R Packages to Aid in Handling Web Access Logs

by Oliver Keyes, Bob Rudis, Jay Jacobs

Abstract Web access logs contain information on HTTP(S) requests and form a key part of both industry and academic explorations of human behaviour on the internet - explorations commonly performed in R. In this paper we explain and demonstrate a series of packages designed to efficiently read in, parse and munge access log data, allowing researchers to handle it easily.

Introduction

The rise of the World Wide Web over the last few decades has made it dramatically easier to access and transfer data, and R (R Core Team, 2015) boasts abundant functionality when it comes to taking data *from* the web. Base R itself has simple file downloading and page reading capabilities, through the `download.file` and `readLines` functions, and additional functionality is made available for handling web-accessible data through packages such as **httr** (Wickham, 2015).

Data *on* the web is not, however, the only kind of web data that interests researchers; web traffic is, in and of itself, an interesting data source. Access logs - records of connections between users and a web server - are an asset and resource for people studying everything from user behaviour on the internet (Halfaker et al., 2014), to website performance (Ryckbosch and Diwan, 2014), to information security (Bhingarkar and Shah, 2015).

As a statistically-oriented programming language, R is commonly used by these same researchers for data analysis, testing and reporting - but it lacks tools designed for the kinds of data sources and formats that are encountered when dealing with access logs. In this article we review the use cases for particular operations over web data, the limitations in base R when it comes to performing those operations, and a suite of R packages designed to overcome them: **webreadr** (Keyes, 2015), designed for reading access logs in, **urltools** (Keyes et al., 2015a), for URL manipulation, **iptools** (Rudis and Keyes, 2015) for handling IP addresses, and **rgeolocate** (Keyes et al., 2015b) for direct IP geolocation.

Reading access logs

The first task with any data analysis is to read the data into R so that it can be manipulated. With access logs this is slightly complicated by the fact that there is no one standard for what a log should look like; instead, there are multiple competing approaches from different software platforms and eras. These include the Common Log Format (CLF), the confusingly-named Combined Log Format, and formats used by individual, commonly-used software platforms - such as the custom format for the Squid internet caching software, and the format used by Amazon Web Services (AWS).

The difference between formats can easily be shown by looking at how timestamps are represented:

Table 1: Timestamps in Common Access Log Formats

Log Type	Timestamp Columns	Timestamp Format
Common Log Format	1	10/Oct/2000:13:55:36 -0700
Combined Log Format	1	26/Apr/2000:00:23:48 -0400
Squid	1	1286536309.450
AWS	2	2014-05-23 01:13:11

With four log types, we have three different timestamp formats - and that's only one of the many columns that could appear. These logs also vary in whether they specify quoting fields (or sanitising unquoted ones), the columns they contain and the data each column contains in turn.

To make reading access logs into R as easy as possible we created the **webreadr** package. This contains user-friendly equivalents to `read.table` for each type of log, detecting the fields that should appear, converting the timestamps into POSIX objects, and merging fields or splitting fields where necessary. The package contains four core functions, one for each form of log: `read_clf` for the Common Log Format, `read_combined` for the Combined Log Format, and `read_squid` and `read_aws` for Squid and AWS formats respectively. Each one abstracts away the complexity of specifying column names and formats, and instead allows a researcher to read a file in with a minimal amount of work.

As the name suggests, it is built not on top of base R but on top of the **readr** (Wickham and Francois) package, allowing us to take advantage of substantial speed improvements that package's base functions have over base R. These improvements can be seen in the visualisation below, which uses **microbenchmark** (Mersmann, 2014) to compare 100 reads of a 600,000-line "squid" formatted file with webreadr to the same operation performed in base R:

