Accessing the Public Library of Science (PLOS) in R using rplos

In this chapter we look at the use of the rplos[[1]](#footnote-1) package from rOpenSci[[2]](#footnote-2) to access the scientific literature from the Public Library of Science[[3]](#footnote-3) using the PLOS Search API[[4]](#footnote-4).

The Public Library of Science (PLOS) is the main champion of open access peer reviewed scientific publications and has published somewhere in the region of 140,000 articles. These articles are a fantastic resource. PLOS includes the following titles.

* PLOS ONE
* PLOS Biology
* PLOS Medicine
* PLOS Computational Biology
* PLOS Genetics
* PLOS Pathogens
* PLOS Neglected Tropical Diseases
* PLOS Clinical Trials ()
* PLOS Collections (collections of articles)

PLOS is important because it provides open access to the full text of peer reviewed research. For researchers interested in working with R, rplos and its bigger sister package, the rOpenSci fulltext package[[5]](#footnote-5) are very important tools for gaining access to research.

This article is part of work in progress for the WIPO Manual on Open Source Patent Analytics. The Manual is intended to introduce open source analytics tools to patent researchers in developing countries and to be of wider use to the science and technology research community. An important part of patent research is being able to access and analyse the scientific literature.

This article makes no assumptions about knowledge of R or programming. rplos is a good place to start with learning how to access scientific literature in R using Application Programming Interfaces (APIs). Because rplos is well organised and the data is very clean it is also a good place to learn some of the basics of working with data in R. This provides a good basis for working with the ROpenSci fulltext package[[6]](#footnote-6). fulltext allows you to retrieve scientific literature from multiple data sources and we will deal with that next.

We will also use this as an opportunity to introduce some of the popular packages for working with data in R, notably the family of packages for tidying and wrangling data developed by Hadley Wickham at RStudio (namely, plyr, dplyr, stringr and tidyr). We will only touch on these but we include then as everyday working packages that you will find useful in learning more about R.

The first step is to make sure that you have R and RStudio.

## Install R and RStudio

To get up and running you need to install a version of R for your operating system. You can do that from here[[7]](#footnote-7). Then download RStudio Desktop for your operating system from here[[8]](#footnote-8) using the installer for your system. Then open RStudio.

## Create A Project

Projects are the best way of organising your work in RStudio. To create a new project select the dropdown menu in to top right where you see the blue R icon. Navigate to where you want to keep your R materials and give your project a name (e.g. rplos). Now you will be able to save you work into an rplos project folder and R will keep everything together when you save the project.

## Install Packages

First we need to install some packages to help us work with the data. This list of packages are common "go to" packages for daily use.

install.packages("rplos") #the main event  
install.packages("readr") #for reading data  
install.packages("plyr") #for wrangling data  
install.packages("dplyr") #for wrangling data  
install.packages("tidyr") #for tidying data  
install.packages("stringr") #for manipulating strings  
install.packages("tm") #for text mining  
install.packages("XML") #for dealing with text in xml

Then we load the libraries. Note that rplos will install and load any other packages that it needs (in this case ggplot2 for graphing) so we don't need to worry about that.

library(rplos)  
library(readr)  
library(plyr) # load before dplyr to avoid errors  
library(dplyr)  
library(tidyr)  
library(stringr)  
library(tm)  
library(XML)

Next let's take a look at the wide range of functions that are available for searching using rplos by moving over to the Packages tab in RStudio and clicking on rplos. A very useful tutorial on using rplos can be found here[[9]](#footnote-9) and can be cited as "Scott Chamberlain, Carl Boettiger and Karthik Ram (2015). rplos: Interface to PLOS Journals search API. R package version 0.5.0 <https://github.com/ropensci/rplos>". If you are already comfortable working in R you might want to head to that introductory tutorial as this article contains a lot more in the way of explanation. However, we will also add some new examples and code for working with the results to add to the resource base for rplos.

## Key functions in rplos

R is an object oriented language meaning that it works on objects such as a vector, table, list, or matrix. These are easy to create. We then apply functions to the data from base R or from packages we have installed for particular tasks.

* searchplos(), the basic function for searching plos
* plosauthor(), search on author name
* plostitle(), search the title
* plosabstract(), search the abstract
* plossubject(), search by subject
* citations(), search the PLOS Rich Citations[[10]](#footnote-10)
* plos\_fulltext(), retrieve full text using a DOI
* highplos(), highlight search terms in the results.
* highbrow(), browse search terms in a browser with hyperlinks.

Functions in R take (accept) arguments which are options for the type of data we want to obtain when using an API or the calculations that we want to run on the data. For rplos we will mainly use arguments setting out our search query, the fields that we want to search, and the number of results.

If you are new to R this will typically takes the form of a short piece of code that is structured like this.

newobject <- function(yourdata, argument1, argument2, etc)

A new object is likely to be a table or list containing data. the sign <- gets or passes the results of the function (such as seachplos) to the new object. To specify what we want we first include our data (yourdata) and then one or more arguments which control what we get, such as the number of records or the title etc.

## Data Fields in rplos

There are quite a number of fields that can be searched with rplos or used to refine a search. We will only use a few of them. To see the range of fields type plosfields into the console and press Enter.

plosfields

For example, if we wanted to search the title, abstract and conclusions we would use these fields in building the query (see below). If we wanted to search everything but those fields we would probably use body. If we wanted to retrieve the references then we would include reference in the fields. In rplos a field is denoted by fl = with the fields in quotes such as fl = "title" and so on as we will see below.

## Basic Searching using searchplos(), Navigating and Exporting Data

searchplos() is the basic rplos search function and returns a list of document identifiers (DOIs) or other data fields. The basic search result is a set of DOIs that can be used for further work. To get help for a function, or to find working examples, use ? in front of the function in the console:

`?`(searchplos)

This will bring up the help page for that function with a description of the arguments that are available and with examples at the bottom of the page.

The examples are there to help you. In rplos they presently focus on the use of single search terms such as ecology. However, as we will see below, it is possible to use phrases in searching and to use multiple terms. There are quite a number of arguments (options) available for refining the results and we will include some of these in the examples.

