Collecting Newspaper Data in R

Francesca Giannetti

2023-10-02

Introduction

In this introductory workshop, we will access historical and current newspaper data via web APIs using the programming language R. We will use freely available newspaper data from the Library of Congress's Chronicling America website and the New York Times Developer Portal. We will generate dataframes and create some simple data visualizations from downloaded data.

What is the advantage of using an API, as opposed to the regular user interface? The user interface has baked-in limitations that restrict the possible range of queries. One possible justification for using an API is to execute a more complex query. As an example, I can use the state dropdown menu in Chronicling America to return the number of pages of digitized newspapers from New Jersey. But I cannot use that same interface to return the number of times "cats" is mentioned per year. Of course, APIs have limitations too, but generally speaking they support a more granular approach to the data. For quantitative research, web APIs may be used to collect the data corresponding to a query and analyze it in one's programming language of choice.

Web APIs

An interface is the point at which two different systems meet and communicate. An Application Programming Interface (API) thus represents a way of communicating with a computer application by writing a computer program (a set of machine-readable instructions).

APIs commonly provide an interface either for using functionality or for accessing data. With web services, the interface (the set of "functions" one may call to access the data) takes the form of HTTP requests—a request for data sent following the HyperText Transfer Protocol. This is the same protocol (way of communicating) used by your browser to view a web page. An HTTP request represents a message that your computer sends to a web server (another computer on the internet which "serves," or provides information). That server, upon receiving the request, will determine what data to include in the response it sends back to the requesting computer. With a web browser, the response data takes the form of HTML files that the browser can render as web pages. With data APIs, the response data will be structured data, usually JSON or XML, that may be read and acted upon in a programming language.

In short, collecting data from a web API involves sending an HTTP request to a server for a particular piece of data, and then receiving and parsing the response to that request.

URIs

The resource you want to access is specified with a Uniform Resource Identifier (URI). A URI is a generalization of a URL (Uniform Resource Locator)—what one commonly thinks of as a web address. URIs behave much like the address on a posted letter sent within a large organization like a university; you indicate the business address as well as the department and the person. You will get a different response (and different data) from Alice in Accounting than from Sally in Sales.

Note that the URI is the identifier (think: variable name) for the resource, while the resource is the actual data value that you want to access.

Like postal addresses, URIs have a specific format used to direct the request to the right resource.

- *The Base URI*: the domain and part of the path that is included on all resources. It acts as the "root" for any particular resource. For example, Chronicling America has a base URI of `https://chroniclingamerica.loc.gov`.
- *An Endpoint*: a resource on that domain that you wish to access. APIs generally have many endpoints.
- *Query parameters*: to access only partial sets of data from a resource (e.g., only articles mentioning cats) you also need to include a set of query parameters. These are like extra arguments that are given to the request function. Query parameters are listed after a question mark `?` in the URI, and are formed as key-value pairs.
- *An API key and/or secret*: many but not all web services require you to register with them in order to send requests. This allows them to limit access to the data, as well as to keep track of who is asking for what data. To facilitate tracking, many services will provide access tokens or API keys.

Chronicling America

Chronicling America, a project of the Library of Congress, is a searchable database of U.S. newspapers with descriptive information and select digitized newspaper pages. Rutgers University Libraries contributed to Chronicling America through its participation in the National Digital Newspaper Program (NDNP).[1]

Chronicling America provides access to its data in several ways. It is, of course, possible to use the web interface for keyword and metadata searches. Additionally, Chronicling America provides newspaper metadata and data via APIs and bulk downloads.[2] We will experiment with their Search API in this workshop.

New Jersey Titles

While it is arguably more efficient to query the website to find out how many pages of newspapers from New Jersey are included in Chronicling America, we will answer this question using the Search API to illustrate how to formulate API queries.

As mentioned above, the base URI for Chronicling America is `https://chroniclingamerica.loc.gov`.

The endpoint we wish to use is the one for digitized newspaper pages: `/search/pages/results/`.

Some (but not all :P) parameters for the pages endpoint are listed here: https://chroniclingamerica.loc.gov/search/pages/opensearch.xml. As an example, let's use the `state` and `format` parameters to return the digitized newspaper pages for New Jersey in each of the available formats. Load these URLs in a browser window to see what they look like. Note that a few extra pieces of information are visible with the XML and JSON responses, like the author (or provider) and the date of last update. We can also see how many results we get in a single API request: 10 for XML and 20 for JSON.