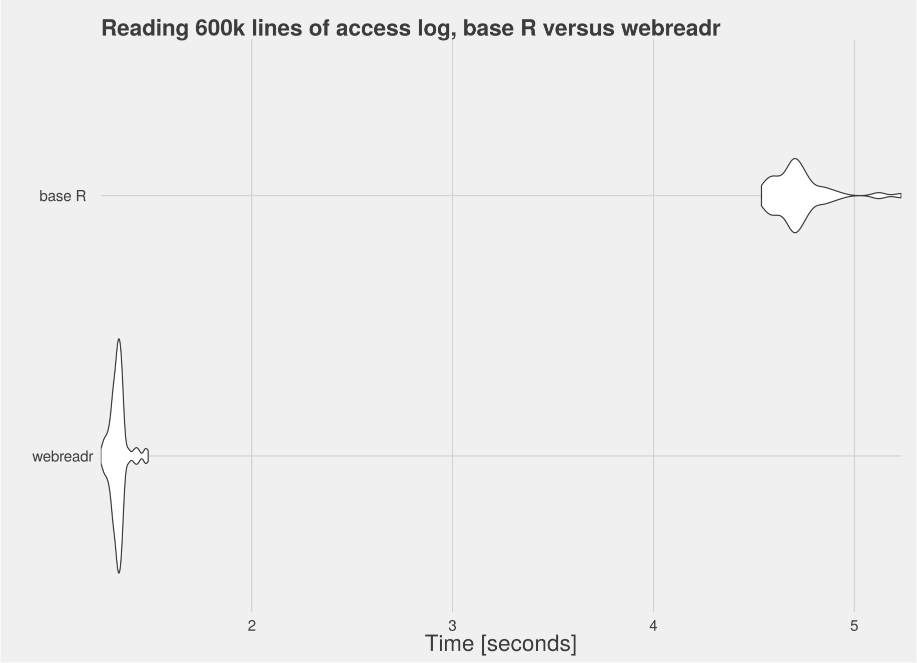

Figure 1: Results of microbenchmark run: read_squid versus base-R equivalent code

webreadr takes 1.2 seconds to read this file in at a minimum, and 1.48 at maximum, with a median of 1.33. This is consistently 3.5-6 times faster than the equivalent base R functionality - and is also, as explained, far easier to use.

Decoding and parsing URLs

Within access logs, URLs appear - both to describe the web asset or page that the user requested and where the user came from. These fields are usually named url and referer respectively.

Decoding

Both values can be percent-encoded, allowing them toinclude characters that are valid but reserved by the URL specification as having special meanings ("reserved characters"). A '`#`' symbol, for example, is encoded as '`%23`': a percentage symbol, followed by the byte-code for that character.

The encoding of reserved characters is useful, since it means that URL paths and queries can contain a vast array of values - but it makes data analysis tougher to do. Examples of common data analysis or cleaning operations that become more difficult are:

1. **Aggregation**. Aggregating URLs together is useful to identify, for example, the relative usage and popularity of particular pages in your data - but it becomes tougher if encoding is inconsistent, because two URLs could hold the same value but *look* very different.
2. **Value selection**. With text-based data, regular expressions are a common way of filtering or selecting entries that meet particular conditions, but things become fuzzy when you have to look not just for particular characters (a space, say) but also the encoded values (%20).
3. **Exploratory data analysis** (EDA). EDA is a common initial step to investigate a data set, examining the variables and values it contains and whether they meet a researcher's expectations - but on a practical basis it becomes difficult when the values aren't human-readable.

The solution is to be able to consistently decode URLs, which makes URL-based data far easier to analyse. Base R does contain a function, `URLdecode`, for doing just that, but is neither vectorised nor based on compiled code; with large datasets it takes an extremely long time.

To solve this common problem in analysing request logs, the **urltools** package was created. This contains a function, `url_decode`, which decodes URLs - but relies on vectorised, compiled code.