The author of this article is a big fan of pizza. So, in the first example we will carry out a simple search for the term pizza and then specify the results we want to see using the argument fl = (for fields) and the number of results that we want to see using limit = 20. In specifying the fields we will use c() to combine them together.

a <- searchplos(q = "algeria", fl = c("id","publication\_date", "title", "abstract"), limit = 20)

What searchplos() has done in the background is to send a request to the PLOS API to bring back the id, publication\_date, title and abstract for 20 records across the PLOS journals. To see the results type:

p

Results in R are stored in objects (in this case the object is a list). To see the type of object in R use:

class(p)

When working with R it is generally more useful to understand the structure of the data so that you can work out how to access it. That can be done using str() for structure. This is one of the most useful functions in R and well worth writing down.

str(p)

The results might seem a little confusing at first but what this is telling us is that we have an R object that is a list consisting of two components. The first is an item called meta that reports the number of records found and the type of object (a data.frame). The second is data which contains the information on the two results in the form of a data frame (basically a table) containing the id, date, title and abstract information that we asked PLOS for.

Note that the list contains a marker $ for the beginning of the two lists with the data they contain appearing as ..$ signifying that they are nested under meta or data. This hierarchy helps us with accessing the data using subsetting in R. For example, if we wanted to access the meta data (and we do) we can use the following:

p$meta

That will just print the full meta data entries. If we wanted to just access the number of records (num$Found) then we would extend this a little by moving to that position in the hierarchy with:

p$meta$numFound

That will print out just the number of records returned by our search. An alternative way of subsetting is to use the "" and "" and the numeric position in the list. In Hands on Programming with R[[11]](#footnote-11) Garrett Grolemund compares this to a train with numbered carriages where ]\* selects the train carriage e.g. 1]\* and 1]] selects the contents of carriage number 1. We don't need to worry about this but it is very helpful as a way of remembering the difference. For example the following selects the contents of the first item in our list (meta):

p[[1]]

and is the same as p$meta. While:

p[[1]][[1]]

is the same as p$meta$numFound.

Subsetting the data by its numeric position rather than its name makes life much easier when working with lists with lots of items. As we will see below, when applying a function to a list with multiple items we can also use "", 2. This will retrieve the second item in each of our line of train carriages.

Another useful tip for navigating the data in RStudio is using autocomplete. Try typing the following into the console.

p #type me in the console, do not cut and paste

When we type the $ a popup will appear and display two entries as tables for meta and data. Click on meta, then add another $ sign at the end. It will now display three items in purple (for vectors). Select numFound and hey presto! As you work with RStudio you will notice that when you start to type a function name, lists of names will start to pop up. Type search into the console but do not press enter and wait a moment. A list with three items should pop up with search {base}, searchpaths {base}, and searchplos {rplos}. This is really helpful because it saves a lot of typing. As you become more familiar with R it also helpfully displays what a function does and a reminder of its arguments. The soft brackets around {base} indicate the package where the function can be found (this can be useful for discovering functions when you get stuck).

Finally, you can also see the items in your project in the Environment pane. Click on the blue arrow for p in the Environment pane under Values and you will see the structure of the data in p and some of its content.

### Creating a New Object and Writing to File

Ok so we have a list with some results containing meta and data. We now want to export data to a .csv file that we can work with in Excel or another programme.

While we will want to make a note of the total number of results in meta, what we really want will be in data. We can simply create a new object using the code above and assign it to a name using <-. Note that there is no space here and < - will not work.

dat <- p$data

If we look at the class of this object (class(dat)) we now have a data.frame (a table) that we can write to a .csv file to use later. We can do this easily using write.csv() and start by naming the object we want to write (dat) and then giving it a file name. Because we created an rplos project in RStudio earlier (didn't we), the file will be saved into the project folder. If you didn't create a project or want to check the directory then use:

getwd()

This will show your current working directory. If you do not see the name of your rplos project then copy the full file path so that it looks something like this (don't forget the "" around the path):

setwd("/Users/pauloldham/Desktop/open\_source\_master/rplos")

Ok, we now know where we are. So, let's save the file.

write.csv(dat, "dat.csv", row.names = FALSE)

If we open this up in Excel or Open Office Calc then we will see two blank entries in the abstract fields. Blank cells can create calculation problems. Inside R we can handle this by filling in the blanks with NA as follows 2]\*. In this case we are subsetting into dat and then asking R to identify those cells that exactly match == with "". We then fill those cells in dat with NA (for Not Available).

datdat[==""] <- NA

We can then simply write the file as before. If we wanted to remove the NAs we have just introduced then we could use write.csv(dat, "dat.csv", row.names = FALSE, na = "") which will convert them back to blank spaces.

A faster way to deal with writing files is to use the recent readr package as this will not add row numbers to exported files. Here we will use the write\_csv() function.

write\_csv(dat, "dat.csv")

The advantage of readr is that it is fast and does not require the same number of arguments as the standard write.csv such as specifying row names or with read.csv specifying stringsAsFactors = FALSE.

Finally, if we wanted to write the entire list p, including meta to file then we could use:

write.csv(p, "p.csv", row.names = FALSE)

We have now retrieved some data containing pizza through the PLOS API using rplos and we have written the data to a file as a table we can use later. We will now move on to some more sophisticated things we can do with rplos.

## Limit by journal

As we have seen above, PLOS contains 7 journals and in rplos the results for a search can be limited to specific journals such as PLOS ONE or PLOS Biology. Note that the short journal names appear to use the old format for PLOS consisting of mixed upper and lowercase characters (e.g. PLoSONE not PLOSONE). A nice easy way to find the short journal names is to use:

journalnamekey()

Here we will limit the search to PLOS ONE by adding fq = to the arguments and then the cross\_published\_journal\_key argument. Note that the fq= argument takes the same options as fl=. But, fq = filters the results returned by PLOS to only those specified in fq =.

pizza <- searchplos(q = "pizza", fl = c("id", "publication\_date", "title", "abstract"), fq = 'cross\_published\_journal\_key:PLoSONE', start = 0, limit = 20)  
head(pizza$data)

We have retrieved 20 records here using limit = 20 (the default is 10). It is generally a good idea to start with a small number of results to test that we are getting what we expect back rather than lots of irrelevant data. What if we wanted to retrieve all of the results? Here we will need to do a bit more work using the meta field.  
##Obtaining the full number of results

One way to do this is to take our original number of results and then subset in to the data and create a new object containing the value for the number of records in numFound. Note that the number of records for a particular query below may well have gone up by the time that you read this article.

