13: Strings and things

Graphical Analysis of Biological Data

This web page is available as a PDF file

Strings

Carefully read R4ds Chapter 14: Strings sections 1-4. I suggest you run the examples but I am not requiring that you do so. The assignment will require you to use some of the functions.

Strings are sequences of characters in `char` format. Any character, including numbers, can be treated as strings. Strings in R are enclosed in double or single quotes. Use double quotes to ensure best practice.

Focus, I mean really **focus** on Sections 3 and 4, regular expressions (regex). Regex is tricky to learn but once you learn it, you are able to harness a lot of power.

For the DNA section, you will have to use the `read_lines()` function to import the file. `read_lines()` reads in the entire file, one line at a time. Each line is treated as a separate string.

The joy of regular expressions

Fair warning: I suspect this section will give you the most trouble. You will use several `stringr` functions in combination with regex patterns. Here are functions you will use at various places in the assignment.

- `str_c()`
- `str_replace()`
- `str_replace_all()`
- `str_to_upper()`
- `str_count()`
- `str_length()`
- `str_extract_all()`
- `str_which()`

Here are some things to remember:

- Read the sections listed above and practice the examples and questions. They are not required but regex patterns take practice.
- R uses backslashes to escape certain characters such as `\t` for tab and `\n` for new line characters. To use these as part of regex pattern strings, you have to escape the backslash with a second backslash, like `"\\t"` or `"\\n"`.
- Use parentheses to capture groups that can be used later. For example, `"(CG)\\1\\1"` would find a repeating pattern of "CGCGCG". This technique is very useful when using `str_replace` when you want to keep part of the original string. Consider this example. You want to remove the letters and the dash, keeping only the numbers.

```
tb <- data.frame(x = c("aa11-bb22", "xx88-yy99"))
tb
```

```
##           x
## 1 aa11-bb22
## 2 xx88-yy99
```

```
tb <- tb %>%
  mutate(x = str_replace(x, "[a-z]+(\\d+)-[a-z]+(\\d+)", "\\1\\2"))
tb
```

```
##      x
## 1 1122
## 2 8899
```

- Use combinations of matching patterns to build up the pattern you want. For example, assume you have species names in the form of "Genus_species". You want to replace them as "G. species". Notice the use `\\.` in the replacement pattern to use an actual period.

```
bacteria <- tibble(species = c("Escherischia_coli", "Bacillus_subtilis"))

bacteria <- bacteria %>%
  mutate(species = str_replace(species, "([A-Z]{1}).+_(.+)$", "\\1\\. \\2"))

bacteria
```

```
## # A tibble: 2 x 1
##   species
##   <chr>
## 1 E. coli
## 2 B. subtilis
```

- Use | in regex patterns as "or." For example, `"this|that"` will match "this" or "that."

```
strings = c("String with this.", "String with that.", "String with the other.")

str_extract(strings, ".*(this|that).*")
```

```
## [1] "String with this." "String with that." NA
```

- You can use `\s` or `[:blank:]` to represent whitespace. You can use `\d`, `[:digits:]`, or `[0-9]+` to represent a string of digits. They must be properly formatted as strings, as described in the text.

- Use `str_count()` to count the number of times a pattern occurs. For example, how many times does a seqence of 2-4 As occur in this DNA sequence? Do not count overlapping sequences. Notice the use of curly braces `{ }` and two numbers separated by a comma to find strings of minimum and maximum lengths. In this case, the search is for 2-4 repetitions of `A`. The general pattern is `{min,max}`. Section 14.3.4 Repetition of your text uses `{n,m}` but I use `{min,max}` to emphasize the minimum and maximum amount of repeitition. You'll see more on `{min,max}` below.

```
dna <- "ATCACTAAATATGATTTGTGTAAAACCAAAATAAGATCTACAAACGAATAGAAGCTAGAGCGAAAAAATGG"

str_count(dna, "[CGT]A{2,4}[CGT]")
```

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

- Run `example(str_c)` or `?str_c` for an example of how to **c**ombine separate strings into a single string. I recommend trying the first one. `example(function_name)` runs the examples shown when you enter `?function_name`.

Going deeper with {min,max}

Consider the string `ABCDEFMSTGHIJKLMNOPQRBCDESTMSTUVWXYZ`. Assume we want to find a string that begins with `BCDE` and ends with `MST`. There are two short strings and one long string that match the pattern. I've used spaces to highlight the matches.