Benchmarking the two approaches against each other shows that the **urltools** implementation is approximately 60-70 times faster over large datasets. Again using **microbenchmark**, if we compare the vectorised decoding of 1,000,000 URLs with **urltools** against `URLdecode` in a `vapply` loop, we see a 60-70 times speed improvement:

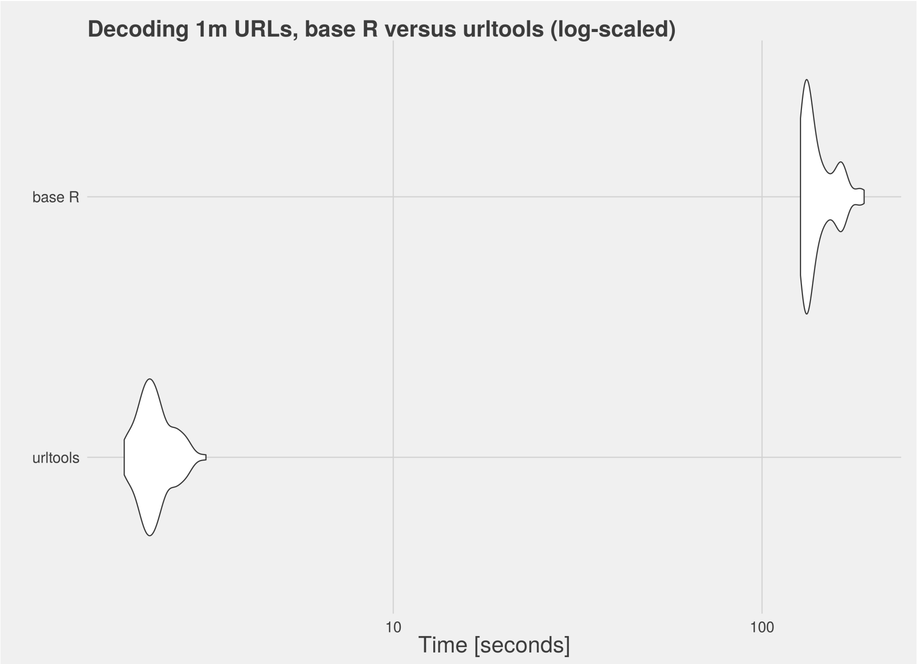

Figure 2: Results of microbenchmark run: `url_decode` versus base-R equivalent code

Parsing

The standard for URLs (Berners-Lee et al., 1994) divides them into a heirarchical sequence of components - the scheme (‘`http`’), host (‘`en.wikipedia.org`’), port (‘`800`’), path (‘`wiki/Main_Page`’) and searchpart, or parameters (‘`action=edit`’). Together, these make up a URL (‘`http://en.wikipedia.org:800/wiki/Main_Page?action=edit`’).

Parsing URLs to isolate and extract these components is a useful ability when it comes to exploring request logs; it lets a researcher pick out particular schemes, paths, hosts or other components to aggregate by, identifying how users are behaving and what they are visiting. It makes anonymising data - by removing, for example, the parameters or path, which can contain relatively unique information - easier.

Base R does not have native code to parse URLs, but the **httr** package (Wickham, 2015) contains a function, `parse_url`, designed to do just that. Built on R’s regular expressions, this function is not vectorised, does not make use of compiled code internally, and produces a list rather than data.frame, making looping over a set of URLs to parse each one a time-consuming and awkward experience. This is understandable given the intent behind that function, which is to decompose individual URLs within the context of making HTTP requests, rather than to analyse URLs *en masse*.

urltools contains `url_parse` - which does the same thing as the equivalent **httr** functionality, but in a vectorised way, relying on compiled code, and producing a data.frame. Within the context of parsing and processing access logs, this is far more useful, because it works efficiently over large sets: httr’s functionality, which was never designed with vectorisation in mind, does not. Indeed, benchmarking showed that `url_parse` is approximately 570 times faster than httr’s equivalent function:

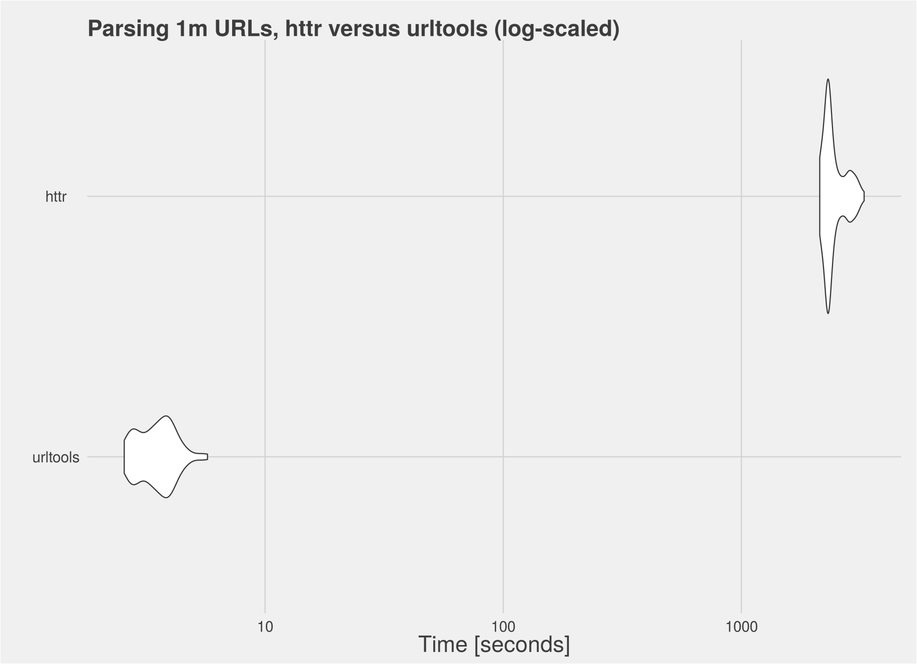

Figure 3: Results of microbenchmark run: `url_parse` versus httr's equivalent code

A vector of URLs passed into `url_parse` produces a data.frame, with one column for each of the IETF-supported components, and empty strings representing components that could not be found. Additionally, influenced by the style of the **lubridate** package (Grolemund and Wickham, 2011), **urltools** contains functions to get or *set* individual components:

```
# Load urltools and construct a URL
library(urltools)
url <- "http://www.google.com/"

# Get the scheme
scheme(url)

#> [1] "http"

# Set the scheme and observe the modification to the resulting URL
scheme(url) <- "https"
url

#> [1] "https://www.google.com/"
```

As a result of this functionality, **urltools** makes URL manipulation faster, easier and far more accessible, reducing the burden associated with munging access logs or similar datasets and allowing a researcher to get to the statistical analysis faster.

IP manipulation

Access logs also contain IP addresses - unique numeric values that identify a particular computer or network in the context of the internet. Working with these values allows an analyst to validate their data (by checking for false or spoof addresses) and is a necessary prerequisite to the use of some IP geolocation systems (covered later), and extract a limited amount of metadata from the values themselves.

Based around the *Boost ASIO* C++ library (boo), **iptools** is a package designed for this kind of IP manipulation (and more). Built around combined code, it is extremely fast (for that code that does not rely on internet connections) and boasts a range of features.

One of the most crucial is `ip_classify`, which, when provided with a vector of IP addresses, identifies whether they follow the IPv4 or IPv6 standard. As a side-effect of this, it can be used to identify if IP addresses are invalid, or spoofed, prior to further work based on the assumption that they are correct:

```
# Load iptools and construct a vector of IPs
library(iptools)
ip_addresses <- c("192.168.0.1", "2607:f8b0:4006:80b::1004", "Chewie")
ip_classify(ip_addresses)
```

```
#> [1] "IPv4"    "IPv6"    "Invalid"
```

iptools also contains (for the same purpose) code to identify the actual client's IP address. Access requests usually contain not only an IP address but a *X-Forwarded-For* field - a field identifying which *other* IP addresses the request passed through, if the user who made the request is using some kind of proxy. If a user *has* used a proxy, the contents of the IP address field won't actually be them - it will be the last proxy the request went through before getting to the server logging the requests. The actual IP address of the user will instead be the earliest value of the X-Forwarded-For field.