r <- pizza$meta$numFound

To run a new search we can now insert r into the limit = value. This will be interpreted as the numeric value of r (210).

pizza <- searchplos(q = "pizza", fl = c("id", "publication\_date", "title", "abstract"), fq = 'cross\_published\_journal\_key:PLoSONE', start = 0, limit = r)  
head(pizza$data)

An alternative way of doing this is to make life a bit easier for ourselves by first running our query and setting the limit as limit = 0. This will only return the meta data. We then add the subset for number found to the end of the code as $meta$numFound. That will pull back the value (210) directly.

r <- searchplos(q = "pizza", fq = "cross\_published\_journal\_key:PLoSONE", limit = 0)$meta$numFound

We can then run the query again using the value of r in limit = :

pizza <- searchplos(q = "pizza", fl = c("id", "publication\_date", "title", "abstract"), fq = 'cross\_published\_journal\_key:PLoSONE', start = 0, limit = r)   
head(pizza$data)

## Obtaining the number of records across PLOS Journals

That has returned the full 210 results for PLOS ONE. We could attempt to make life even easier by first getting the results across all PLOS journals. We do this by removing the fq = argument limiting the data to PLOS ONE and saving the result in and object we will call r1. Note that the number of records will probably have gone up by the time you read this.

r1 <- searchplos("pizza", limit = 0)$meta$numFound

This produces 244 results. What happens now if we run our original query using the value of r1 (244 records) but limiting the results only to PLOS ONE?

pizza <- searchplos(q = "pizza", fl = c("id", "publication\_date", "title", "abstract"), fq = 'cross\_published\_journal\_key:PLoSONE', start = 0, limit = r1)  
pizza$meta$numFound

The answer is that the 210 results in PLOS ONE are returned from the total of 244 across the PLOS journals. Why? The reason this works is that searchplos() initially pulls back all of the data from the PLOS API and then applies our entry in fq = as a filter. So, in reality the full 244 records are fetched and then filtered down to the 210 from PLOS ONE. In this case, this makes our lives easier because we can use the results across PLOS journals and then restrict the data.

## Writing the results and using a codebook

We now have a total of 210 results for pizza. We can simply write the results to a .csv file.

write.csv(pizza, "plosone\_pizza.csv", row.names = FALSE)

As this illustrates, it is very easy to use rplos() and rapidly create a file that can be used for other purposes.

When working in R you will often create multiple tables and take multiple steps. To keep track of what you do it is a good idea to create a text file as a codebook. Use the codebook to note down the important steps you take. The idea of a codebook is taken from Jeffrey Leek's Elements of Data Analytic Sytle[[12]](#footnote-12) which provides a very accessible introduction to staying organised. To create a codebook in RStudio simply use File > New File > Text File. This will open a text file that can be saved with your project. The codebook allows you to recall what actions you performed on the data months or years later. It also allows others to follow and reproduce your results and is important for reproducible research[[13]](#footnote-13).

## Proximity Searching

We will typically want to carry out a search by first retrieving a rough working set of results to get a feel for the data and then experimenting until we are happy with the data to noise ratio (see this article[[14]](#footnote-14) for an example).

In thinking about ways to refine our search criteria we can also use proximity searching. Proximity searching focuses on the distance between words that we are interested in. To read more about this use ?searchplos in the console and scroll down to example seven in the help list. We reproduce that example here using the words synthetic and biology as our terms.

We can set the proximity of terms using tilde ~ and a value. For example, ~15 will find instances of the terms synthetic and biology within 15 words of each other in the full texts of PLOS articles.

searchplos(q = "everything:\"synthetic biology\"~15", fl = "title", fq = "doc\_type:full")

Note that while synthetic and biology appear inside quotes (suggesting they are a phrase to be searched) in reality the API will treat this as synthetic AND biology. That is, the query will look first for documents that contain the words synthetic AND biology and then for those cases where the words appear within 15 words of each other. In this case we get 1,684 results across PLOS (everything) and full texts (fq = "doc\_type:full) as we can see from this code.

searchplos(q = "everything:\"synthetic biology\"~15", fl = "title", fq = "doc\_type:full")$meta$numFound

We can narrow the search horizon to ~1 to capture those cases where the terms appear next to each other (within 1 word either to the left or the right) which produces 1001 results.

searchplos(q = "everything:\"synthetic biology\"~1", fl = "title", fq = "doc\_type:full")$meta$numFound

This is actually about 10 records higher than the total returned on an exact match for the phrase suggesting that there could be cases of "biology synthetic" or other issues (such as punctuation) or API performance that account for the variance. As noted in the searchplos() documentation:

"Don't be surprised if queries you perform in a scripting language, like using rplos in R, give different results than when searching for articles on the PLOS website. I am not sure what exact defaults they use on their website."

As a result, it is a good idea to try different approaches. Even if it is not possible to get to the bottom of any variance it is very useful to note it down in your codebook to highlight the issue to others who may try and repeat your work.

It is also important to emphasise that when using rplos() it is possible to return a fragment of the text with the highlighted terms using highplos() and the hl.fragsize argument to set the horizon for the fragment of text around the search. This is particularly useful for text mining.

In many cases the most useful information comes from searching using phrases and multiple terms. Unlike words, phrases can articulate concepts. This generally makes them more useful than single words for searching for information.

## Searching Using Multiple Phrases

To search by phrases we start by creating an object containing our phrases and put the phrases inside double quotation marks. If we do not use double quotation marks the search will look for documents containing both words rather than the complete phrase (e.g. synthetic AND biology rather than "synthetic biology"). Note that the code below will display "" as "" but you don't need to enter the \.

We will use the search query developed in this PLOS ONE article on synthetic biology[[15]](#footnote-15) in this example and retrieve the id, data, author, title and abstract across the PLOS journals.

First we create the search query. Note that we use c(), for combine, to combine the list of terms into a vector inside the object called s.

s <- c("\"synthetic biology\"", "\"synthetic genomics\"", "\"synthetic genome\"",   
 "\"synthetic genomes\"")

We now want to get the maximum number of results returned by one of the search terms. This is slightly tricky because rplos will return a list containing four list items (one for each of our search terms). Each of those lists will contain meta and data items. What we want to do is find out which of the search terms returns the highest number of results inside meta in numFound. Then we can use that number as our limit.

This involves more than one step.