Short strings

```
A BCDEFMST GHIJKLMNOPQR BCDESTMST UVWXYZ
```

Long string

```
A BCDEFMSTGHIJKLMNOPQRBCDESTMST UVWXYZ
```

The letters that separate `BCDE` from `MST` can be any letter. In regular expressions, the `.` represents any character and `+` represents 1 or more matches so we can begin to think of a search string with the pattern `BCDE.+MST`. This is `MST` is separated from `BCDE` by 1 or more characters.

Say we know from other data that `MST` in our sequence of interest is at least 10 characters away from `BCDE` but no more than 25 characters. That means we would have to find the long string but neither of the two short strings.

Therefore, we use `{min,max}` along with `.` to modify our search sequence. We want `BCDE.{10,25}MST`. That gives as a RegEx pattern of `BCDE` followed by a string of any characters (`.`) that is a minimum of 10 and maximum of 25 characters long, followed by `MST`.

Run this code in your R console.

```
# Starting sequence. Copy and paste this.
my_seq <- "ABCDEFMSTGHIJKLMNOPQRBCDESTMSTUVWXYZ"

# Search string.  Notice the `.` after the E. The period (any character)
# is being matched 10-25 times.
search_seq <- "BCDE.{10,25}MST"

# Count number of times the matched string is found
str_count(my_seq, search_seq)
```

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

```
# Extract all instances, to show we're getting the right string.
str_extract_all(my_seq, search_seq)
```

```
## [[1]]
## [1] "BCDEFMSTGHIJKLMNOPQRBCDESTMST"
```

To demonstrate that it works, change the search sequence to find all instances where MST is separated from BCDE by only 1-10 characters, and run again.

```
# Search string
search_seq <- "BCDE.{1,10}MST"

# Count number of times the matched string is found
str_count(my_seq, search_seq)
```

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

```
# Extract all instances
str_extract_all(my_seq, search_seq)
```

```
## [[1]]
## [1] "BCDEFMST"  "BCDESTMST"
```

Both short strings were found, not the long one.

Now, some of you may have thought that you could just do `BCDE.+MST`. In the example above it would work but in the DNA exercise it would not. RegEx is greedy, meaning it would find the longest possible sequence that matches the pattern. Using `BCDE.+MST` for HW13 would return a string that is 3,224 characters long, which is nearly the entire sequence given to you. That would be incorrect.

With careful use of `{min,max}`, you can control how you search for specific patterns in text.

Separating

Read R4ds Chapter 12.4.1: Separate for details of separating a single column into multiple columns

The `separate` function is another `tidyr` function to help you separate data in a single column into multiple columns. You have to provide the names of the columns to receive the data with the `into` argument and a regex pattern with the`sep` argument so the function knows where to separate the data. This example separates the names of the Song Sparrow subspecies into three columns. The name is split into the columns, using space (`" "`) as the separation character.

```
sparrows <- tibble(species = c("Melospiza melodia fisherella",
                               "Melospiza melodia cleonensis",
                               "Melospiza melodia heermanni",
                               "Melospiza melodia mailliardi"))

sparrows
```

```
## # A tibble: 4 x 1
##   species
##   <chr>
## 1 Melospiza melodia fisherella
## 2 Melospiza melodia cleonensis
## 3 Melospiza melodia heermanni
## 4 Melospiza melodia mailliardi
```

```
sparrows <- sparrows %>%
  separate(species,
           into = c("Genus", "species", "subspecies"),
           sep = " ")

sparrows
```

```
## # A tibble: 4 x 3
##   Genus     species subspecies
##   <chr>     <chr>   <chr>
## 1 Melospiza melodia fisherella
## 2 Melospiza melodia cleonensis
## 3 Melospiza melodia heermanni
## 4 Melospiza melodia mailliardi
```

Here is another example that has flower colors in a column. The data were not coded consistently, so colors may be separated by spaces, commas, or semicolons. Notice the use of `|` for “or”. We’ll also assume that the first color listed is most important to us, so we can drop extra text using the `extra` argument.

```r
flowers <- tibble(color = c("red",
                            "red, pink, or white",
                            "white and yellow",
                            "silver; blue; gold"))

flowers
```

```
## # A tibble: 4 x 1
##   color
##   <chr>
## 1 red
## 2 red, pink, or white
## 3 white and yellow
## 4 silver; blue; gold
```

```r
flowers <- flowers %>%
  separate(color,
           into = "main_color",
           sep = ",| |;",
           extra = "drop")

flowers
```

```
## # A tibble: 4 x 1
##   main_color
##   <chr>
## 1 red
## 2 red
## 3 white
## 4 silver
```

The examples above are filled with hints to help you be successful with this assignment.