- https://chroniclingamerica.loc.gov/search/pages/results/?state=New%20Jersey (HTML or default response)
- https://chroniclingamerica.loc.gov/search/pages/results/?state=New%20Jersey&format=atom (XML response)
- https://chroniclingamerica.loc.gov/search/pages/results/?state=New%20Jersey&format=json (JSON response)

To return to our earlier question, there are 303,534 results or pages for the state of New Jersey.

[1] For more information about the Rutgers side of things, visit the New Jersey Digital Newspaper Project website.

[2] Information on the Chronicling America APIs and bulk downloads is available at https://chroniclingamerica.loc.gov/about/api/.

The Search API may be used to query both the U.S. Newspaper Directory (metadata only) and the OCRed text of digitized newspapers. Bear in mind that the OCR is not error free. We will focus on searching newspaper pages (OCR) in this workshop, so I'll only note in passing that the endpoint for the directory is `/search/titles/results/`.

Constructing a Request

Let's transfer our knowledge about URIs to the Search API. To do this, let's introduce a few more parameters.

- `andtext`: the search query
- `format`: 'html' (default), or 'json', or 'atom' (optional)
- `page`: for paging results (optional)

```
# here we use the `request()` function of the httr2 package to construct a request

# create a request
base_url <- "https://chroniclingamerica.loc.gov"
endpoint <- "/search/pages/results/"
params <- "?andtext=cats&format=json" # search for "cats", return a JSON response

req <- request(paste0(base_url, endpoint, params)) # paste together the URI
req
```

```
## <httr2_request>
## GET
## https://chroniclingamerica.loc.gov/search/pages/results/?andtext=cats&format=json
## Body: empty
```

Let's see what the request will send to the server.

```
req %>% req_dry_run()
```

```
## GET /search/pages/results/?andtext=cats&format=json HTTP/1.1
## Host: chroniclingamerica.loc.gov
## User-Agent: httr2/0.2.3 r-curl/5.1.0 libcurl/8.1.2
## Accept: */*
## Accept-Encoding: deflate, gzip
```

Let's actually perform the request and fetch a response.

```
resp <- req %>% req_perform()
resp
```

```
## <httr2_response>
## GET
## https://chroniclingamerica.loc.gov/search/pages/results/?andtext=cats&format=json
## Status: 200 OK
## Content-Type: application/json
## Body: In memory (79372 bytes)
```

We can have a look at the headers (or metadata) of the response we retrieved with `resp_headers()` or a specific header with `resp_header()`

```
resp %>% resp_headers()
```

```
## <httr2_headers>
```

```
## date: Fri, 17 Nov 2023 20:22:52 GMT
## content-type: application/json
## last-modified: Fri, 17 Nov 2023 20:13:54 GMT
## x-robots-tag: noindex, nofollow
## expires: Sat, 18 Nov 2023 20:13:54 GMT
## cache-control: s-maxage=86400, public, max-age=86400
## access-control-allow-origin: *
## access-control-allow-headers: X-requested-with
## x-frame-options: SAMEORIGIN
## content-encoding: gzip
## vary: Accept-Encoding
## x-varnish: 452991170
## age: 539
## via: 1.1 varnish (Varnish/5.2)
## cf-cache-status: HIT
## server: cloudflare
## cf-ray: 827ab0329a7a41af-EWR
```

```
resp %>% resp_header('content-length') # the byte count of the data in the body
```

```
## NULL
```

Let's now examine the structure of the body of the JSON response with the `str()` function.

```
resp %>% resp_body_json() %>% str()
```

Parsing and Transforming the Body of the Response

R is really good at manipulating tables (AKA data frames), less so JSON. Therefore working with a web API often means transforming a JSON or XML response into another format. JSON and XML are both branching structures, and tables are flat, which is to say there's just one level of hierarchy. This is why we will use the `flatten` argument of the `jsonlite` package to transform the response into something R can work with more readily.

