)

# replace NA values in the 'conservation' column with the string "missing"
msleep_subset <- msleep %>% # create a new dataset
  select(name, conservation) %>% # keep relevant columns
  mutate(conservation = replace_na(conservation, "missing")) # fill NAs with "missing"

# view the first 10 rows of the updated data
msleep_subset %>%
  slice(1:10) # preview top 10 rows
```

```
## # A tibble: 10 x 2
##    name                        conservation
##    <chr>                       <chr>
##  1 Cheetah                     lc
##  2 Owl monkey                  missing
##  3 Mountain beaver             nt
```

```
##  4 Greater short-tailed shrew lc
##  5 Cow                        domesticated
##  6 Three-toed sloth           missing
##  7 Northern fur seal          vu
##  8 Vesper mouse               missing
##  9 Dog                        domesticated
## 10 Roe deer                   lc
```

Lists are an important data type in R that also function as a data structure. We will explore data structures in the next section. For now, we will focus on lists from the perspective of data types.

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

```
# create a mixed list of strings, whole numbers, and numbers with decimals
continents_population <- list("South America", 644.54, "Africa", 1305, "Europe", 745.64, "Asia", 4587)
```

```
# view the data type
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:

```
# view the type of each item in the list
for (item in continents_population) { # loop through the list
  print(typeof(item))} # print the type
```

```
## [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. In R, data structures are foundational for analyzing historical sources, organizing metadata, processing texts, and analyzing results. Some common data structures in R that we have used for text mining are represented by the following table.

Structure	Dimensions	Homogeneous	Key Use for Text Mining	Example
Matrix	2D	Yes	Numeric structure for word embeddings or term-document counts	matrix(rnorm(300), nrow = 100, ncol = 3)
List	1D	No	Flexible storage for mixed or uniform data—e.g., texts, dates, names	list(1800, "text", "author")
Data Frame	2D	No	Standard tabular format for sources like digitized records or CSVs	data.frame(name = . . . , date = . . .)
Tibble	2D	No	Tidyverse alternative to data frames, optimized for large datasets	tibble(year, text)

As with data types, data structures determine the ways in which data can be accessed, transformed, and analyzed in R. 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. This deliberate choice minimizes the complications often introduced by less interoperable or idiosyncratic data structures.

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" This structure 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 qunateda ecosystem given how effcient `quanteda` is at processing.

While this 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.

```
# load the R packages
library(quanteda) # for text analysis and tokenization
```

```
## Warning: package 'quanteda' was built under R version 4.5.1
```

```
## Package version: 4.3.1
## Unicode version: 15.1
## ICU version: 74.1
```

```
## Parallel computing: 24 of 24 threads used.
```

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

```
library(tidyverse) # for data wrangling
library(hansardr) # for loading historical hansard data

# load hansard data from the 1800s
data("hansard_1800")

# extract text column
texts <- hansard_1800$text

# create a tokens object from the hansard text
tokens_hansard <- tokens(texts)  # tokenize text
```

```r
# perform a kwic (keyword-in-context) search for the word "corn"
kwic_result <- kwic(tokens_hansard, pattern = "corn", window = 5)  # 5-word context window
```

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

```r
head(kwic_result)
```

```
## Keyword-in-context with 6 matches.
##   [text560, 34]         to the practice of hoarding | corn |
##  [text1124, 23]       paid to foreign countries for | corn |
##  [text1283, 58]              Mutiny Bills, the Seed | Corn |
##  [text2284, 11]  and encouraging the importation of | corn |
##  [text2358, 33] all the last regulations respecting | corn |
##   [text3349, 3]                           The Irish | Corn |
##
##  in this of scarcity,
##  and other articles of provision
##  Exportation, the Greenland Whale
##  the act for regulating the
##  , permitting the importation,
##  , Potatoe, and Provision
```

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

```r
# this does not work
slice(tokens_hansard, 1:5)
```

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

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.

```r
# convert the tokens object into a tidy tibble with one token per row
# each token is paired with its corresponding document ID and flattened into a single vector
tokens_tibble <- tibble(doc_id = rep(names(tokens_hansard), # repeat document names the number of token
                                     lengths(tokens_hansard)), # repeat doc IDs by token count
                        token = unlist(tokens_hansard, use.names = FALSE)) # flatten tokens
```

Now we can use `tidyverse` functions freely.

```r
# return the first five rows
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
```

Conclusion

This chapter introduced the core concepts and practical skills historians need to work with historical data in computational environments. Our central argument is that datasets are not neutral or fixed—they are constructed through a series of interpretive decisions. Creating, cleaning, and structuring data are not just preparatory steps for analysis; they are themselves acts of interpretation. Every choice—from how to handle OCR errors to how to structure nested metadata—shapes the historical questions we can ask and the narratives we can build. For this reason, data work requires both technical and critical awareness. We encourage analysts to recognize their place within a larger digital ecosystem, one shaped by data infrastructures, file formats, and technological affordances that influence what can be seen, studied, and preserved.

As we have shown, engaging with digital archives today often means encountering fragmented or inconsistent infrastructures. But rather than viewing these challenges as obstacles, we frame them as opportunities for understanding the larger information ecosystem that surrounds us. In exploring APIs, bulk downloads, and web scraping, we encourage analysts to think expansively about what counts as a source today—and to recognize that the boundaries of an archive are not given, but constructed and contested. When we collect data, we participate in that act of constructing and contesting the boundaries of an archive. In working through the complexities of data cleaning, we have similarly emphasized that each act of data standardization or cleaning reflects a choice with interpretive stakes.

Equally important are the technical dimensions that underlie how data is represented and processed: format, type, and structure. Understanding these concepts equips historians not only to work more effectively with their materials, but also to approach computational tools with greater critical agency. Rather than relying passively on digital platforms, historians can use this knowledge to interrogate and shape the tools themselves.

Ultimately, our goal is to offer a methodological foundation that is both flexible and transferable—one that can support the work of historians, archivists, social scientists, literary scholars, and others engaged in the analysis of historical data. Whether working with digitized print materials, born-digital records, or experimental datasets, readers will find in this chapter a set of techniques for engaging with vast data. Our hope is to encourage a form of historical inquiry that is not overwhelmed by the abundance of digital sources, but instead approaches them with curiosity, creativity, and care. In doing so, we aim to foster a practice that is at once computationally informed and deeply humanistic.