Working with strings

Tad Dallas

Contents

VERENA

The Viral Emergence Research Initiative (VERENA) is a global consortium. Our goal is to curate the largest ecosystem of open data in viral ecology, and build tools to help predict which viruses could infect humans, which animals host them, and where they could someday emerge.

Learn more about the consortium here

VIRION

With over 3 million records, the Global Virome in One Network (VIRION) database is a living encyclopedia of vertebrate viruses - including the ones that pose the greatest threats to human health. Data like these pave the way for a new era of predictive science, and form the backbone for a broader data ecosystem we're building for animal disease surveillance.

Exploring the Virion data

Download the Virion data from the link below:

> https://github.com/viralemergence/virion/tree/main/Virion

There are many different bits of information, including detection information (how was the host-virus association quantified?), taxonomic information on hosts and viruses, and the host-virus association data. Let's load the entire dataset and get a better idea of the scope and nature of the data

> https://github.com/viralemergence/virion/blob/main/Virion/Virion.csv.gz

is where the data lives. It is compressed to avoid file size limitation on GitHub, but can be read into `R` using base functionality (or using the `vroom` package).

```
# if you downloaded the file
virion <- read.delim(gzfile('Virion.csv.gz'))
```

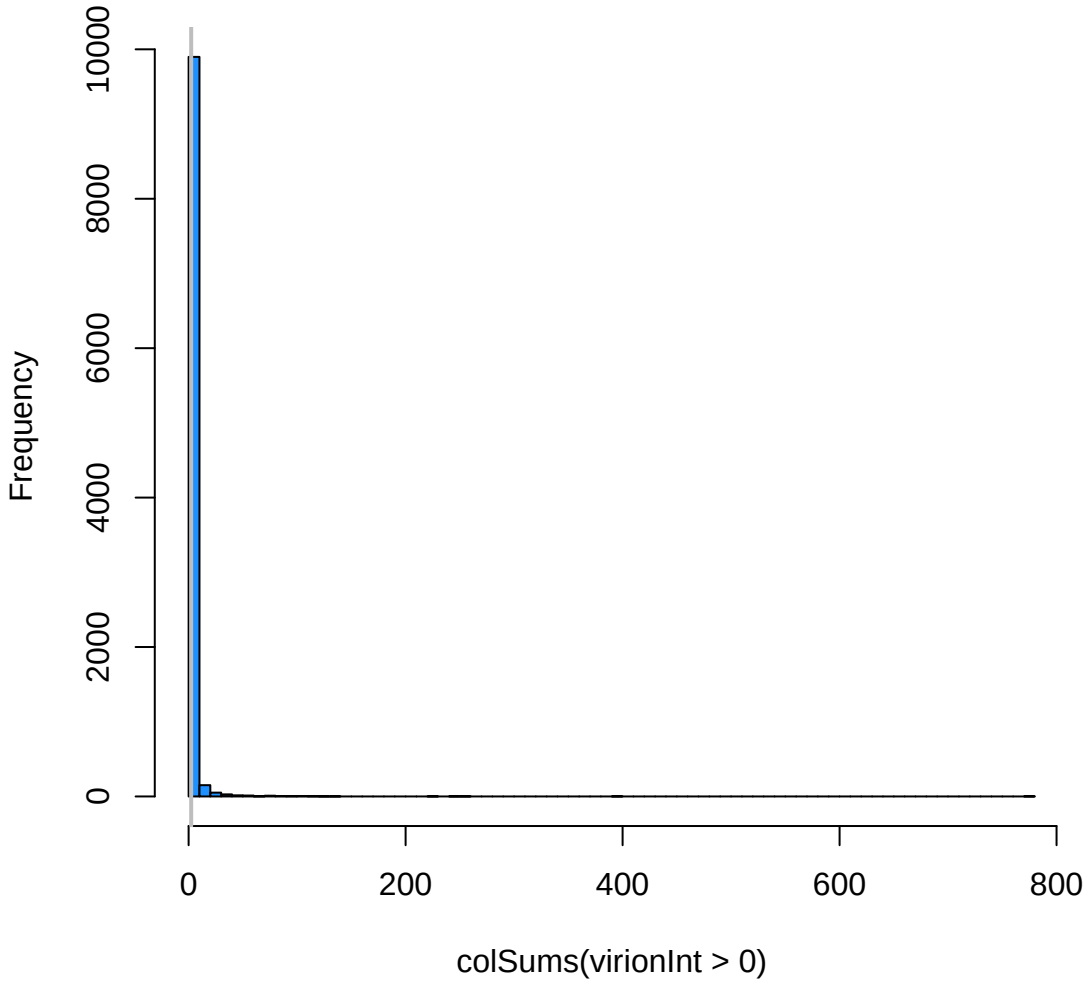
```
# reading directly from web resource
con <- gzcon(url("https://github.com/viralemergence/virion/raw/main/Virion/Virion.csv.gz"))
txt <- readLines(con)
virion <- read.delim(textConnection(txt))
```

```r
# make an interaction matrix
virionInt <- table(virion$Host, virion$Virus)


dim(virionInt)
```

```
## [1]  3718 10192
```

```r
# how host-specific are viruses?
hist(colSums(virionInt>0),
  col='dodgerblue', breaks=100,
  main='')
abline(v=mean(colSums(virionInt>0)), lwd=2, col='grey')
```




```r
# on average, viruses infect around 2 species
mean(colSums(virionInt>0))
```

```
## [1] 2.362834
```

```r
# but this one infects over x species
which.max(colSums(virionInt[,-1] > 0))
```

```
## influenza a virus
##              4931
```