1. First we need to fetch the data.
2. Then we need to extract meta from each list.
3. Then we need to select numFound and find and return the maximum value across the lists of results.

The easiest way to do this is to create a small function that we will call plos\_records. To load the function into your Environment copy it and paste it into your console and press enter. The comments following # explain what is happening will be ignored when the function runs. When you have done this if you move over to Environment you will see plos\_records under Functions.

plos\_records <- function(q) {  
 library(plyr) #for ldply  
 library(dplyr) #for pipes, select and filter  
 lapply(q, function(x) searchplos(x, limit = 0)) %>%  
 ldply("", 1) %>% #get meta from the lists  
 select(numFound) %>% #select numFound column of meta  
 filter(numFound == max(numFound)) %>% #filter on max numFound  
 print() #print max value of numFound  
}

Now we can run the following code using s as our query (q = s) in the function. If all goes well a result will be printed in the console with the maximum number of results. It can take a few moments for the results to come back from the API.

r2 <- plos\_records(q = s)

You should now see a number around 991 (at the time of writing). Yay!

Now we can use r2 in the limit to return all of the records. We will write this in the standard way and then display a simpler way using pipes %>% below. Note that we use s as our search terms (see q = s) and we have used r2 for the limit (limit = r2). Because we are calling a chunk of data this can take around a minute to run.

Note that at each step in the code below we are creating and then overwriting an object called results. We are also naming results as the first argument in each step.

results <- lapply(s, function(x) searchplos(x, fl = c('id','author', 'publication\_date', 'title', 'abstract'), limit = r2))  
results <- setNames(results, s) #add query terms to the relevant results in the list  
results <- ldply(results, "", 2) #extract the data into a single data.frame

We can make life simpler by using pipes %>% to simplify the code. The advantage of using pipes is that we do not have to keep creating and overwriting temporary objects (see above for results). The code is also much easier to read and faster. To learn more about using pipes see this article from Sean Anderson[[16]](#footnote-16). Again the query might be a bit slow as the data is fetched back.

results <- lapply(s, function(x) searchplos(x, fl = c('id', 'author', 'publication\_date', 'title', 'abstract'), limit = r2)) %>%  
 setNames(s) %>%   
 ldply("", 2)

Pipes are a relatively recent innovation in R (see the magrittr, dplyr and tidyr packages) and most code you will see will be written in the traditional way. However, pipes make R code faster and much easier to follow. While you will need to be familiar with regular R code to follow most existing work, pipes are becoming increasingly popular because the code is simpler and has a clearer logic (e.g. do this then that).

We now have our data consisting of 1,405 records in a single data frame that we can view.

View(results)

We could now simply write this to a .csv file. But there are a number of things that we might want to do first. Most of these tasks fall into the category of wrangling and tidying up data so that we can carry on working with it in R or other software such as Excel.

## Tidying and Organising the Data

Many useful data cleaning and organisational tasks can be easily performed using the dplyr() and tidyr() packages developed by Hadley Wickham at RStudio. Other important packages include stringr()(for working with text strings), plyr() and reshape2() (general wrangling) and lubridate() (for working with dates). These packages were developed by Hadley Wickham and colleagues with the specific aim of making it easier to work with data in R in a consistent way. We will mainly use dplyr and tidyr in the examples below and a very useful RStudio cheatsheet[[17]](#footnote-17) can help you with working with dplyr and tidyr.

### Renaming a column

First we might want to tidy up by renaming a column. For example we might want to rename .id to something more meaningful. We can use rename() from dplyr() to do that (see ?rename).

results <- rename(results, search\_terms = .id)

## Filling Blank Spaces

It is good practice to fill blank cells with NA for "Not Available"" to avoid calculation problems. For example, as in the earlier example, we have some blank cells in the abstract field and there may be others somewhere else. Following this StackOverflow answer[[18]](#footnote-18) we can do this easily.

results[results == ""] <- NA

If for some reason we wanted to remove the NA values we can handle that at the time of exporting to a file (see above).

## Converting Dates

The publication\_date field is a character vector. We can easily turn this into a Date format that can be used in R and drop the T00:00:00 for time information using:

results$publication\_date <- as.Date(results$publication\_date)

## Adding columns

When dealing with dates we might want to simply split the publication\_date field into three columns for year, month and day. We can do that using separate() from tidyr.

results <- separate(results, publication\_date, c("year", "month", "day"), sep = "-",   
 remove = FALSE)

Here we have specified the data (results), the column we want to separate (results) and then the three new columns that we want to create by closing them in c() and placing them in quotes. This creates three new columns. The remove argument specifies whether we want to remove the original column (the default is TRUE) or keep it.

Because working with dates can be quite awkward (to put it mildly) it makes sense to have a range of options available to you early on in working with your data rather than having to go back to the beginning much later on.

## Add a count

One feature of pulling back literature from an API for scientific literature is that the fields tend to be character fields rather than numeric. Character vectors in R are quoted with "". This can make life awkward if we want to start counting things later on. To add a count column we can use mutate from the dplyr() package to create a new column number. number is based on assigning the value 1 to the id columns using mutate(). We are avoiding the term count because it is the name of a function count(). There are other ways of doing this but this approach points to the very useful mutate() function in dplyr for adding a new variable.

results <- mutate(results, number = sum(id = 1))

When we view results we will now see a new column number that contains the value 1 for each entry.

## Remove a column

We will often end up with more data than we want, or create more columns than we need. The standard way to remove a column is to use the trusty $ to select the column and assign it to NULL.

results$columnname <- NULL #dummy example

Another way of doing this, which can be used for multiple columns, is to use select() from dplyr (see ?select()). Select will only keep the columns that we name. We can do this using the column names or position. For example the following will keep the first 8 columns (1:8) but will drop the unnamed 9th column because the default is to drop columns that are not named. We could also write out the column names but using the position numbers is faster in this case. and

test <- select(results, 1:8)

We could also drop columns by position using the following (to remove column 5 and 6). This approach is useful when there are lots of columns to deal with.

test <- select(results, 1:4, 7:9)

An easier approach in this case is to explicitly drop columns using - and keep the others.

test <- select(results, -month, -day)

Select is also very useful for reordering columns. Let's imagine that we wanted to move the id column to the first column. We can simply put id as the first entry in select() and then the total columns to reorder.

test <- select(results, id, 1:9)

The select function is incredibly useful for rapidly organising data as we will see below.