Note that while the number of page hits is 3.5 million, we are only seeing the first 20 results.

```
# Parse the content and transform into a data frame
cats <- jsonlite::fromJSON(resp_body_string(resp), flatten = TRUE) %>% data.frame()

# what are the column names?
names(cats)
```

```
##  [1] "totalItems"                "endIndex"
##  [3] "startIndex"                "itemsPerPage"
##  [5] "items.sequence"            "items.county"
##  [7] "items.edition"             "items.frequency"
##  [9] "items.id"                  "items.subject"
## [11] "items.city"                "items.date"
## [13] "items.title"               "items.end_year"
## [15] "items.note"                "items.state"
## [17] "items.section_label"       "items.type"
## [19] "items.place_of_publication" "items.start_year"
## [21] "items.edition_label"       "items.publisher"
## [23] "items.language"            "items.alt_title"
```

```
## [25] "items.lccn"                 "items.country"
## [27] "items.ocr_eng"              "items.batch"
## [29] "items.title_normal"         "items.url"
## [31] "items.place"                "items.page"
```

```
# have a look at the OCR of the first result
cats$items.ocr_eng[1]
```

```
## [1] "IPPPP!\"\"\nCAT IS MESSENGER OF .DEATH!\ngW- ' '\nU-\nCats Like to Be Fondled by Children, But
```

```
# how many page hits?
cats$totalItems
```

```
##  [1] 3524236 3524236 3524236 3524236 3524236 3524236 3524236 3524236 3524236
## [10] 3524236 3524236 3524236 3524236 3524236 3524236 3524236 3524236 3524236
## [19] 3524236 3524236
```

It is also possible to return to the digitized page by retrieving the `items.url` value for any given row. Load this URL in your browser window and delete the ".json" portion at the end to have a look at the digitized page. You will be treated to a nice graphic that was not apparent from the data we captured in our API call. Just a reminder that there is value in returning to the context of the newspaper page.

```
cats[1,30] # the url field of the first row (url is stored in column index 30)
```

```
## [1] "https://chroniclingamerica.loc.gov/lccn/sn83045487/1916-07-13/ed-1/seq-27.json"
```

Phrase Search

This time, we will use some more functions from the `httr2` package to construct our request. Note how we are still breaking up the base URI, the endpoint, and the parameters for greater clarity. We will use another parameter, `phrasesearch`, to look for a two-word phrase instead of a single word. A "sanitary train" is terminology specific to World War I. It refers to an ambulance unit that was assigned to support an entire infantry division. We can expect results mainly from the 1917-1918 time frame.

Note that, as expected, we only get twenty results in our dataframe. But if we examine the `totalItems` field value, we see there are many more. How could we retrieve these? By constructing a for loop and using the `page` field to iterate through all the pages of results. Remember that there are twenty results per page, so you would iterate through `round(3416 / 20)` times. We won't do it here, but I show an example of a for loop later with the NYT API. It could easily be adapted to do the job.

```
# make request and gather response as JSON
wwi_resp <- request("https://chroniclingamerica.loc.gov") %>%
  req_url_path_append("/search/pages/results/?") %>%
  req_url_query(phrasetext = "sanitary train", format = "json") %>%
  req_perform()

# parse and reformat as data frame
wwi <- fromJSON(resp_body_string(wwi_resp), flatten = TRUE) %>% data.frame()
```

Exercise

Over to you. Alter the `phrasetext` parameter below to construct a request and retrieve the response. Alternatively, you could substitute `phrasetext` for `andtext` if your query is a single word.

```
# make request and gather response as JSON
your_resp <- request("https://chroniclingamerica.loc.gov") %>%
```

```
  req_url_path_append("/search/pages/results/?") %>%
  req_url_query(phrasetext = "INSERT YOUR SEARCH TERMS", format = "json") %>%
  req_perform()

# parse and reformat as data frame
your_resp_flattened <- fromJSON(resp_body_string(your_resp), flatten = TRUE) %>% data.frame()
```

