paperboy — A Collection of News Media Scrapers

Abstract

The philosophy of the R package paperboy is that the package is a repository for webscraping scripts for news media sites, with advanced features for quick data retrieval — even for content behind log-ins or anti-scraping measures. Many data scientists and researchers write their own code when they have to retrieve news media content from websites. At the end of research projects, this code is often collecting digital dust on researchers hard drives instead of being made public for others to employ. paperboy offers writers of webscraping scripts a clear path to publish their code and earn co-authorship on the package, while promising users to deliver news media data from many websites in a consistent format. With 177 covered as of today and a default scraper that often works well enough, paperboy can already facilitate a large range of research projects.

Keywords: data mining, open data

paperboy — A Collection of News Media Scrapers

Drawing inspiration from the 1985 Atari video game of the same name, in which the player needs to deliver newspaper issues despite increasingly difficult to evade hazards, the R package paperboy is designed to navigate the complex world of online news data collection. Despite the common claim in almost every text that applies text-as-data techniques that text is widely available nowadays (Boumans & Trilling, 2016; e.g., Grimmer & Stewart, 2013; Laurer et al., 2024), many projects on media coverage struggle to obtain the data they need. Usually the first attempts to obtain media data is quickly frustrated by one or several inadequacies of commercial platforms such as *LexisNexis* and *Factiva*: the outlets researchers would like to analyse are not available due to copyright disputes; automated batch downloading is against the terms of service and might lead to exclusion from the services; the cost for API access for individual researchers is prohibitive, which means that large scale media research is often limited to researchers at institutions which can afford it; and finally, whatever researchers gather through these services cannot be shared later with or reproduced by the research community — due to legal restrictions, but also because search results change over time and the platforms do not offer unique links or identifiers to previous searches (Weaver & Bimber, 2008).

The next step is then often to obtain data through *web-scraping*. And thankfully, the basics of obtaining data from different news sites and parsing them into a standardized format fit for computational analysis is becoming a skill more widely available. However, obtaining data on a large scale is not usually taught and most sites are not as scraping-friendly as Wikipedia, IMDB and the other examples usually used in tutorials. Finally, once a researcher has overcome the hazards of a particular outlet, the intricately crafted code usually collects digital dust on the hard drives of said researcher after the end of a project. Only for others to duplicate the efforts in the near future.

The aim of paperboy is to bring researchers who need media data, but lack the technical skills to obtain them and those who already have web-scraping code for specific outlets, or the skills and motivation to write them, together and provide both with a reliable and scalable

backbone codebase. For users, the incentive is clear — they get easy access to media data. For developers, the advantage over publishing their code themselves is that `paperboy` is a community project, where contributing a single parser for an outlet earns developers co-authorship. This can lend credibility to CSS researchers who want to add software contributions to their CVs and earn them citations in the field. This model should contribute over time to a comprehensive collection of parsers for all important news sites. In comparison to projects like the Python package `Newspaper3k`, `paperboy` does not follow a 'one-size-fits-all' approach (although a generic parsers, which serves this function is available as a fallback), but acknowledges the small differences between news website that can lead to sub-optimal results when attempting to capture all content with the same approach.

This short overview of the package first covers the user perspective to show how everyone can use `paperboy` to gather news data. It then highlights the procedure of contributing to the package as a developer and showcases the tools built to make a contribution as straightforward as possible.

Usage for Users

In the most basic scenario, data can be obtained with a single command:

```
library(paperboy)
cnn_df <- pb_deliver("https://bit.ly/47bPVWP")
dplyr::glimpse(cnn_df)
#> Rows: 1
#> Columns: 9
#> $ url          <chr> "https://bit.ly/47bPVWP"
#> $ expanded_url <chr> "https://www.cnn.com/travel/lonely-planets-top-places-to-~
#> $ domain       <chr> "cnn.com"
#> $ status       <int> 200
#> $ datetime     <dttm> 2023-10-31 09:12:32
#> $ author       <chr> "Maureen Ohare"
#> $ headline     <chr> "Lonely Planet's top places to go in 2024"
#> $ text         <chr> "Get your wishlist fired up, Lonely Planet just revealed ~
#> $ misc         <list> [<tbl_df[1 x 1]>]
```