## Arranging the Data

We might want to arrange our rows (which can be quite difficult to do in base R). The arrange() function in dplyr makes this easy and arranges a column's values in ascending order by default. Here we will specify descending desc() because we want to see the most recent publications that mention our search terms at the top.

results <- arrange(results, desc(publication\_date))

When we use View(results) we will see that the most recent data is at the top. We will also see that some of the titles towards the top are duplicates of the same article because they include all the terms in our search. So, the next thing we will want to do is to address duplicates.

## Dealing with Duplicates

How you deal with duplicates depends on what you are trying to achieve. If you are attempting to develop data on trends then duplicates will result in overcounting unless you take steps to count only distinct records. Duplicates of the same data will also distort text mining of the frequencies of terms. So, from that perspective duplicates are bad. On the other hand. If we are interested in the use of terms over time within an emerging area of science and technology, then we might well want to look in detail at the use of particular terms. For example, synthetic genomics is an alternative term for synthetic biology favoured by the J. Craig Venter group. We could look at whether this term is more widely used. Do synthetic biologists also use terms such as engineering biology, genome engineering or the fashionable new genome editing technique? In these cases duplicate records using terms are good because shifts in language can be mapped over time. This suggests a need for a strategy that uses different data tables to answer different questions.

As we have already seen, it is very easy in R to create new objects (typically data.frames), take some kind of action, and write the data to a file. In thinking about duplicates we would probably first want to find out what we are dealing with by identifying unique records. There are multiple ways to do this, here are two:

unique(results$id) #displays unique DOIs (base R)

n\_distinct(results$id) #displays the count of distinct DOIs (dplyr)

This tells us there are 1,098 unique DOIs meaning there were 307 duplicates at the time of writing.

Next we have two main options.

1. We can spread the duplicate results across the table
2. We can identify and delete the duplicates.

### Spreading data using spread() from tidyr

Rather than simply deleting our duplicate DOIs, we could create new columns for each search term and its associated DOI. This will be useful because it will tell us which terms are associated with which records over time. This is easy to do with spread() by providing a key and a value in the arguments. In this case, we want to use search\_terms as the key (column names) to spread across the table and the DOIs in the id column as the value for the rows.

spread\_results <- spread(results, search\_terms, id)

This creates a column for each search term with the relevant DOIs as the values. Note that the default is to drop the original column (in this case search\_terms) when creating the new columns. Things will go badly wrong if you try to keep the existing column because R will be simultaneously trying to spread the data, thus reducing the size of the table, and keep the table in the same size. So, we will leave the default to drop the column as is.

We now have a data.frame with 1098 rows and the search terms identified in each column. If we briefly inspect spread\_results on the terms at the end we can detect a potentially interesting pattern where some documents are only using terms such as synthetic genome or synthetic genomics while others are using only synthetic biology or a mix of terms.

We have now reduced our data to unique records while preserving our search terms as reference points. The limitation of this approach is that by spreading the DOIs across 4 columns we no longer have a tidy single column of DOIs.

### Deleting Duplicates

As an alternative, or complement, to spread we can use a logical TRUE/FALSE test to filter our dataset. There are a number of functions that perform logical tests in R (see also which(), %in%, within()). In this case the most appropriate choice is probably duplicated(). duplicated() will mark duplicate records as TRUE and non-duplicated records as FALSE. We will add a column to our data using the trusty $ when creating the new column.

results$duplicate <- duplicated(results$id)

If we use View(results) a new column will have been added to results. Records that are not duplicates are marked FALSE while records that are duplicates are marked TRUE. We now want to filter that table down to the results that are not duplicated (are FALSE) from our logical test. We will use filter() from dplyr (see above). While select() works exclusively with columns filter() works with rows and allows us to easily filter the data on the values contained in a row.

unique\_results <- filter(results, duplicate == FALSE) %>%  
 select(- search\_terms) #drop search\_terms column

Here we have asked filter() to show us only those values in the duplicate column that exactly match with FALSE. We now have a data from with 1097 unique results with the DOIs in one column.

The creation of logical TRUE/FALSE vectors is very useful in creating conditions to filter data. In this case however, in the process note that we will lose information from the search\_terms column which will become incomplete. To avoid potential confusion later on we drop the search\_terms column using select(- search\_terms) in the code above. If we wanted to keep the terms we would use the spread method above.

We now have three data.frames, results, spread\_results, and unique\_results. results is our core or reference set. If we planned to do a significant amount of work with this data we would save a copy of results to .csv and label it as raw with notes in our codebook on its origins and the actions taken to generate it. It can be a good idea to .zip a raw file so that it is more difficult to access by accident.

Going forward we would use the spread\_results and unique\_results for further work.

As we did earlier, use either write.csv(x, "x.csv", row.names = FALSE) or the simpler and faster write\_csv(). R can write multiple files in a blink. This will write all three files to the rplos project folder (use getwd() and setwd() if you want to do something different).

write\_csv(results, "results.csv")  
write\_csv(spread\_results, "spread\_results.csv")  
write\_csv(unique\_results, "unique\_results.csv")

Ok, so we have now a dataset containing the records for a range of terms and we have come a long way. Quite a lot of this has been about what to do with PLOS data once we have accessed it in terms of turning it into tables that we can work with. In the next section we will look at how to restrict searches by section.

# Restricting searches by section

The default for searching with rplos is to search everything. This can produce many passing results and be overwhelming. There are quite a number of options for restricting searches in rplos.

## By author

In creating the results dataset above we included the author field. However, there are some complexities to searching with author names and working with author data that it is important to understand. We will start by searching on author names and then look at how to process the data.

To restrict a search by author name we can use either the full name or the surname:

plosauthor(q = "Paul Oldham", fl = c("author", "id"), fq = "doc\_type:full",   
 limit = 20)

In this example we have specified doc\_type:full to return only the results for full articles. If you do not use this then the search will return a large number of repeated results based on article sections. So, in this case, Paul Oldham - the author of this article on rplos - has published two articles in PLOS ONE. If doc\_type:full isn't specified more than 20 results are returned that display different sections of the two articles. This will create a duplication issue later on, so a sensible default approach is to use doc\_type:full.

As a general observation, considerable caution should be exercised when working with author names because of problems with the lumping of names and splitting of names as described in this PLOS ONE article[[19]](#footnote-19). If a large number of results are encountered on a single author name consider using match criteria from other available data fields to ensure that separate persons are not being lumped together by name. Above all, do not assume that simply because a name is the same, or very similar to the target name, that the name designates the same person.