This is the sort of exploratory data analysis that researchers do to try to understand the data. It can also be used to generate research questions. Why is influenza a virus so common in the data relative to the majority of other pathogens? How do patterns of host-virus associations change across different host groups (i.e., are there some pathogens that only infect certain groups of host species?).

So now we're working with real data, and specifically a lot of the data are structured in a way that we haven't really seen. Here, each row of the data is an interaction between a given host species and a given virus. The names of the virus and host species are given as strings. We've introduced strings, but not really dug into how to handle these types of data. So the goal of this lecture is two-fold; learn how to work with strings and start to gain experience working with real data to gain biological insight.

String manipulation in base R

So let's take a break from working with Virion directly, and instead talk about working with strings more generally. Strings are those variables that are identified as object type `character`. Commonly these will have quotes or single ticks around them.

```
vonn <- c('Everything was beautiful and nothing hurt')
is.character(vonn)
```

```
## [1] TRUE
```

```
# this is a character vector with a single entry

# mess with case
tolower(vonn)
```

```
## [1] "everything was beautiful and nothing hurt"
```

```
toupper(vonn)
```

```
## [1] "EVERYTHING WAS BEAUTIFUL AND NOTHING HURT"
```

Each string is made up of a set of characters, which can be queried using the `nchar` function.

```
nchar(vonn)
```

```
## [1] 41
```

```
length(vonn)
```

```
## [1] 1
```

Why is there a difference between `length` and `nchar` here?

So we often have long vectors of strings, where we want to know something about each string itself. For instance, what if we wanted to extract the first word in a string? We would need to know how many characters to choose. In this case, we might want to say that the first word is defined as all characters from the start of the string until the first space is detected.

```
# get the first word in the string (if we know the size of the word)
substr(vonn, 1, 10)
```

```
## [1] "Everything"
```

```
# get the first word in the string (if we do not know the size of the word)
spacez <- gregexpr(' ', vonn)
str(spacez)
```

```
## List of 1
##  $ : int [1:5] 11 15 25 29 37
##   ..- attr(*, "match.length")= int [1:5] 1 1 1 1 1
```

```
##   ..- attr(*, "index.type")= chr "chars"
##   ..- attr(*, "useBytes")= logi TRUE
```

```r
substr(vonn, 1, spacez[[1]][1]-1)
```

```
## [1] "Everything"
```

```r
# get the last word in the string (if we know size)
substr(vonn, nchar(vonn)-3, nchar(vonn))
```

```
## [1] "hurt"
```

```r
# get the last word in the string (if we do not know size)
substr(vonn, 1+spacez[[1]][length(spacez[[1]])], nchar(vonn))
```

```
## [1] "hurt"
```

?? Let's do some practice work with gregexpr and substr before moving on??

Now we can move on to how we actually manipulate the strings themselves (i.e., above we can pull out bits of strings, but what if we want to replace certain characters in a string?). An example of this could be if there are special characters which may throw errors, spelling mistakes, etc. that we want to resolve before we continue on to analysis.

```r
# replace single characters
chartr('e', 'a', vonn)
```

```
## [1] "Evarything was baautiful and nothing hurt"
```

```r
chartr('et', 'ad', vonn)
```

```
## [1] "Evarydhing was baaudiful and nodhing hurd"
```

```r
# replace entire sets of characters
gsub("Everything", "Nothing", vonn)
```

```
## [1] "Nothing was beautiful and nothing hurt"
```

```r
# the use of the wildcard "."
gsub(".", "Nothing", vonn)
```

```
## [1] "NothingNothingNothingNothingNothingNothingNothingNothingNothingNothingNothingNothingNothingNothi
```