In this case, a parser for cnn.com is already available (and shortened URLs are handled correctly), which can be checked, using `pb_available("https://cnn.com")` (`pb_available()` without arguments shows all available parsers, which comprise 177 outlets at the time of writing). When no parser is available, a warning is issued, once per session, and a

generic approach, which combines the knowledge of all parsers, is used:

```
mbpassion_df <- pb_deliver("https://bit.ly/3MqSnk7")
#> ! No parser for domain mbpassion.de yet, attempting generic approach.
```

The generic parser delivers the correct information in most cases, alas often with some superfluous content, like in the author column here.

```
dplyr::glimpse(mbpassion_df)
#> Rows: 1
#> Columns: 9
#> $ url          <chr> "https://bit.ly/3MqSnk7"
#> $ expanded_url <chr> "https://mbpassion.de/2023/10/mercedes-benz-nutzt-intern-~
#> $ domain       <chr> "mbpassion.de"
#> $ status       <int> 200
#> $ datetime     <dttm> 2023-10-30 23:01:43
#> $ author       <chr> "Markus Jordan, , Markus Jordan·31. Oktober 2023, Markus ~
#> $ headline     <chr> "Mercedes-Benz nutzt intern ChatGPT für Mitarbeiter"
#> $ text         <chr> "Mercedes-Benz führt für seine Beschäftigten eine interne~
#> $ misc         <list> [<tbl_df[1 x 0]>]
```

paperboy is especially well-suited to collect larger datasets of news though. The initial set of URLs for this purpose might come from several sources:

- a research project might share their collection of URLs with news about a specific topic[1] which can then be 'rehydrated' into a complete dataset (Gruber et al., 2023)
- researchers can use a tool like *Media Cloud* to search URLs for stories featuring keywords and download the URLs
- or, for unfolding events, researchers can use paperboy to continuously scrape the RSS feed of an outlet, which is populated with the most recently published articles

To demonstrate the *Media Cloud* approach, we use the httr2 package (Wickham, 2024) to search the API for stories about "harris" since the start of the year:

[1] Unlike scraping the content from news outlets, sharing the full data with others is not considered fair use and may lead to legal ramifications for researchers (Hennesy & Samberg, 2019).

```
library(httr2)
test_data <- request("https://search.mediacloud.org/api/") |>
  req_url_path_append("search/story-list") |>
  req_headers(
    Authorization = paste("Token", Sys.getenv("MC_TOKEN")),
    Accept = "application/json"
  ) |>
  req_url_query(
    q = "harris",
    start = "2024-01-01",
    ss = 107736L # source ID of huffpost.com
  ) |>
  req_perform() |>
  resp_body_json() |>
  purrr::pluck("stories") |>
  dplyr::bind_rows()
dplyr::glimpse(test_data)
#>  Rows: 1,000
#>  Columns: 8
#>  $ id           <chr> "2d52f76f252c8787442654578c09f83e3ff9ed1fe961cf5262ecbf1a~
#>  $ media_name   <chr> "huffpost.com", "huffpost.com", "huffpost.com", "huffpost~
#>  $ media_url    <chr> "huffpost.com", "huffpost.com", "huffpost.com", "huffpost~
#>  $ title        <chr> "Jill Biden Opens Up About President Biden Ending Reelect~
#>  $ publish_date <chr> "2024-10-21", "2024-10-22", "2024-10-22", "2024-10-21", "~
#>  $ url          <chr> "https://www.huffpost.com/entry/jill-biden-president-reel~
#>  $ language     <chr> "en", "en", "en", "en", "en", "en", "en", "en", "en", "en~
#>  $ indexed_date <chr> "2024-10-22 11:25:17.490875", "2024-10-22 11:25:12.527526~
```

To download this data, we can either use `pb_deliver()` directly, or make the underlying process more explicit (and record the time it takes) by using `pb_collect()`:

```
res <- bench::mark(
  test_data_hydrated <- pb_collect(test_data$url)
)
```

On a normal connection, the entire dataset of 1,000 articles is pulled from *huffpost.com* in 7 seconds. The reason why it is so fast is that `pb_collect()` uses asynchronous requests, which means that connections to all target URLs — up to 100 by default — are established in parallel and kept open until all the data has finished streaming to memory. This approach is highly effective and much faster than running requests sequentially. It might also go against what many introductions to web-scraping still recommend, which is to cautiously rate-limit batch requests, e.g., to 5 per minute. Yet, this advice appears outdated today when news corporations serve millions of visits each day—although it might still be applicable to small, community operated fringe media, in which case the setting can be changed.