The next issue we need to address is what to do with the author data when we have retrieved it. The reason for this is that the author field in the results is generally a concatenated field containing the names of the authors of a particular article. We will start with the oldham results set.

In this case we will make the call to plosauthor() and then use ldply() from plyr to return a data frame containing meta and data. Then we will use fill from tidyr to take the numFound and fill down that column. We will remove the start column using select() and finally filter() to limit the table to data.

oldham <- plosauthor(q = "Paul Oldham", fl = c("author", "id"), fq = "doc\_type:full", limit = 20) %>%  
 ldply("", 1:2) %>%  
 fill(numFound, start) %>%  
 select(- start) %>%  
 filter(.id == "data")

We now have a two records with the author and id (DOI) data. The next thing we want to do is to separate the author names out. We can do this using separate(). Note that separate() will need to know the number of names involved before hand. In the oldham data case there are three authors of each article. We will deal with how to calculate the number of author names shortly.

oldham <- separate(oldham, author, 1:3, sep = ";", remove = FALSE)

We now have some other choices. We could simply keep only the first author name. To do that, in this particular case, we could use select() and the numeric position of the columns that we want to remove.

first\_author <- select(oldham, -7, -8)

As an alternative, we could place each author name on its own row so that we can focus in on a specific author later. For that we can use gather() from tidyr and the column position numbers (not their names in this case) of the columns we want to gather.

authors <- gather(oldham, number, authors, 5:7)

As above gather() requires a key and value field. In this case we have used the number as our key and authors as our value. We have then specified that we want to gather columns 6 to 8 into the new column authors.

That was easy because we are dealing with a small number of results with a uniform number of authors. However, our results data is more complicated than this because we have multiple author names for each article and the number of authors for the articles could vary considerably.

We will need to organise the data and to run some simple calculations to make this work. This will take six steps. The full working code is below.

1. We calculate the number of columns in our dataset. We do this because the number may vary depending on what fields we retrieve from rplos. We will use ncols() to make the calculation.
2. We use a short function from stringr to calculate the number of authors based on the author name separator ";" (+1 to capture the final names in the sequence). This gives us the maximum number of authors across the dataset that we need to split the data into (in this case 83 as the value of n). Copy and paste the function below into the console to access it.

author\_count <- function(data, col = "", sep = "[[:alnum:]]+") {  
 library(stringr)  
 authcount <- str\_count(data[[col]], pattern = sep)  
 n <- as.integer(max(authcount) + 1)  
 print(n)  
}

1. We use select() from dplyr to move our target column to the first column. This simply makes it easier to specify column positions in separate() and gather() later on.
2. We use the value of n to separate the author names into multiple columns
3. We then gather them back in using the value of n.
4. Splitting on a separator such as ; normally generates invisible leading and trailing white space. This will prevent author names from ranking correctly (e.g. in Excel or Tableau). The str\_trim() function from stringr provides an easy way of removing the white space (specify side as right, left or both).

Copy and paste the code below and then hit Enter.

# ---calculations---  
colno <- ncol(unique\_results) #calculate number of columns  
n <- author\_count(unique\_results, "author", ";") # See function above. Calculate n as an integer to meet requirement for separate()  
# ---select, separate and gather---  
full\_authors <- select(unique\_results, author, 1:colno) #bring author to the front  
full\_authors <- separate(full\_authors, author, 1:n, sep = ";", remove = TRUE,   
 convert = FALSE, extra = "merge", fill = "right") #separate  
full\_authors <- gather(full\_authors, value, authors, 1:n, na.rm = TRUE) #gather  
# ---trim authors----  
full\_authors$authors <- str\_trim(full\_authors$authors, side = "both") #trim leading and trailing whitespace

We can simplify this with pipes to bring together the actions on the new full\_authors object.

# ---calculations---  
colno <- ncol(unique\_results)  
n <- author\_count(unique\_results, "author", ";")  
# ---select, separate, gather---  
full\_authors <- select(unique\_results, author, 1:colno) %>% separate(author,   
 1:n, sep = ";", remove = TRUE, convert = FALSE, extra = "merge", fill = "right") %>%   
 gather(value, authors, 1:n, na.rm = TRUE)  
# ---trim authors----  
full\_authors$authors <- str\_trim(full\_authors$authors, side = "both")

In running this code we will remove the original author column (column 1) by specifying remove = TRUE in separate(). gather() will place the new authors column at the end. So, make sure you scroll to the final column when viewing the results. We could also drop unwanted columns.

We now have a complete list of individual author names that could be used to look up individual authors, to clean up author names for statistical use and for author network mapping. As a brief example, if we wanted to look up contributions by Jean Peccoud who leads the PLOS SynBio blog[[20]](#footnote-20) we might use the following based on this useful Stack Overflow answer[[21]](#footnote-21). See ?grepl for more info.

Peccoud <- filter(full\_authors, grepl("Peccoud", authors))

We will not go into depth on these topics, but generating this type of author list is an important step in enabling wider analytics and visualisation. While the code used to get to this list of authors may appear quite involved, once the basics are understood it can be used over and over again.

Let's write that data to a .csv file to explore later.

write\_csv(full\_authors, "full\_authors.csv")

## Title search using plostitle()

For a title search we can use plostitle(). As above you may want to count the number of records first using:

t <- plostitle(q = "synthetic biology", limit = 0)$meta$numFound

Then we run the search to return the number of results we would like. Here we have set it to the value of t above (11). We have limited the results to the data field by subsetting with $data.

title <- plostitle(q = "synthetic biology", fl = "title", limit = t)$data

## Abstract search using plosabstract()

For confining the searches to abstracts we can use plosabstract(). We will start with a quick count of records.

a <- plosabstract(q = "synthetic biology", limit = 0)$meta$numFound

To retrieve the results we could use the value of a. As an alternative we could set it arbitrarily high and the correct results will be returned. Of course if we don't know what the total number of results are then we will be unsure whether we have captured the universe. But, an arbitrary number can be useful for exploration.

abstract <- plosabstract(q = "synthetic biology", fl = "id, title, abstract",   
 limit = 200)  
abstract$data