Join together now

- Read R4ds Chapter 13.4: Mutating joins for details of how to merge two tibbles. You will use an inner join later.

Joins allow you to merge together separate data frames into one data frame. Many types of joins are possible, depending on what you want to accomplish. Look over Jenny Bryan's Join cheatsheet. She uses a fun example to show the results of different types of joins.

We will use only an `inner_join` so at a minimum, study her example of an `inner_join` in detail.

Communicating with graphics

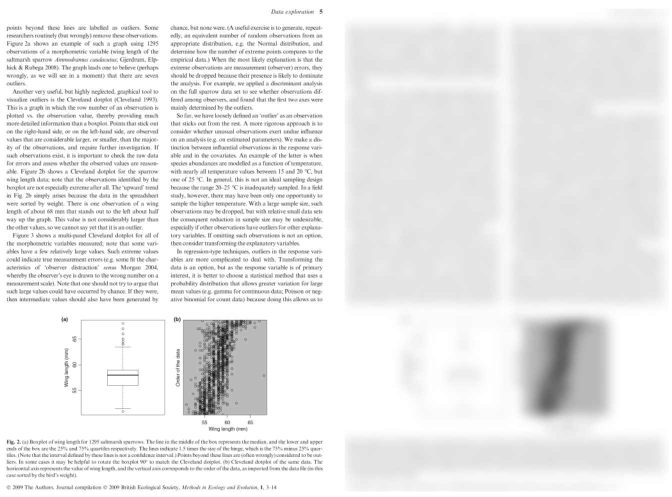
- Read R4ds Chapter 28: Graphics for communication for reference.

Most of our assigments have involved data visualization but we have accepted the default output from `ggplot`, with only a few minor changes. The defaults are good while you are exploring your data. When you are ready to present your data to the broader community (scientific or public), more preparation is necessary.

`ggplot` has a lot of power to make nearly any type of graph you can think of. The depth provided by `ggplot` is greater than we have time to explore but there are a few things I want to touch on.

For starters, you might have wondered why `ggplot` defaults to a gray background for its plots. This relates to what typographers call typographic color or "page color", which represents the average look of the text on the page. Most pages you read are black text on whitish paper. The average typographic color is going to be somewhat gray.

This figure shows one of the pages from Zuur et al. that we used earlier. The page on the right was blurred to bring out the typographic color. Notice how the graph in the right panel tends to match the overall "grayness" of the page but the graph in the left panel leaves a big, white hole in the overall grayness.



Data exploration **5**

points beyond these lines are labelled as outliers. Some researchers routinely (but wrongly) remove these observations. Figure 2a shows an example of such a graph using 1295 observations of a morphometric variable (wing length of the saltmarsh sparrow *Ammodramus caudacutus*; Gjerdrum, Elphick & Rubega 2008). The graph leads one to believe (perhaps wrongly, as we will see in a moment) that there are seven outliers.

Another very useful, but highly neglected, graphical tool to visualize outliers is the Cleveland dotplot (Cleveland 1993). This is a graph in which the row number of an observation is plotted vs. the observation value, thereby providing much more detailed information than a boxplot. Points that stick out on the right-hand side, or on the left-hand side, are observed values that are considerable larger, or smaller, than the majority of the observations, and require further investigation. If such observations exist, it is important to check the raw data for errors and assess whether the observed values are reasonable. Figure 2b shows a Cleveland dotplot for the sparrow wing length data; note that the observations identified by the boxplot are not especially extreme after all. The 'upward' trend in Fig. 2b simply arises because the data in the spreadsheet were sorted by weight. There is one observation of a wing length of about 68 mm that stands out to the left about half way up the graph. This value is not considerably larger than the other values, so we cannot say yet that it is an outlier.