The raw HTML data could be archived at this point, which can be useful if an error in the parsing is discovered later. To make the raw data suitable for analysis, it can be parsed using

`pb_deliver()`:

```
test_data_hydrated_parsed <- pb_deliver(test_data_hydrated)
dplyr::glimpse(test_data_hydrated_parsed)
#>  Rows: 1,000
#>  Columns: 9
#>  $ url          <chr> "https://www.huffpost.com/entry/jill-biden-president-reel~
#>  $ expanded_url <chr> "https://www.huffpost.com/entry/jill-biden-president-reel~
#>  $ domain       <chr> "huffpost.com", "huffpost.com", "huffpost.com", "huffpost~
#>  $ status       <int> 200, 200, 200, 200, 200, 200, 200, 200, 200, 200, 200, 20~
#>  $ datetime     <dttm> 2024-10-21 20:31:26, 2024-10-21 15:17:50, 2024-10-21 14:~
#>  $ author       <chr> NA, NA, NA, "Adriana Gomez Licon", NA, NA, "Amanda Seitz"~
#>  $ headline     <chr> "Jill Biden Opens Up About President Biden Ending Reelect~
#>  $ text         <chr> "Reporter\nIn a "Good Morning America" interview on Monda~
#>  $ misc         <list> [<tbl_df[1 x 1]>], [<tbl_df[1 x 1]>], [<tbl_df[1 x 1]>],~
```

This data is now suitable for analysis in other `R` packages, like `quanteda` (Benoit et al., 2018):

```
library(quanteda)
corpus(test_data_hydrated_parsed) |>
  tokens(
    remove_punct = TRUE,
    remove_symbols = TRUE,
    remove_numbers = TRUE
  ) |>
  dfm() |>
  dfm_remove(stopwords("en")) |>
  topfeatures()
#>     trump  huffpost      said        us    harris      free president   support
#>      6211      5660      4584      4545      4338      3160      3092      2872
#>      like      help
#>      2812      2589
```

Another aspect of web-scraping that is automatically taken care of is managing cookies. Cookies are used by websites to remember preferences, for example if a privacy banner is displayed over the text of an article, and to handle log-ins. This means that even articles behind a paywall can be obtained, if a subscribed user is willing to read their cookies into `R` after logging in on a website with their credentials. Take the article below from a German outlet, which displays only a short snippet of the article, accessible only to subscribers. Using a standard approach with `rvest` (Wickham, 2022), we can obtain this snippet:

```
URL <- "https://www.zeit.de/arbeit/2024-10/viertagewoche-arbeitszeit-umsatz-effizienz-deutschland"

library(rvest)
html <- read_html(URL)
text <- html |>
  html_elements(".article-body p") |>
```

```
  html_text2() |>
  paste(collapse = "\n")
nchar(text)
#> [1] 589
```

After supplying the cookies of an authenticated subscriber, however, `paperboy` can get the html of the full text by accessing cookies stored using `cookiemonster` (Gruber, 2023)[2]:

```
cookiemonster::add_cookies("cookies.txt")
#> v Cookies for zeit.de, www.zeit.de put in the jar!
content_df <- pb_collect(URL, use_cookies = TRUE)
```

Since this data is compatible with `rvest`, we can draw a direct comparison:

```
html2 <- read_html(as.character(content_df$content_raw))
text2 <- html2 |>
  html_elements(".article-body p") |>
  html_text2() |>
  paste(collapse = "\n")
nchar(text2)
#> [1] 5064
```

Checking the number of characters, we see that by sending the cookies with the request, we get more than 8 times as much content as before — which is the entire article.