Some Data Manipulation and Visualization

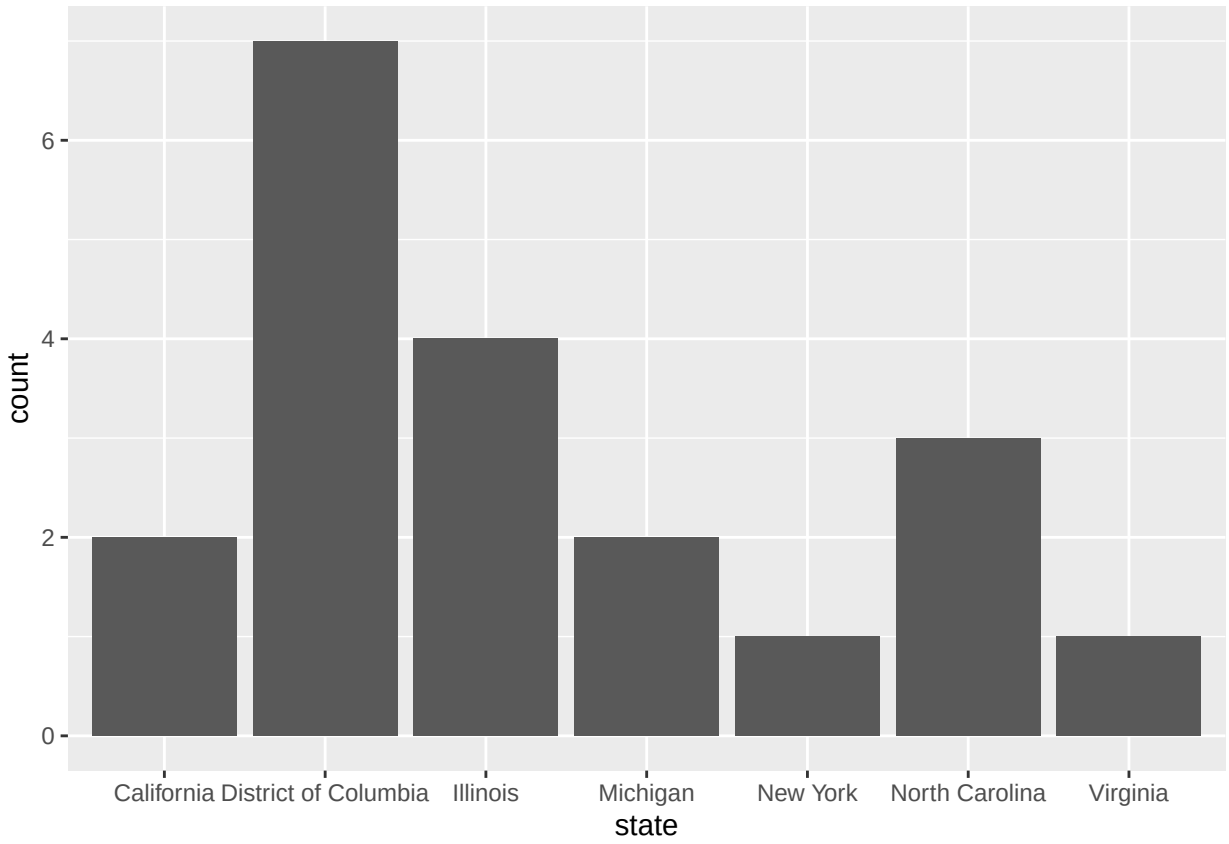
As a reminder, we've only got the first page of results in the `cats` object. But we can begin to form an idea of which state is the most catified with the following code chunk that makes use of the `dplyr` library for data manipulation. We will add a ggplot2 function call to visualize it as a bar chart.

```
# install.packages("dplyr")
library(dplyr)

# the items.state column is still stored as a list; let's create a new column based upon it with charac
cats$state <- unlist(cats$items.state)

# create a derivative dataset called cats_by_state
cats_by_state <- cats %>%
  group_by(state) %>%
  summarize(count = n()) %>%
  arrange(desc(count))

# visualize as a bar plot
ggplot(cats_by_state, aes(x = state, y = count)) + geom_bar(stat = "identity")
```




```
# for help learning dplyr, enter help(package = "dplyr") in the console
```

NYT APIs: Preliminaries

Unlike Chronicling America, the New York Times APIs do require user credentials.

Go to https://developer.nytimes.com/signup. Click Create account and follow the prompts to register. Notice that the New York Times offers several APIs, including Article Search, Archive, Books, Comments, Movie Reviews, Top Stories, and more.

Once logged in, create an app for your research by clicking on your user name in the upper right and selecting Apps > + NEW APP. In your app, be sure to enable the Article Search API. You may elect to enable others in the same app.

Next, copy your API Key and paste it below where indicated. The API Key is personal to you and should be closely guarded. A good way to start is to store it as an environment variable. Alternatively, you can save your key to file locally and access it via the `load()` function.

```
## set your NYT as system variables to access during our session
## replace the stuff in quotes with your personal key

Sys.setenv(nyt_apikey="YOUR_NYT_API_KEY_HERE")

## alternatively, you can save your key to file and access it when needed with the `load()` function

nyt_apikey <- "YOUR_NYT_API_KEY_HERE"
save(nyt_apikey, file = "nytimes_apikey")
```

NYT Article Search API

The New York Times APIs are well documented. Googling will yield lots of tutorials and documentation.

To experiment with queries before running them in R, I recommend using the "Try this API" feature of the Developers' website. Under APIs, go to Article Search. Press "Authorize" (upper right) to authenticate your connection to the API. In the left panel, click `/articlesearch.json` under PATHS to launch the "Try this API" feature. And now let's test some queries.

