Quantitative Textanalyse

Sitzung 6: Datenerschließung – Scraping II und APIs

Mirko Wegemann

Universität Münster
Institut für Politikwissenschaft

13. November 2024

Logistik

- letzte Woche: erste Sitzung zur Datenerschließung
- heute: weitere Formen der Datenerschließung (Scraping dynamischer Webseiten und APIs)
- zunächst: Kurzpräsentationen von Leon und Jan

Verschiedene Arten von Web-Scraping

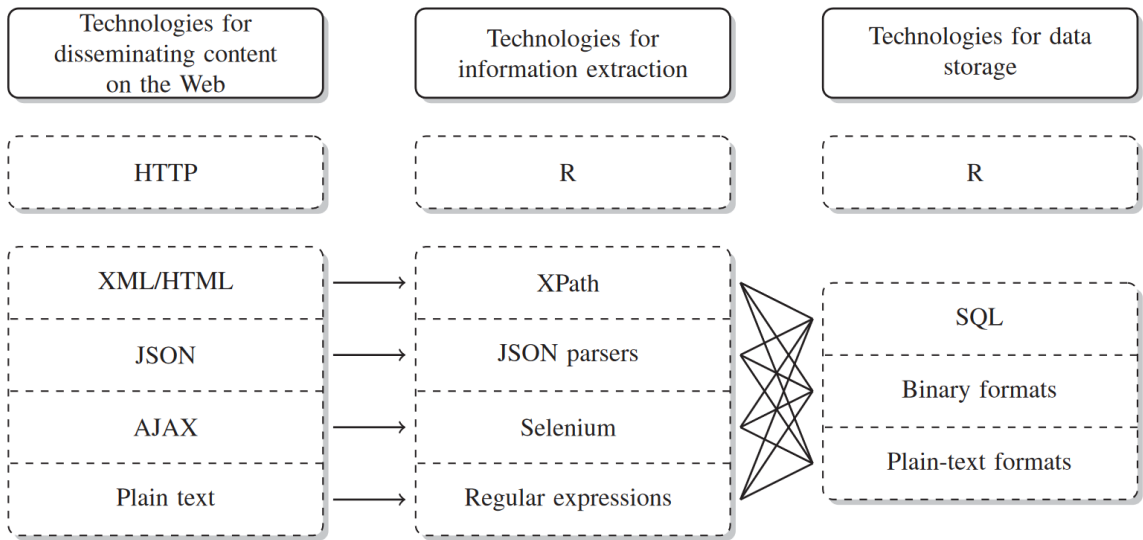
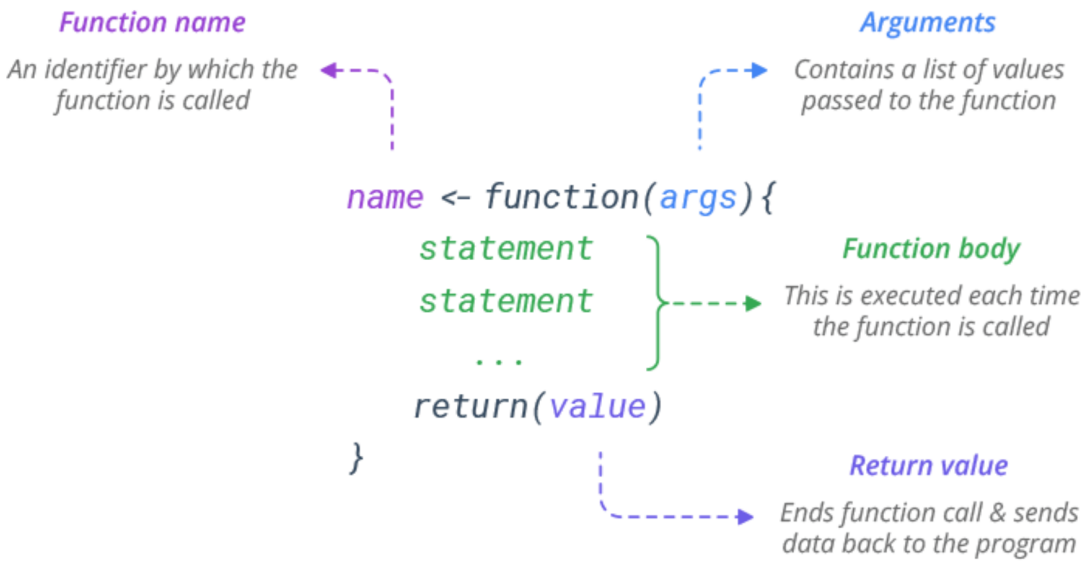