But this all assumes that we are working with a vector of length 1 basically, right? A lot of times, when we work with characters, we will be working with longer vectors. As a toy example, we will take the `vonn` character vector (of length 1) and split it into a character vector where every word is a separate entry to the vector. If this does not make sense, no worries. We'll explore this in code below.

```r
vonn2 <- strsplit(vonn, ' ')

## what is the structure of vonn2?
str(vonn2)
```

```
## List of 1
##  $ : chr [1:6] "Everything" "was" "beautiful" "and" ...
```

```r
vonn2 <- unlist(vonn2)

## but what if we want to paste it back together?
paste(vonn2, collapse=' ')
```

```
## [1] "Everything was beautiful and nothing hurt"
```

```
## paste is also really useful for appending or combining character strings with different properties

paste(vonn2, LETTERS[1:length(vonn2)], sep='-')
```

```
## [1] "Everything-A" "was-B"        "beautiful-C"  "and-D"        "nothing-E"
## [6] "hurt-F"
```

```
## but let's get back to vectors of characters instead of just whole sentences as strings
vonn2
```

```
## [1] "Everything" "was"        "beautiful"  "and"        "nothing"
## [6] "hurt"
```

So we have now have a vector of words that we can mess with. Luckily, a lot of the functionality we have covered about working with single strings translates to working with vectors of strings.

```
nchar(vonn)
```

```
## [1] 41
```

```
nchar(vonn2)
```

```
## [1] 10  3  9  3  7  4
```

```
tolower(vonn)
```

```
## [1] "everything was beautiful and nothing hurt"
```

```
tolower(vonn2)
```

```
## [1] "everything" "was"        "beautiful"  "and"        "nothing"
## [6] "hurt"
```

```
startsWith(vonn2, 'b')
```

```
## [1] FALSE FALSE  TRUE FALSE FALSE FALSE
```

```
endsWith(vonn2, 'g')
```

```
## [1]  TRUE FALSE FALSE FALSE  TRUE FALSE
```

Introducing `grep`. We sort of saw `grep` come up before in `grepexpr`. The idea behind much of these is that you have a `pattern` and you want to know either 1) if the pattern is present (`grepl`) or 2) where the pattern is present (`grep`). We'll go through some examples to clarify the differences and when you might want one over the other.

```
vonn2
```

```
## [1] "Everything" "was"        "beautiful"  "and"        "nothing"
## [6] "hurt"
```

```
## So let's say we want to identify where the word `hurt` occurs in our character vector. We can do thi
grepl("hurt", vonn2)
```

```
## [1] FALSE FALSE FALSE FALSE FALSE  TRUE
```

```
grep("hurt", vonn2)
```

```
## [1] 6
```

How are these two related? Do you see any advantages or disadvantages to their use?

But `grep` is quite extensible. Here, we'll go over how to format queries to `grep` depending on what you want as an output. Much of this will simply boil down to how the query is formulated. There is some specific

syntax to `grep` that will be helpful.

For instance, the use of **anchors** allows you to match only those strings with your query in a particular place in the string. For instance, if we want to identify words in a string starting with a letter, we can use the `^query` notation (the little `^` symbol indicates that the line must start with that letter or string). Further, we can use `$` at the end of a query to mean only find those strings that end in some query.

```
vonn2
```

```
## [1] "Everything" "was"        "beautiful"  "and"        "nothing"
## [6] "hurt"
```

```
# how many words contain an `a`?
grep('a', vonn2)
```

```
## [1] 2 3 4
```

```
# how many words start with `a`?
grep('^a', vonn2)
```

```
## [1] 4
```

```
# how many words end with `l`?
grep('l$', vonn2)
```

```
## [1] 3
```

We cannot directly combine these, as the combination of `^` and `&` will search for an **exact match** of a given string. This is useful, but perhaps not what we're looking for. In the command line `grep` (`grep` is not just a thing in `R`), we can issue multiple commands

```
# how many words start and end with `h`?
grep('^h$', vonn2)
```

```
## integer(0)
```

```
grep('^h$', 'huh')
```

```
## integer(0)
```

```
grepl('^h', 'huh') & grepl('h$', 'huh')
```

```
## [1] TRUE
```

String manipulation in the tidyverse

I believe this is the first mention of the `tidyverse`, so let's talk about what that is. One of the strengths of R is that users can develop bodies of code (called packages or libraries) that offer additional functionality on top of the base language. We may have already introduced a package at this point, but hopefully not. The `tidyverse` is a collection of packages developed by the folks behind the RStudio IDE (company is now known as posit). A lot of introductory courses start with `tidyverse` packages as a way to get students up and running with R quickly. It certainly does help, and you are almost certainly going to run into many stackoverflow posts that offer `tidy` solutions to problems. Whether you want to use the `tidyverse` is 100% personal preference. I am typically of the mindset to limit `dependencies` in my code, where `dependencies` are those things that my code needs in order to run. If I were to need `stringr` (as the code below uses), it means that for someone else to use my code, they also need this library installed. But it's not just this library, but any library which `stringr` depends on.