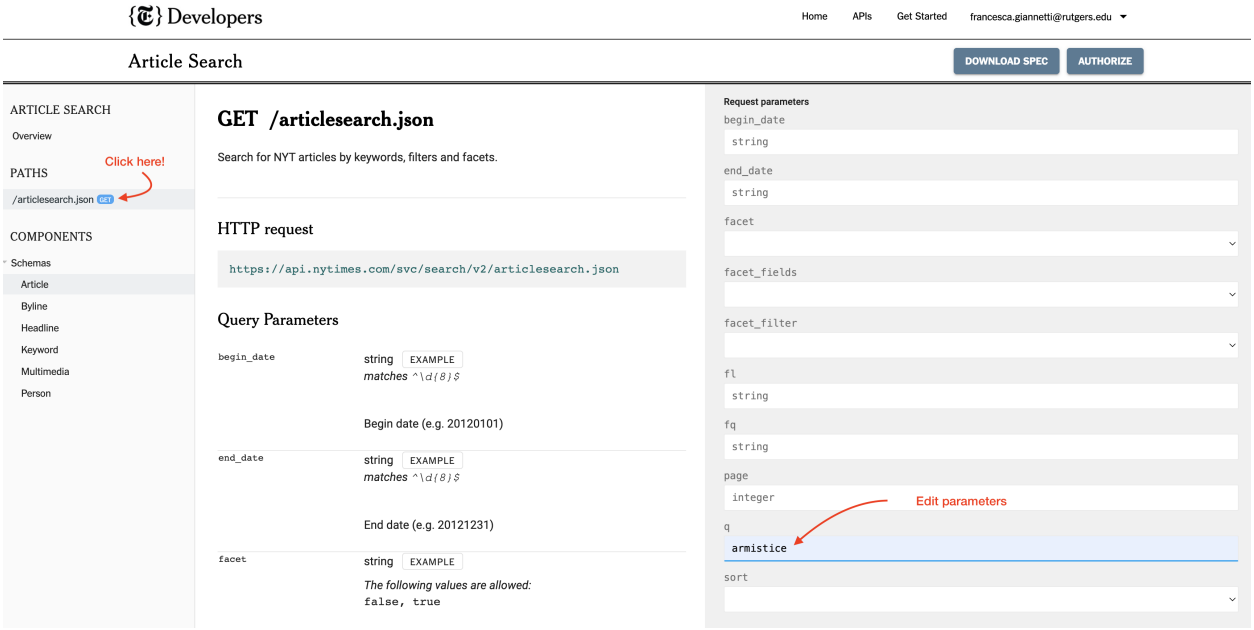


Figure 1: Try this API

As a first example, let's keep it simple. Set the `q` value to 'armistice' (it is not case sensitive) and press "EXECUTE." The response is in JSON. Note how many hits you get and observe the different document types you retrieve.

Now, let's try using some of the filter query parameters. Leave the `q` value set to 'armistice' and this time set `facet` to `true`, `facet_fields` to `document_type`, `facet_filter` to `true`, and `fq` to 'article'. Press "EXECUTE". You should go down from 38,385 hits to 302 hits.

NYT Article Search API: First Request

Let's start with the q parameter, which searches the body, headline, and byline for relevant results. Note that we are still breaking down the URI as before, into base URI, endpoint, and parameters. This time we need to pass our API key into the URI as well.

```
# read your NYT api key back into memory, if needed
load("nytimes_apikey")

nobel <- request("https://api.nytimes.com/svc/search/v2") %>%
  req_url_path_append("/articlesearch.json") %>%
  req_url_query(`api-key` = nyt_apikey,
                q = "Nobel prize literature") %>%
  req_perform()
```

```
nobel
```

```
## <httr2_response>
## GET
## https://api.nytimes.com/svc/search/v2/articlesearch.json?api-key=G4eclG62mG4MIynrC5N5B2TSgFDvimps&q=
## Status: 200 OK
## Content-Type: application/json
## Body: In memory (160616 bytes)
```

Like most web APIs, this one returns a JSON response.

```
nobel %>% resp_body_json() %>% str()
```

As usual, we are going to convert JSON into a dataframe to make it more manageable in R. First, we convert the response to text (string), then we parse and flatten it.

```
nobel <- nobel %>% resp_body_string()

nobel_flat <- fromJSON(nobel, flatten = TRUE) %>% data.frame()
```

NYT Article Search API: For Loop and Rate Limits

As before with the Chronicling America API, we're dealing with the first page of results. Only 10 in this case. This time we will build a for loop to retrieve more.