Figure 1.4 Technologies for disseminating, extracting, and storing web data

Munzert (2015, p. 10)

Funktionen in R



Tutorial zu Funktionen
Graphik und Anleitung zu Funktionen

Funktionen für die Automatisierung

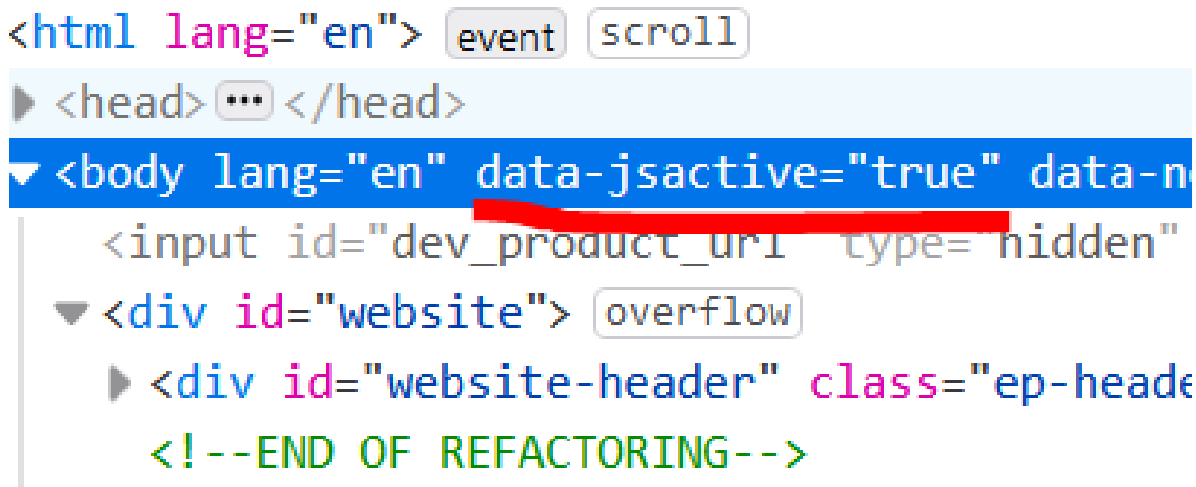
```
1  > h1_scrape <- function(url){
2  +      html <- read_html(urls[[url]])
3  +      links[url] <- html %>%
4  +          html_node("h1") %>%
5  +          html_text()
6  + }
7  >
8  > (links <- sapply(1:length(urls), h1_scrape))
9  [1] "Universitaet Muenster" "Hauptinhalt"
```

Dynamische Webseiten I

Manchmal ändern sich Webseiten nur, wenn wir in einer Browser-Sitzung mit ihnen interagieren (z. B. durch Klicken auf bestimmte Objekte). Für diese **Webseiten** ist `rvest` nicht anwendbar.

Dynamische Webseiten II

Normalerweise erkennt man solche Seiten an der Verwendung von Javascript



Funktionalität von RSelenium

Lösung?

Figure: RSelenium

RSelenium wurde ursprünglich entwickelt, um Webseiten zu testen; wir verwenden es, um Befehle an ein virtuelles Browserfenster zu senden (z.B. um einen Button zu klicken)

Vor der Verwendung von RSelenium

Wenn möglich, ist der statische dem dynamischen Ansatz vorherzuziehen, da er einfacher und weniger fehleranfällig ist:

- andere Webseiten überprüfen, die möglicherweise die gleichen Informationen enthalten
- nach anderen Unterverzeichnissen suchen, die denselben Inhalt speichern könnten
- die Wayback Machine überprüfen
- die Suchfunktion einer Webseite verwenden, um Ergebnisse aufzulisten (Beispiel: Pressemitteilungen der Fraktion von Die Linke)

Pipeline von RSelenium und wdman

Ein typischer Ablauf sieht wie folgt aus:

- Konfiguration einer Sitzung (mittlerweile etwas komplex; wir benötigen zusätzlich `wdman`)
- Browser öffnen
- URL aufrufen
- ggf. Cookies akzeptieren/ablehnen
- CSS-Selektor für Button identifizieren und klicken (falls nötig, wiederholen)
- gewünschtes Objekt abrufen (wie zuvor suchen wir meistens nach Links auf individuelle Pressemitteilungen)

Setup

- Treiber konfigurieren
- Client extrahieren
- Webseite aufrufen

```
> url <- "https://www.europarl.europa.eu/news/en"
> rd <- rsDriver(browser = "firefox",
+                chromever = NULL,
+                port = sample(1:65535, 1),
+                verbose = FALSE)
> browser <- rd[["client"]]
> browser$navigate(url)
```

Und jetzt in R!