Figure 3 shows a multi-panel Cleveland dotplot for all of the morphometric variables measured; note that some variables have a few relatively large values. Such extreme values could indicate true measurement errors (e.g. some fit the characteristics of 'observer distraction' *sensu* Morgan 2004, whereby the observer's eye is drawn to the wrong number on a measurement scale). Note that one should not try to argue that such large values could have occurred by chance. If they were, then intermediate values should also have been generated by chance, but none were. (A useful exercise is to generate, repeatedly, an equivalent number of random observations from an appropriate distribution, e.g. the Normal distribution, and determine how the number of extreme points compares to the empirical data.) When the most likely explanation is that the extreme observations are measurement (observer) errors, they should be dropped because their presence is likely to dominate the analysis. For example, we applied a discriminant analysis on the full sparrow data set to see whether observations differed among observers, and found that the first two axes were mainly determined by the outliers.

So far, we have loosely defined an 'outlier' as an observation that sticks out from the rest. A more rigorous approach is to consider whether unusual observations exert undue influence on an analysis (e.g. on estimated parameters). We make a distinction between influential observations in the response variable and in the covariates. An example of the latter is when species abundances are modelled as a function of temperature, with nearly all temperature values between 15 and 20 °C, but one of 25 °C. In general, this is not an ideal sampling design because the range 20–25 °C is inadequately sampled. In a field study, however, there may have been only one opportunity to sample the higher temperature. With a large sample size, such observations may be dropped, but with relative small data sets the consequent reduction in sample size may be undesirable, especially if other observations have outliers for other explanatory variables. If omitting such observations is not an option, then consider transforming the explanatory variables.

In regression-type techniques, outliers in the response variables are more complicated to deal with. Transforming the data is an option, but as the response variable is of primary interest, it is better to choose a statistical method that uses a probability distribution that allows greater variation for large mean values (e.g. gamma for continuous data; Poisson or negative binomial for count data) because doing this allows us to

Fig. 2. (a) Boxplot of wing length for 1295 saltmarsh sparrows. The line in the middle of the box represents the median, and the lower and upper ends of the box are the 25% and 75% quartiles respectively. The lines indicate 1.5 times the size of the hinge, which is the 75% minus 25% quartiles. (Note that the interval defined by these lines is not a confidence interval.) Points beyond these lines are (often wrongly) considered to be outliers. In some cases it may be helpful to rotate the boxplot 90° to match the Cleveland dotplot. (b) Cleveland dotplot of the same data. The horizontal axis represents the value of wing length, and the vertical axis corresponds to the order of the data, as imported from the data file (in this case sorted by the bird's weight).

© 2009 The Authors. Journal compilation © 2009 British Ecological Society, *Methods in Ecology and Evolution*, **1**, 3–14

Studies by typographers suggest that pages with more uniform color are easier to read and cause less straing. That is why `ggplot` defaults to a medium gray background. This is also one default that many people prefer to change for the final output, perhaps in part because we're used to graphs with white background.

`ggplot` can be customized with little to lots of effort, depending on what you want to accomplish. `ggplot` also comes with several built-in "themes" that you can add to a plot to quickly change the appearane of the plots. I will often use `theme_minimal()`, `theme_bw()`, or `theme_classic`.

Additional themes are available via the ggthemes package. I show an example with `theme_tufte()` in this assignment. We'll use `theme_map()` next assignment I encourage you to try different themes. Eventually, you will probably dig more deeply under the theme hood to tweak your plots even further.

The other thing I want to mention is color. Color is not always necessary. If you can get it right in black and white, then your graph has the power to convey detailed information efficiently. However, color is often helpful and it is easier than ever to generate and publish color figures. But, different colors do *not* convey information equally.

Here are some references for you to reference and think about visualizing your data in the future.

Expert Color Choices for Presenting Data by Maureen Stone.

Practical Rules for Using Color in Charts by Stephen Few.

Uses and Misuses of Color by Stephen Few.

What about "color blindness?" by Maureen Stone.

Some of the referenced articles discuss colorblindness. Colorblindness is not an accurate term because most non-blind people see color[1] but not all people see color the same way. Choosing the right combination of colors is critical to present your data fairly and so everyone can interpret your results easily. A good rule of thumb is to not depend on just color to distinguish groups in a graph. Shapes and shades of gray, as you have seen, are also useful and often preferred.

[1] Achromatopsia is one form of true color blindness.

When you prepare figures for others to see, you should run draft versions through a colorblind similator. which allows you to see how your figure looks to those with other forms of color vision.