It's rather easy to hit rate limits with the New York Times APIs. Rate limits are imposed to prevent you from making too many server requests and possibly overwhelming their servers. The NYT FAQs indicate that you should insert 12 seconds in between calls and make no more than 5 requests per minute. There are various ways of handling this, including the `req_throttle()` argument of httr2. We will use `Sys.sleep()` to introduce a generous 15 second pause in between requests.

Nota bene: this code chunk will take approx. 30-45 minutes to run. If you don't want to wait that long, the easiest way to tweak it is to shorten the date window.

```
# read your NYT api key back into memory, if needed
load("nytimes_apikey")

# Parameters
term <- "nobel+prize+literature" # Need to use + to string together separate words
begin_date <- "20030101" # Let's look at 10 years of matches
end_date <- "20130101"

baseurl <- paste0("http://api.nytimes.com/svc/search/v2/articlesearch.json?q=",term,
                  "&begin_date=",begin_date,"&end_date=",end_date,
                  "&facet_filter=true&api-key=",nyt_apikey, sep="")

# create request
initialQuery <- fromJSON(baseurl)

# determine how many pages of results to go through (147)
maxPages <- round((initialQuery$response$meta$hits[1] / 10)-1)

# initialize object to store page results
nobel_pages <- vector("list",length=maxPages)
```

```r
# loop through pages
for(i in 0:maxPages){
  nytSearch <- fromJSON(paste0(baseurl, "&page=", i), flatten = TRUE) %>% data.frame()
  nobel_pages[[i+1]] <- nytSearch
  Sys.sleep(15) # wait 15 seconds between calls
}

# paste together page objects into a new dataframe
nobel_articles <- rbind_pages(nobel_pages)

nobel_articles[1:10,21] # looking at headline for the first 10 rows
```

NYT Article Search API: Visualizations

Coverage by section.

```r
# Visualize coverage by section
p1 <- nobel_articles %>%
  group_by(response.docs.type_of_material) %>%
  summarize(count=n()) %>%
  mutate(percent = (count / sum(count))*100) %>%
  arrange(desc(percent)) %>%
  filter(!is.na(response.docs.type_of_material)) %>%
  top_n(10, percent) %>%
  ggplot(aes(x=reorder(response.docs.type_of_material, percent), y=percent, fill = response.docs.type_o
  geom_bar(stat = "identity") +
  labs(x = "Type of Material by Percent", y = NULL) +
  guides(fill=guide_legend(title=NULL)) +
  theme_minimal() +
  coord_flip()

p1
```

Years with the most coverage.

```r
p2 <- nobel_articles %>%
  mutate(pubDay = gsub("T.*","",response.docs.pub_date),
         pubYear = lubridate::year(pubDay)) %>%
  group_by(pubYear) %>%
  summarise(count=n()) %>%
  ggplot(aes(x=reorder(pubYear, count), y=count)) +
  geom_bar(stat="identity") +
  labs(x = "Year of Publication", y = "Count", title = "Number of Articles Mentioning 'Nobel Prize Lite
  theme_minimal() +
  coord_flip()

p2
```

So, what went on in 2006? There are several factors, but two obvious contributors were the revelation of Gûnther Grass's controversial past as well Turkish writer Orhan Pamuk's award that year, which came as something of a surprise.

```r
# reformat the pub_date as a date object in R, then pass a filter for 2006
nobel_articles %>%
```