Usage for Developers

For developers, `paperboy` promises to make writing web-scrapers as easy and efficient as possible. The recommended workflow is to first search for an RSS feed to obtain some data for development and testing. For this the function `pb_find_rss()` can be used to query different sources and test common paths:

```
pb_find_rss("https://www.novinky.cz/")
#> v Looking through links on the main page [367ms]
#> v Looking through common pahts on the site [429…
#> v Querying feedly API [890ms]
#>   Discovered 2 URLsCheck manually to see which ones fit
#> # A tibble: 2 × 2
#>   source           url
#>   <chr>            <chr>
#> 1 common locations https://novinky.cz/rss
#> 2 feedly API       https://www.novinky.cz
```

[2] See the `cookiemonster` package vignette for details on how to supply cookies.

In this case, `/rss`, which is the most common location for RSS feeds, was found to contain RSS encoded data of recent news. Using this, we can create a test `data.frame` by parsing the RSS feed and obtaining all stories from it:

```
rss <- "https://novinky.cz/rss"
test_df <- pb_collect(rss)
dplyr::glimpse(test_df)
#> Rows: 30
#> Columns: 5
#> $ url          <chr> "https://www.novinky.cz/clanek/krimi-hromadna-nehoda-devi~
#> $ expanded_url <chr> "https://www.novinky.cz/clanek/krimi-hromadna-nehoda-devi~
#> $ domain       <chr> "novinky.cz", "novinky.cz", "novinky.cz", "novinky.cz", "~
#> $ status       <int> 200, 200, 200, 200, 200, 200, 200, 200, 200, 200, 200, 20~
#> $ content_raw  <html_cnt> <!doctype html>
#> <html lang="cs"><head><meta charSet=~
```

Developers can then use the `usethis`-style (Wickham et al., 2023) function below to create a new source file, generated from a template:

```
use_new_parser(
  x = test_df$expanded_url,
  rss = rss,
  test_data = test_df
)
#> v Created R file with function template [26ms]
#> v Added entry to inst/status.csv [24ms]
#> v ./R/deliver_ceskatelevize_cz.R passed tests [334ms]
#> i Trying to parse raw data
#> i Checking parser ./R/deliver_ceskatelevize_cz.R for consistency
#> i Checking results
#> X 6.67% of text values failed to parse
#> X Some tests failed. But you will get there! Don't stop making now!
#> X ./R/deliver_ceskatelevize_cz.R did not pass tests. [456ms]
#> Would you like me to load the test data and results into your environment? (yes/No/cancel)
```

The first time the `use_new_parser()` is used on a new website, the source file for the new parser, containing just the template at this point, is opened. As can be seen below, the template code already includes all important elements and some comments to guide even a novice developer with basic understanding of web-scraping to complete a working parser in a short time.

```
#' @export
pb_deliver_paper.novinky_cz <- function(x, verbose = NULL, pb, ...) {

  # updates progress bar
  pb_tick(x, verbose, pb)

  # raw html is stored in column content_raw
  html <- rvest::read_html(x$content_raw)
```

```
  # datetime
  datetime <- html %>%
    rvest::html_element("") %>%
    rvest::html_attr("") %>%
    lubridate::as_datetime()

  # headline
  headline <- html %>%
    rvest::html_element("") %>%
    rvest::html_attr("")

  # author
  author <- html %>%
    rvest::html_element("")  %>%
    rvest::html_text2() %>%
    toString()

  # text
  text <- html %>%
    rvest::html_elements("") %>%
    rvest::html_text2() %>%
    paste(collapse = "\n")

  # the helper function safely creates a named list from objects
  s_n_list(
    datetime,
    author,
    headline,
    text
  )

}
```

When `use_new_parser` is called again as shown above, the parser is automatically tested against the test data. If the test should not succeed at first, an encouraging text is shown. Only when more than 95% of the data is parsed correctly does the developer get the option to enter their name into the `status.csv` file, which contains all parsers, and contribute the new function. A gamification aspect, which rewards the developer by showing them some ASCII art and displaying a leaderboard of contributors on the upcoming paperboy website are planned.

```
use_new_parser(
  x = test_df$expanded_url,
  rss = rss,
  test_data = test_df
)
#>   Created R file with function template [565ms]
#>   ./deliver_novinky_cz.R passed initial tests [1.1s]
#>   Trying to parse raw data
#>   Checking parser ./deliver_novinky_cz.R for consistency
#>   Checking results
#>   100.00% of text values failed to parse
#>   Some tests failed. But you will get there! Don't stop coding now!
#>   ./deliver_novinky_cz.R did not pass tests. [16s]
```