Identifizieren und Klicken auf einen Button

1. Finde ein Element anhand seines CSS-Selektors
2. speichere es als Objekt
3. verwende die Funktion clickElement()

```
1  > cookies <- browser$findElement(using = 'css
       selector', value='.epjs_agree:nth-child(1)
       span')
2  > cookies$clickElement()
```

Wie man den Text einer dynamischen Webseite extrahiert

- Elemente mit CSS-Selektor identifizieren (wie zuvor über SelectorGadget möglich)
- Elemente enthalten meist viele weitere (für uns unnötige) Informationen
- um nur den reinen Text zu extrahieren, müssen wir eine Schleife basteln und nur den Text des Elements extrahieren (`getElementText()` funktioniert ähnlich wie `html_text()` von `rvest`)

Wie man den Text einer dynamischen Webseite extrahiert II

```
1  > paragraphs <- browser$findElements(using = 'css
       selector', value='p')
2  > text <- c()
3  > for(i in 1:length(paragraphs)){
4  +     text[i] <-
         paragraphs[[i]]$getElementText()[[1]]
5  + }
6  > text[1]
7  [1] "On Thursday afternoon, the Eurovision debate
       between the lead candidates for the presidency
       of the European Commission took place in the
       European Parliament."
```

Wie man den Link einer dynamischen Webseite extrahiert

- um nur den Link zu extrahieren, benötigen wir die Funktion `getElementAttribute()`, die `html_attr()` von `rvest` ähnelt

```
> urls_euparl <- browser$findElements(using = 'css selector', value='.ep_title > a')
> urls_euparl2 <- c()
> for(i in 1:length(urls_euparl)){
+     urls_euparl2[i] <- urls_euparl[[i]]$getElementAttribute('href')[[1]]
}
```

Datenverarbeitung

Sobald wir die einzelnen Links zu Seiten heruntergeladen haben, können wir oft in `rvest` weitermachen (was einfacher zu handhaben ist).

Weitere Möglichkeiten mit RSelenium

Wir können mit RSelenium prinzipiell jegliche Browser-Funktion nachahmen. Eine oftmals hilfreiche Funktion ist es bspw. Formulare auf Seiten auszufüllen (wie ein Suchfeld).

```
1     > browser$navigate(url)
2     > search <- browser$findElement(using = 'css
        selector', value="#search-field")
3     > search$clearElement
4     >
        search$sendKeysToElement(list("economy",key="enter")
```

Ein letztes Wort zu RSelenium

Selenium ist ein klassisches Tool für Programmierer*innen. Ein Großteil von ihnen arbeitet mit Python. Dementsprechend hinkt die Entwicklung für R teils hinterher.
Um den vollen Umfang von Selenium nutzen zu können, steigt auf Python um.

Was sind APIs?

Application **P**rogramming **I**nterface sind von Webseiten-Betreiber*innen angebotene Schnittstellen, welche uns erlauben Daten herunterzuladen und weiterzuverarbeiten.

- meist an Webentwickler*innen gerichtet, welche bspw. Apps, die auf Daten einer Webseite zugreifen, entwickeln können
- dementsprechend ist es möglich, dass für Forscher*innen interessante Daten nicht immer zugänglich gemacht worden sind (z.B. können wir über die NYT-API keine Volltexte herunterladen)

Zugang zu APIs

Zugang zu APIs kann meist über eine Registrierung erworben werden.

- einige Registrierungen sind kostenfrei und simpel (Bsp.: Manifesto Project oder NYT)
- andere sind stark eingeschränkt und nur bedingt kostenfrei (z.B. X API)

Nutzung der API

Nach der Registrierung erhalten wir meist einen Zugangsschlüssel, über den wir auf die API zugreifen können.