The solution is to be able to identify the 'real' IP address, by checking:

1. Whether the X-Forwarded-For field contains any values;
2. Extracting the earliest non-invalid IP address in that field's values if so, and the contents of the IP address field if not.

With the `xff_extract` function, you can do just that:

```
ip_address <- "192.168.0.1"
x_forwarded_for <- "foo, 193.168.0.1, 230.98.107.1"
xff_extract(ip_address, x_forwarded_for)
```

```
#> [1] "192.168.0.1"
```

This returns the IP address field value if X-Forwarded-For is empty, and otherwise splits the X-Forwarded-For field and returns the earliest valid IP address. It is fully vectorised and highly useful for analysis predicated on IP addresses being valid, such as geolocation - without this kind of resolution, what you're actually geolocating might be your own servers.

Other operations supported by **iptools** include the conversion of IP addresses from their standard *dotted-decimal* form to a numeric form, the extraction of IP ranges (and their contents), resolving IP addresses to hostnames, and a series of datasets covering the IPv4 registry and port database.

The limitation of the package is that it does not yet work on Windows - building ASIO on that platform is a uniquely frustrating experience - and that some operations do not yet support IPv6 (since they require the storage of numbers bigger than R can currently handle).

Geolocation

As a side-effect of how IP addresses tend to be assigned - in geographic blocks, to individual machines or to local networks - they can be used to geolocate requests, identifying where in the world the request came from, sometimes down to the level of individual post codes or pairs of latitude/longitude coordinates.

This is tremendously useful in industry, where the geographic reach of a service has substantial implications for its viability and survivability, and in academia, where the locality of internet-provided information and the breadth of internet access are active concerns and areas of study (Sen et al., 2015).

Many services and databases exist for extracting geographic metadata from IP addresses. One of the most common is the service provided by MaxMind, which has both proprietary and openly-licensed databases, in binary and comma-separated formats. The free databases have been used by various web APIs, which makes the data they contain accessible from R. Unfortunately, dependence on web APIs means that handling large numbers of IP addresses can be very slow (they tend to be designed to only accept one IP address at a time, and may contain throttling beyond that) and has privacy concerns, since it essentially means sending user IP addresses to a third party. And even without these issues, there are no convenient wrappers for those APIs available on CRAN: users have to hand-roll their own.

With these concerns in mind, we wrote the **rgeolocate** package. Through **httr** this contains convenient, vectorised bindings to various web services that provide geographic metadata about IP addresses. More importantly, using the **Rcpp** package to integrate C++ and R, **rgeolocate** also features a direct, compiled binding to the MaxMind API. This means that local binary databases can also be queried, which is far faster and more robust than web-based equivalents and avoids the privacy concerns associated with transmitting users' IP addresses externally.

The MaxMind API requires a paid or free binary database - one of which, for country-level IP resolution, is included in **rgeolocate** - and allows you to retrieve the continent, country name or ISO code, region or city name, tzdata-compatible timezone, longitude, latitude or connection type of a particular IP address. Multiple fields can be selected (although which are available depends on the type of database used), and results are returned in a data.frame:

```
# Load rgeolocate
library(rgeolocate)

# Find the rgeolocate-provided binary database
geo_file <- system.file("extdata","GeoLite2-Country.mmdb", package = "rgeolocate")

# Put together some example IP addresses
ip_addresses <- c("174.62.175.82", "196.200.60.51")

# Geolocate
rgeolocate::maxmind(ip_addresses, geo_file, fields = c("continent_name", "country_code"))

#>   continent_name country_code
#> 1  North America           US
#> 2         Africa           ML
```

Direct speed comparisons aren't possible, since the functionality it provides is not (to the authors' knowledge) replicated in other R packages, but it is certainly faster (and more secure) than internet-dependent alternatives.