Conclusion

The philosophy behind the development of `paperboy` was to provide easy and fun to use tools for development of html parsers and advanced infrastructure functions to obtain public news content as well as content behind paywalls. This was done to accomplish the larger goal of building a community project where people with at least basic understanding of web-scraping can contribute to map the landscape of media outlets and eventually provide a comprehensive collection of web-scrapers for all relevant news outlets. The currently implemented 178 parsers should thus be seen as just a first step towards this larger goal.

Nevertheless, in it's current state, `paperboy` already might encourage researchers to create their own datasets rather than relying on commercial services or wasting their efforts by duplicating work that has already been done. Additionally, to the issues named above, datasets obtained from commercial services are usually impossible to recreate or share, making research irreproducible (Chan et al., 2024). As `paperboy` makes it quick and easy to 'rehydrate' these URL collections, it opens a new avenue to include copyrighted data in reproduction materials by simply sharing a list of links (Gruber et al., 2023), solving a crucial issue in computational communication research.

References

Benoit, K., Watanabe, K., Wang, H., Nulty, P., Obeng, A., Müller, S., & Matsuo, A. (2018). Quanteda: An r package for the quantitative analysis of textual data. *Journal of Open Source Software*, *3*(30), 774. https://doi.org/10.21105/joss.00774

Boumans, J. W., & Trilling, D. (2016). Taking Stock of the Toolkit: An overview of relevant automated content analysis approaches and techniques for digital journalism scholars. *Digital Journalism*, *4*(1), 8–23. https://doi.org/10.1080/21670811.2015.1096598

Chan, C., Schatto-Eckrodt, T., & Gruber, J. B. (2024). What makes computational communication science (ir)reproducible? *Computational Communication Research*. https://doi.org/10.5117/CCR2024.1.5.CHAN

Grimmer, J., & Stewart, B. M. (2013). Text as Data: The Promise and Pitfalls of Automatic Content Analysis Methods for Political Texts. *Political Analysis*, *21*(3), 267–297. https://doi.org/10.1093/pan/mps028

Gruber, J. B. (2023). *cookiemonster: Your friendly solution to managing browser cookies in R*. https://github.com/JBGruber/cookiemonster

Gruber, J. B., Atteveldt, W. van, & Welbers, K. (2023). *Sharing is caring (about research): Three avenues for sharing (copyrighted) text collections and the need for non-consumptive research*. https://opted.eu/fileadmin/user_upload/k_opted/OPTED_Deliverable_D7.6.pdf

Hennesy, C., & Samberg, R. (2019). Law and literacy in non-consumptive text mining: Guiding researchers through the landscape of computational text analysis. *Copyright Conversations: Rights Literacy in a Digital World*. https://escholarship.org/uc/item/55j0h74g

Laurer, M., Van Atteveldt, W., Casas, A., & Welbers, K. (2024). Less Annotating, More Classifying: Addressing the Data Scarcity Issue of Supervised Machine Learning with Deep Transfer Learning and BERT-NLI. *Political Analysis*, *32*(1), 84–100. https://doi.org/10.1017/pan.2023.20

Weaver, D. A., & Bimber, B. (2008). Finding News Stories: A Comparison of Searches Using Lexisnexis and Google News. *Journalism & Mass Communication Quarterly*, *85*(3),

515–530. https://doi.org/10.1177/107769900808500303

Wickham, H. (2022). *Rvest: Easily harvest (scrape) web pages*. https://CRAN.R-project.org/package=rvest

Wickham, H. (2024). *httr2: Perform HTTP requests and process the responses*. https://CRAN.R-project.org/package=httr2

Wickham, H., Bryan, J., Barrett, M., & Teucher, A. (2023). *Usethis: Automate package and project setup*.