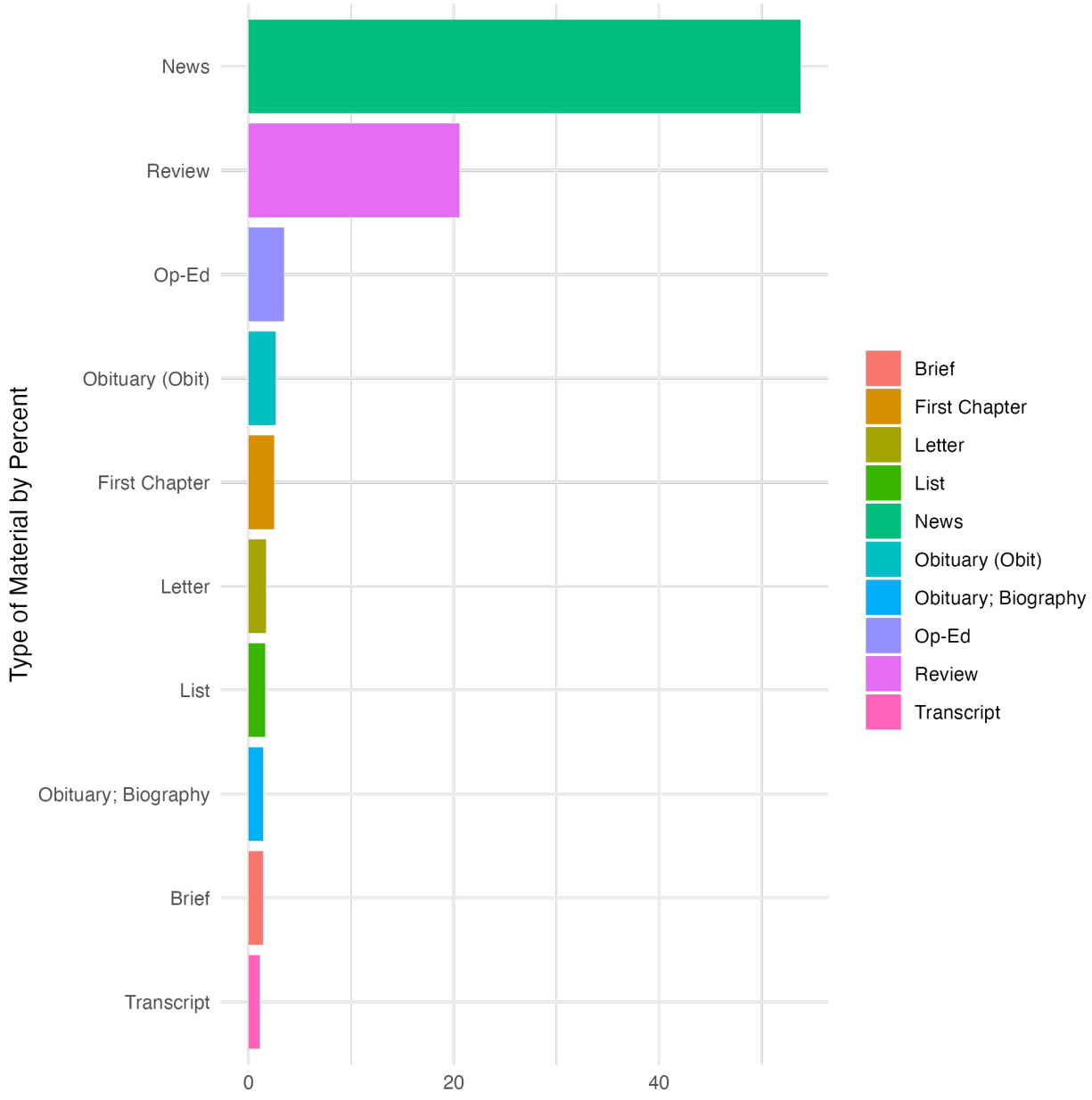


Figure 2: Coverage by section

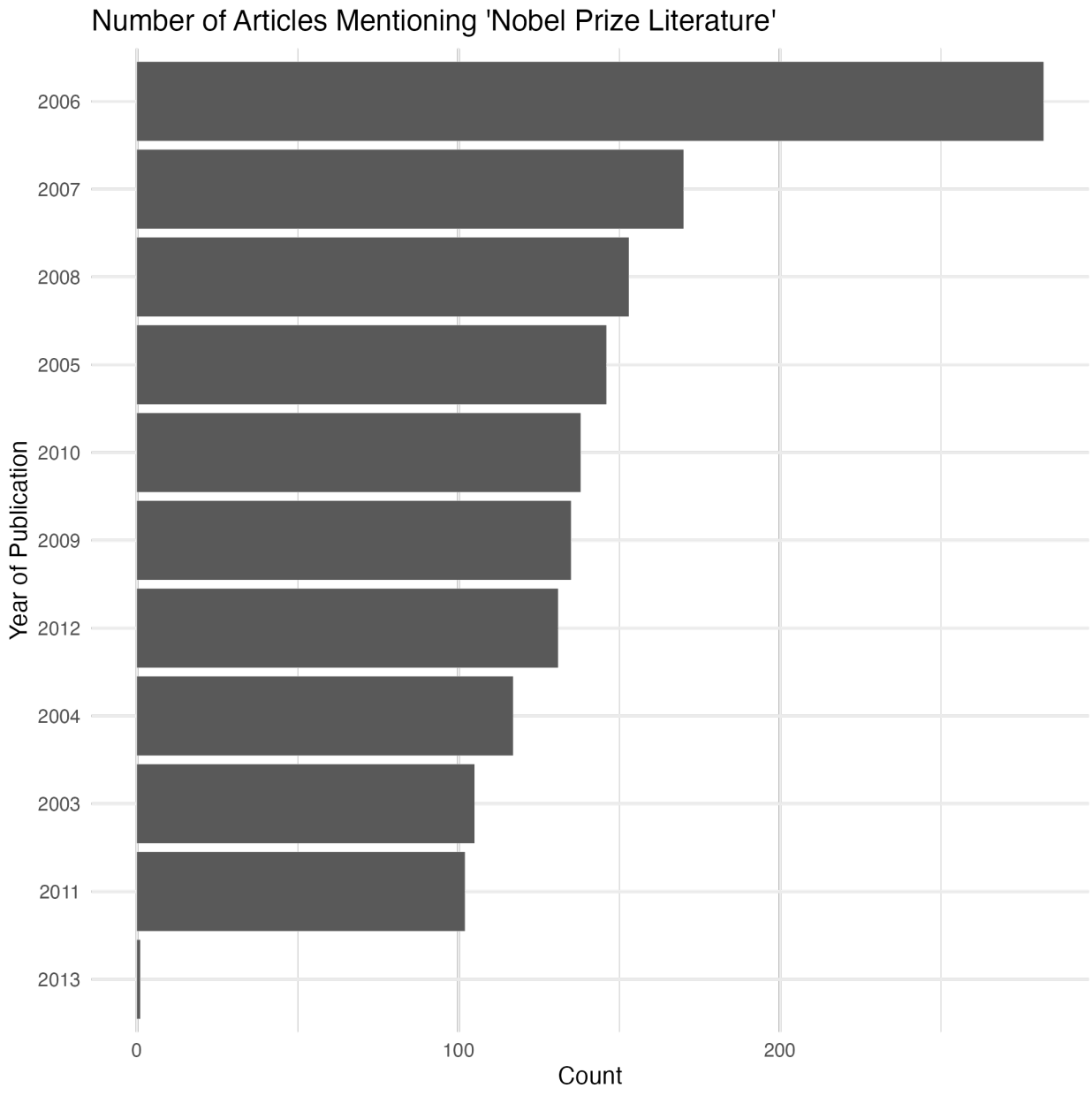


Figure 3: Coverage by year

```
  mutate(pub_date = as.POSIXct(response.docs.pub_date,
                       format = "%Y-%m-%dT%T%z", tz="")) %>%
  filter(pub_date >= "2006-01-01 00:00:00" &
         pub_date < "2007-01-01 00:00:00") %>%
  select(response.docs.headline.main, response.docs.word_count) %>% #select headline and word count
  arrange(desc(response.docs.word_count)) %>% # descending sort by word count
  View() # open table view
```

Exercise: NYT Article Search API

Develop your own query for Article Search. Use Boehmer's slides, which are excellent, as a reminder of the possibilities and syntax: http://jamesboehmer.github.io/nytd2013/#/6.

```
# read your NYT api key back into memory, if needed
# load("nytimes_apikey")

your_nyt <- request("https://api.nytimes.com/svc/search/v2") %>%
  req_url_path_append("/articlesearch.json") %>%
  req_url_query(`api-key` = nyt_apikey,
                q = "YOUR KEYWORDS") %>%
  req_perform()

your_nyt <- your_nyt %>% resp_body_string()

your_nyt_flat <- fromJSON(your_nyt, flatten = TRUE) %>% data.frame()
```

Sources

Boehmer, James. "The New York Times Article Search API." 2013. http://jamesboehmer.github.io/nytd2013/.

Evans, Daniel. "Chronicling America's Newspaper Biographies as Data." The National Endowment for the Humanities (blog), February 16, 2023. https://www.neh.gov/blog/chronicling-americas-newspaper-biographies-data.

Fitzgerald, Jonathan D. "Working with The New York Times API in R." 2018. https://www.storybench.org/working-with-the-new-york-times-api-in-r/.

Freeman, Michael, and Joel Ross. "Accessing Web APIs." In *Technical Foundations of Informatics*. Seattle, WA: UW Information School, 2019. https://info201.github.io/.

LC Labs. "Library of Congress Data Exploration." Last updated May 17, 2023. https://github.com/LibraryOfCongress/data-exploration

Projects that Use Newspaper Data

Interested in exploring projects in the humanities that use newspaper data? You might be interested in the winners of the Chronicling America Data Challenge. I'd also recommend searching *Reviews in Digital Humanities* for any mention of "newspaper."

Lastly, Martin Saveedra (thanks, Martin!) tipped me off to the work of Melissa Dell on the Chronicling America data that improves OCR and detects content areas. I haven't dipped into it yet, but I thought I'd pass along the information. "American Stories": https://dell-research-harvard.github.io/resources/americanstories.