Alright. So we can now start to use R packages to do some of the same manipulations (and more) that we previously did in base R.

```
#install.packages('stringr', repos='https://cloud.r-project.org')
library(stringr)

paste(vonn2, collapse=' ') == stringr::str_c(vonn2, collapse = ' ')
```

```
## [1] TRUE
```

```
substr(vonn2, 1, 3) == stringr::str_sub(vonn2, 1, 3)
```

```
## [1] TRUE TRUE TRUE TRUE TRUE TRUE
```

```
str_to_upper(vonn2)==toupper(vonn2)
```

```
## [1] TRUE TRUE TRUE TRUE TRUE TRUE
```

```
str_to_upper(vonn2, locale='tr')
```

```
## [1] "EVERYTHİNG" "WAS"        "BEAUTİFUL"  "AND"        "NOTHİNG"
## [6] "HURT"
```

```
str_detect(vonn2, 'hurt')
```

```
## [1] FALSE FALSE FALSE FALSE FALSE  TRUE
```

```
str_detect(vonn2, '[0-9]')
```

```
## [1] FALSE FALSE FALSE FALSE FALSE FALSE
```

```
str_count(vonn2, 'u')
```

```
## [1] 0 0 2 0 0 1
```

```
str_locate(vonn2, 'ur')
```

```
##      start end
## [1,]    NA  NA
## [2,]    NA  NA
## [3,]    NA  NA
## [4,]    NA  NA
## [5,]    NA  NA
## [6,]     2   3
```

```
str_replace(vonn2, 'Everything', 'nothing')
```

```
## [1] "nothing"   "was"       "beautiful" "and"       "nothing"   "hurt"
```

VIRION

So let's go back to VIRION! We now have a set of tools that we can use to start to explore patterns of host-virus associations.

How many records exist for host species of the genus 'Abramis'?

How many unique pathogen species in the data have the word 'virus' in them?

How many host-virus records are from GenBank?

How many host-virus records have a reference text that is a website?

Replace all instances of raccoon (species name is 'procyon lotor') with 'trash panda'.

How many times does the character string 'ad' appear in the `VirusFamily` column?

Tie back into plotting. Maybe plot out parasite species richness across host species or something like that if time is available?

How many host species have more than 3 'e's in them? (easy with tidyverse, more challenging in base)

sessionInfo

```
sessionInfo()
```

```
## R version 4.3.1 (2023-06-16)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Ubuntu 22.04.3 LTS
##
## Matrix products: default
## BLAS:   /usr/lib/x86_64-linux-gnu/atlas/libblas.so.3.10.3
## LAPACK: /usr/lib/x86_64-linux-gnu/atlas/liblapack.so.3.10.3;  LAPACK version 3.10.0
##
## locale:
##  [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
##  [3] LC_TIME=en_US.UTF-8        LC_COLLATE=en_US.UTF-8
##  [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
##  [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
##  [9] LC_ADDRESS=C               LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C
##
## time zone: America/New_York
## tzcode source: system (glibc)
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
## [1] stringr_1.5.0 plotly_4.10.2 ggplot2_3.4.2 dplyr_1.1.2   plyr_1.8.8
##
## loaded via a namespace (and not attached):
##  [1] viridis_0.6.3      sass_0.4.7         utf8_1.2.3         generics_0.1.3
##  [5] tidyr_1.3.0        stringi_1.7.12     lattice_0.21-8     digest_0.6.33
##  [9] magrittr_2.0.3     evaluate_0.21      grid_4.3.1         RColorBrewer_1.1-3
## [13] fastmap_1.1.1      jsonlite_1.8.7     tinytex_0.45       gridExtra_2.3
## [17] httr_1.4.6         purrr_1.0.1        fansi_1.0.4        crosstalk_1.2.0
## [21] viridisLite_0.4.2  scales_1.2.1       lazyeval_0.2.2     jquerylib_0.1.4
## [25] cli_3.6.1          rlang_1.1.1        ellipsis_0.3.2     munsell_0.5.0
## [29] withr_2.5.0        cachem_1.0.8       yaml_2.3.7         tools_4.3.1
## [33] colorspace_2.1-0   vctrs_0.6.3        R6_2.5.1           lifecycle_1.0.3
## [37] htmlwidgets_1.6.2  pkgconfig_2.0.3    hexbin_1.28.3      pillar_1.9.0
## [41] bslib_0.5.0        gtable_0.3.3       glue_1.6.2         data.table_1.14.8
## [45] Rcpp_1.0.11        xfun_0.39          tibble_3.2.1       tidyselect_1.2.0
## [49] highr_0.10         knitr_1.43         farver_2.1.1       htmltools_0.5.5
## [53] labeling_0.4.2     rmarkdown_2.23     compiler_4.3.1
```