- Manueller Zugang über JSON-Funktionen (`jsonlite`), in Kombination mit API-Dokumentation (z.B. NYT Article Search API)
- Viele APIs verfügen über sogenannte Wrapper; Implementationen, welche deren Nutzung in `R` stark vereinfachen

API: API-Key festlegen

Nutzt das Paket `usethis`, um über die Funktion `edit_r_environ()` Euren API-Key in der Projektumgebung zu hinterlegen. Dies dient zur Sicherheit vor missbräuchlichem Verhalten, damit Euer Key nicht offen sichtbar im Code ist.

```
1  > if (!require('usethis'))
       install.packages("usethis")
2  > edit_r_environ()
3   Modify 'C:/Users/Nutzer/Documents/.Renviron'
4   Restart R for changes to take effect
```

Für die NYT-API würdet ihr NYTIMES_API_KEY="YOUR_KEY" in die Umgebung schreiben (ersetzt "YOUR_KEY" mit Eurem API-Key). Nun müsst ihr die Umgebung speichern und RStudio neustarten.

API: Query definieren und senden

```
1  > if (!require("jsonlite"))
       install.packages("jsonlite")       # manual
       handling of APIs
2  > url_query <-
       paste0("https://api.nytimes.com/svc/search/v2/
3  articlesearch.json?q=trump&api-key=",
       Sys.getenv("NYTIMES_API_KEY"))
4  > query <- fromJSON(url_query)
```

API: Daten in Datensatz umwandeln

Die Daten erhalten wir in einer Liste, welche wiederum aus mehreren Listen (u.a. Meta-Informationen zur API-Anfrage) besteht. Wir sind meistens natürlich an den Daten interessiert, welche wir in diesem Fall über die geschachtelte Struktur in einen `data.frame` umwandeln können.

```
> df <- data.frame(query$response$docs)
```

API: Automatisierung

Auch bei APIs müssen wir oftmals Loops bauen, um mehr als eine gewisse Anzahl an Datenpunkten zu erhalten.

```
> df <- c()
> for(i in 0:5){
  url_query <-
      paste0("https://api.nytimes.com/svc/search/v2/
  articlesearch.json?q=trump&api-key=",
      Sys.getenv("NYTIMES_API_KEY"), "&page=", i)
  query
  df_temp <- data.frame(date =
      query$response$docs$pub_date, url =
      query$response$docs$web_url, summary =
      query$response$docs$abstract,
      lead_paragraph =
      query$response$docs$lead_paragraph)
  df <- rbind(df, df_temp)
  Sys.sleep(12)
}
```

Das Manifesto Project

Figure: Das Manifesto Project

Das Manifesto Project ist ein internationales Forschungsprojekt, welches Wahlprogramme von mehr als 1,000 Parteien in 68 Ländern gesammelt und codiert hat.

ManifestoR

Das Manifesto Project bietet eine API an, mithilfe derer wir auf den Korpus des Projekts (mit mehr als 1.9 Millionen "Quasi-Sätzen") zugreifen können.

- Zugriff entweder wieder über `jsonlite`
- ...oder `manifestoR`, einen `R`-Wrapper

Hier besprechen wir nur die Umsetzung mithilfe von `manifestoR`.

Und jetzt in R!

API-Key festlegen

Wie zuvor müssen wir als erstes den API-Key in unsere Umgebungsvariable hinterlegen und festlegen.

```
1  > if (!require("manifestoR"))
       install.packages("manifestoR")
2  > mp_setapikey(key=Sys.getenv("MP_API_KEY"))
```

Datenzugriff

Die Daten des Manifesto-Korpus lassen sich recht einfach über die Funktion `mp_corpus` abrufen. Wir müssen immer **spezifizieren**, welche Textstellen wir benötigen (entweder über *countryname* oder *party*).

```
> german_mp <-
    mp_corpus_df(countryname=="Germany")
```

Ausblick

- nächste Woche bereiten wir Daten für die Textanalyse vor
- Literatur:
  1. Denny, M., & Spirling, A. (2017, September). Text Preprocessing for Unsupervised Learning: Why It Matters, When It Misleads, and What to Do about It. https://doi.org/10.2139/ssrn.2849145
  2. Hvitfeldt, E., & Silge, J. (2022). *Supervised Machine Learning for Text Analysis in R*. CRC Press – Kapitel 2-4

Literatur I

Denny, M., & Spirling, A. (2017, September). Text Preprocessing for Unsupervised Learning: Why It Matters, When It Misleads, and What to Do about It. https://doi.org/10.2139/ssrn.2849145

Hvitfeldt, E., & Silge, J. (2022). *Supervised Machine Learning for Text Analysis in R*. CRC Press.

Munzert, S. (2015). *Automated Data Collection with R: A Practical Guide to Web Scraping and Text Mining* (1st ed.). Wiley.