As before, we can easily create a new object containing the data.frame. In this case we will also include the meta data and then use fill() from tidyr() to fill down the numFound field and the start with 0. Note that meta will appear at the top of the list and will create a largely blank row. To avoid this, while keeping number of records for reference, we will use filter from tidyr(). This short code will do that.

abstract\_df <- ldply(abstract, "", 1:2) %>% fill(numFound, start) %>% filter(.id ==   
 "data")

## Subject Area using plossubject()

To search by subject area use plossubject. The default return is 10 results of the total results. So, try starting with a search such as this to get an idea of how many results there are. In this case the query has been limited to PLOS ONE and full text articles.

sa <- plossubject(q = "\"synthetic+biology\"", fq = list("cross\_published\_journal\_key:PLoSONE",   
 "doc\_type:full"))$meta$numFound

At the time of writing this returns 739 results. We will simply pull back 10 results. To pull back all of the results replace 10 with sa above or type the number into limit =.

plossubject(q = "\"synthetic+biology\"", fl = "id", fq = list("cross\_published\_journal\_key:PLoSONE",   
 "doc\_type:full"), limit = 10)

As noted in the documentation, the results we return from the API and the results on the website are not necessarily the same because the settings used by PLOS on the website are not clear.

In this case we return 740 results while, at the time of writing, PLOS ONE lists 417 articles in the Synthetic Biology subject area[[22]](#footnote-22). This will merit clarification of the criteria for counts used on the PLOS website and the API returns.

# Highlighting terms and text fragments with highplos()

highplos() is a great function for research in PLOS, particularly when combined with opening results in a browser using highbrow().

Highlighting will pull back a chunk of text with the search term highlighted with the emphasis tag enclosing the individual words in a search phrase. It is possible that an entire phrase can be highlighted (see hl.usePhraseHighlighter) but this requires further exploration.

In this example we will simply use the term synthetic biology and then highlight the terms in the abstract hl.fl = and limit this to 10 rows of results. We will also add the function highbrow() (for highlight browse) at the end. This will open the results in our browser. In the examples we use a pipe (%>%) meaning this %then% that. This means that we do not have to enter the name snippet into the highbrow function and simplifies the code.

When reviewing the results in a browser note that we can click on the DOI to see the full article. This is a really useful tool for assessing which articles we might want to take a closer look at.

highplos(q = '"synthetic biology"', hl.fl = 'abstract', fq = "doc\_type:full", rows = 10) %>%  
 highbrow() #launches the browser

Note that in some cases, even though we are restricting to doc-type:full, we retrieve entries with no data. In one case this is because we are highlighting terms in the abstract when the term appears in the full text. In a second case we have picked up a correction where one of the authors is at a synthetic biology centre but neither the abstract or text mention synthetic biology. So, bear in mind that some further exploration may be required to understand why particular results are being returned. These issues are minor and this is a great tool.

There are two additional options (arguments) for highplos() that we can use. The first of these is snippets using hl.snippets = and the second is hl.fragsize =. Both can be used in conjunction with highbrow().

### Snippets using hl.snippets

snippet <- highplos(q = '"synthetic biology"', hl.fl = list("title", "abstract"), hl.snippets = 10, rows = 100) %>%  
 highbrow()

The snippets argument is handy (the default value for a snippet is 1 but goes up to as many as you like). It become very interesting when we add hl.mergeContiguous = 'true'. This will display the entries captured in the order of the articles to provide a sense of its uses by the author(s).

highplos(q='"synthetic biology"', hl.fl = "abstract", hl.snippets = 10, hl.mergeContiguous = 'true', rows = 10) %>%  
 highbrow()

### fragment size using hl.fragsize

Greater control over what we are seeing is provided using the hl.fragsize option. This allows us to specify the number of characters (including spaces) that we want to see in relation to our target terms.

In the first example we will highlight the phrase synthetic biology in the titles and abstracts and set the fragment size (using hl.fragsize ) to a high 500. This will return the first 500 characters including spaces rather than words. We will set the number of rows to a somewhat arbitrary 200. This can easily be pushed a lot higher but expect to wait for a few moments if you move this to 1000 rows.

highplos(q = '"synthetic biology"', hl.fl = list("title", "abstract"), hl.fragsize = 500, rows = 200) %>%  
 highbrow()

We can also do the reverse of a larger search by reducing the fragment size to say up to 100 characters. At the moment it is unclear whether it is possible to control whether characters are selected to the right or the left of our target terms. Note that results will display up to 100 characters where they are available (short results will be for sentences such as titles that are less than 100 characters)

highplos(q = '"synthetic biology"', hl.fl = list("title", "abstract"), hl.fragsize = 100, rows = 200) %>%  
 highbrow()

What is great about this is that we can easily control the amount of text that we are seeing and then select articles of interest to read straight from the browser. We can also start to think about ways to use this information for text mining to identify terms used in conjunction with synthetic biology or types of synthetic biology.

## Get the full text of one or more articles

We will finish this article by briefly demonstrating how to retrieve and save the full text of one or more articles. rplos uses a combination of the XML and the tm (for text mining) package.

Retrieving full text should initially be used rather sparingly because you could pull back a lot of data in XML format that you may then struggle to process. So, it is probably best to start small.

Using the unique\_results data that we created above we have a list of DOIs in the id field. We can create a vector of these using the following:

doi <- unique\_results$id

That has created a vector of 1097 dois. To limit those results, let's create a shorter version where we select five rows.

short\_doi <- doi1:5]\*

Now we can use plos\_fulltext() to retrieve the full text.

ft <- plos\_fulltext(short\_doi)

When we pull back the two articles an object is created of class plosft. To see the full text of one of the individual articles we use the trusty $ and then select a doi.

ft$`10.1371/journal.pone.0140969`

This displays a lot of the XML tags inside the text. We would now like to extract the text without the XML tags. The rplos documentation for plos\_fulltext() helps us to do this using the following code. The first part of the code uses the XML package to parse the results removing the xml tags in the process.

library(tm)  
library(XML)  
ft\_parsed <- lapply(ft, function(x) {  
 xpathApply(xmlParse(x), "//body", xmlValue)  
})

If we type ft\_parsed we will now see the text (the body without title and abstract) fly by without all of the tags.

ft\_parsed

The object returned by this is a list (use class(ft\_parsed)). Next, we can transform this into a corpus (a text or collection of texts) that we can save to disk using the following code from the rplos plos\_fulltext() example.

tmcorpus <- Corpus(VectorSource(ft\_parsed))

If we type tmcorpus$ into the console then we will see 1 to 5 pop up, but this will return NULL if selected. The data is there but we need to use str(tmcorpus) to see the structure of the corpus. If we want to view a text within the corpus we can use writeLines()

writeLines(as.character(tmcorpus2]\*]\*))

We can also view the five texts in our corpus (be prepared for a lot of scrolling) by using lapply to read over the two texts as character.

lapply(tmcorpus1:5]\*, as.character)

For more information see the Ingo Feinerer (2015) Introduction to the tm package[[23]](#footnote-23) (also available in the tm documentation) from which the above is drawn.