Conclusions and further work

In this research article we have demonstrated a set of tools for handling access logs during every stage of the data cleaning pipeline - reading them in with **webreadr**, decoding, manipulating and extracting value from URLs with **urltools**, and retrieving geographic metadata from IP addresses with **iptools** and **rgeolocate**. We have also demonstrated the dramatic speed improvements in using these tools in preference to existing methods within R.

Further work - integrating more sources of geolocation information within **rgeolocate**, supporting UTF-8 URL decoding within **urltools**, and increasing IPv6 support in **iptools** - would make these packages even more useful to Human-Computer Interaction researchers and other specialists who rely on access logs as primary data sources. Even without that functionality, however, this suite of packages is a dramatic improvement upon the status quo.

Bibliography

BOOST C++ Libraries. URL http://www.boost.org. http://www.boost.org. [p4]

T. Berners-Lee, L. Masinter, and M. McCahill. Uniform resource locators (url). RFC 1738, December 1994. URL https://tools.ietf.org/html/rfc3986. [p3]

A. S. Bhingarkar and B. D. Shah. A survey: Securing cloud infrastructure against edos attack. In *Proceedings of the International Conference on Grid Computing and Applications (GCA)*, page 16. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2015. [p1]

G. Grolemund and H. Wickham. Dates and times made easy with lubridate. *Journal of Statistical Software*, 40(3):1–25, 2011. URL http://www.jstatsoft.org/v40/i03/. [p4]

A. Halfaker, O. Keyes, D. Kluver, J. Thebault-Spieker, T. T. Nguyen, K. Shores, A. Uduwage, and M. Warncke-Wang. User session identification based on strong regularities in inter-activity time. *CoRR*, abs/1411.2878, 2014. URL http://arxiv.org/abs/1411.2878. [p1]

O. Keyes. *webreadr: Tools for Reading Formatted Access Log Files*, 2015. URL http://CRAN.R-project.org/package=webreadr. R package version 0.3.0. [p1]

O. Keyes, J. Jacobs, M. Greenaway, and B. Rudis. *urltools: Vectorised Tools for URL Handling and Parsing*, 2015a. URL http://CRAN.R-project.org/package=urltools. R package version 1.3.2. [p1]

O. Keyes, D. Schmidt, D. Robinson, I. Maxmind, and P. Gloor. *rgeolocate: IP Address Geolocation*, 2015b. URL http://CRAN.R-project.org/package=rgeolocate. R package version 0.5.0. [p1]

O. Mersmann. *microbenchmark: Accurate Timing Functions*, 2014. URL `http://CRAN.R-project.org/package=microbenchmark`. R package version 1.4-2. [p2]

R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2015. URL `https://www.R-project.org/`. [p1]

B. Rudis and O. Keyes. *iptools: Manipulate, Validate and Resolve IP Addresses*, 2015. URL `http://CRAN.R-project.org/package=iptools`. R package version 0.3.0. [p1]

F. Ryckbosch and A. Diwan. Analyzing performance traces using temporal formulas. *Software: Practice and Experience*, 44(7):777–792, 2014. [p1]

S. W. Sen, H. Ford, D. R. Musicant, M. Graham, O. S. Keyes, and B. Hecht. Barriers to the localness of volunteered geographic information. *Proceedings of the 2015 ACM Conference on Human Factors in Computing*, 2015. [p5]

H. Wickham. *httr: Tools for Working with URLs and HTTP*, 2015. URL `http://CRAN.R-project.org/package=httr`. R package version 1.0.0. [p1, 3]

H. Wickham and R. Francois. *readr: Read Tabular Data*. URL `https://github.com/hadley/readr`. R package version 0.1.1.9000. [p1]

Oliver Keyes
Wikimedia Foundation
line 1
line 2
`author1@work`

Bob Rudis
Rapid7
line 1
line 2
`author2@work`

Jay Jacobs
Magical Python Land
line 1
line 2
`author3@work`