## Writing a corpus to disk

To write a corpus we first need to create a folder where the files will be housed (otherwise they will simply be written into your project folder with everything else).

The easiest way to create a new folder is to head over to the Files Tab in RStudio (normally in the bottom right pane) and choose New Folder. We will call it tm.

Now use getwd() and copy the file path into the following function, from the writeCorpus examples, adding /tm at the end. It will look something like this but replace the path with your own, not forgetting the /tm. Then press Enter.

writeCorpus(tmcorpus, path = "/Users/paul/Desktop/open\_source\_master/rplos/tm")

When you look in the tm folder inside rplos (use the Files tab in RStudio) you will now see five texts with the names 1 to 5. For more details, such as naming files and specifying file types, see ?writeCorpus and the tm package documentation.

## Round Up

In this article we have focused on using the rplos package to access scientific articles from the Public Library of Science (PLOS). As we have seen, with short pieces of code it is easy to search and retrieve data from PLOS on a whole range of subjects whether it be pizza or synthetic biology.

One of the most powerful features of R is that it is quite easy to access free online data using APIs. rplos is a very good starting point for learning how to retrieve data using an API because it is well written and the data that comes back is remarkably clean.

Perhaps the biggest challenge facing new users of R is what to do with data once you have retrieved it. This can result in many hours of frustration staring at a list or object with the data you need without the tools to access it and transform it into the format you need. In this article we have focused on using the plyr, dplyr, tidyr and stringr suite of packages to turn rplos data into something you can use. These packages are rightly very popular for everyday work in R and becoming more familiar with them will reap rewards in learning R for practical work. At the close of the article we used the tm (text mining) package to save the full text of articles. This is only a very small part of this package and rplos provides some useful examples to begin text mining using tm (see the plos\_fulltext() examples). R now has a rich range of text mining packages and we will address this in a future article.

In the next article we will focus on a second rOpenSci package fulltext for obtaining the scientific literature.

In the meantime, if you would like to learn more about R try the resources below. If you would like to learn R inside R then try the very useful Swirl package (details below).

## Resources

1. rOpenSci[[24]](#footnote-24)
2. Winston Chang's R Cookbook[[25]](#footnote-25)
3. RStudio Online Learning[[26]](#footnote-26)
4. r-bloggers.com[[27]](#footnote-27)
5. Datacamp[[28]](#footnote-28)
6. Swirl (developed by the free Coursea R Programming course[[29]](#footnote-29) team at John Hopkins University

install.packages("swirl")  
library(swirl)

1. <https://github.com/ropensci/rplos> [↑](#footnote-ref-1)
2. <https://ropensci.org> [↑](#footnote-ref-2)
3. <https://www.plos.org> [↑](#footnote-ref-3)
4. <http://api.plos.org/solr/faq/> [↑](#footnote-ref-4)
5. <https://ropensci.org/tutorials/fulltext_tutorial.html> [↑](#footnote-ref-5)
6. <https://github.com/ropensci/fulltext> [↑](#footnote-ref-6)
7. <http://cran.rstudio.com/> [↑](#footnote-ref-7)
8. <https://www.rstudio.com/products/rstudio/download/> [↑](#footnote-ref-8)
9. <https://ropensci.org/tutorials/rplos_tutorial.html> [↑](#footnote-ref-9)
10. <http://api.richcitations.org/> [↑](#footnote-ref-10)
11. <http://www.amazon.com/Hands-On-Programming-Write-Functions-Simulations/dp/1449359019> [↑](#footnote-ref-11)
12. <https://leanpub.com/datastyle> [↑](#footnote-ref-12)
13. <https://ropensci.org/blog/2014/06/09/reproducibility/> [↑](#footnote-ref-13)
14. <http://rsta.royalsocietypublishing.org/content/372/2031/20140065> [↑](#footnote-ref-14)
15. <http://journals.plos.org/plosone/article?id=10.1371/journal.pone.0034368> [↑](#footnote-ref-15)
16. <http://seananderson.ca/2014/09/13/dplyr-intro.html> [↑](#footnote-ref-16)
17. <https://www.rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf> [↑](#footnote-ref-17)
18. <http://r.789695.n4.nabble.com/How-to-convert-blanks-to-NA-td895155.html> [↑](#footnote-ref-18)
19. <http://journals.plos.org/plosone/article?id=10.1371/journal.pone.0070299> [↑](#footnote-ref-19)
20. <http://blogs.plos.org/synbio/> [↑](#footnote-ref-20)
21. <http://stackoverflow.com/questions/22850026/filtering-row-which-contains-a-certain-string-using-dplyr> [↑](#footnote-ref-21)
22. <http://www.plosone.org/browse/Synthetic+biology?startPage=0&filterAuthors>=&filterSubjectsDisjunction=&filterArticleTypes=&pageSize=13&filterKeyword=&filterJournals=PLoSONE&query=&ELocationId=&id=&resultView=&sortValue=&unformattedQuery=*%3A*&sortKey=Most+views%2C+all+time&filterSubjects=Synthetic%20biology&volume=& [↑](#footnote-ref-22)
23. <https://cran.r-project.org/web/packages/tm/vignettes/tm.pdf> [↑](#footnote-ref-23)
24. <https://ropensci.org/> [↑](#footnote-ref-24)
25. <http://www.cookbook-r.com/> [↑](#footnote-ref-25)
26. <https://www.rstudio.com/resources/training/online-learning/> [↑](#footnote-ref-26)
27. <http://www.r-bloggers.com/> [↑](#footnote-ref-27)
28. <https://www.datacamp.com/> [↑](#footnote-ref-28)
29. <https://www.coursera.org/course/rprog> [↑](#footnote-ref-29)