Midterm II

2023-05-26

Your name:

Questions

Q1

Consider the following data frame `df_waste` which represents the waste generation (in tonnes) of three types of waste materials: Plastic, Metal, and Glass in a city.

```
glimpse(df_waste)
## Rows: 9
## Columns: 3
## $ plastic <dbl> 150, 120, 130, 200, 210, 180, 190, 220, 250
## $ metal   <dbl> 100, 110, 120, 130, 140, 150, 130, 120, 110
## $ glass   <dbl> 200, 190, 180, 170, 160, 150, 160, 170, 180
```

```
median_fun <- function(x) quantile(x, probs = c(0.5))
```

a (10 points)

Given `df_waste` and `median_fun`, write a for loop to calculate the median waste generation for each type of material. Your resulting output should be a named vector with the median waste generation of Plastic, Metal, and Glass.

```
#  your r-code
```

b (10 points)

Describe what is returned by the code below, including the type of R object produced, the length or dimension of the object, and the information contained in the object.

```
map_dfr(df_waste, .f = ~median_fun(.x), .id = "material")
```

Q2

You are given a data frame named `df` with two columns: `Name` and `Age`. The `Name` column contains string values representing names of individuals, and the `Age` column contains numeric values representing the ages of the respective individuals.

```
df <- tibble(Name = c("Anna", "Bob", "Ava", "Charlie", "Aba", "Aku"),
             Age = c(30, 25, 35, 40, 28, 26))
df
## # A tibble: 6 x 2
##   Name      Age
##   <chr>   <dbl>
## 1 Anna       30
## 2 Bob        25
## 3 Ava        35
## 4 Charlie    40
## 5 Aba        28
## 6 Aku        26
```

a. (10 points)

Create a function named `is_palindrome` to check if a string is a three-lettered palindrome using `stringr` functions. A string is considered a palindrome if it reads the same backward as forward, case-insensitively.

```r
# your r-code
```

b. (10 points)

Write a function named `process_data.` This function should take a data frame as an argument and create a list named `palindrome_names` that contains the names of individuals with three-lettered palindrome names.

	Predicted Negative	Predicted Positive
Actual Negative	50	20
Actual Positive	10	70

Q3

Consider a binary classification problem where we use a k-Nearest Neighbors (k-NN) algorithm. We have a confusion matrix as above, which shows the classification performance:

a. What is the accuracy of the model according to the provided confusion matrix?

A) 0.75
B) 0.80
C) 0.85
D) 0.90

b. If we increase the value of k in the k-NN algorithm, which of the following is likely to be true?

A) The model will become more complex, leading to a higher chance of overfitting.
B) The model will become less complex, which might reduce overfitting but increase bias.
C) The accuracy of the model will always improve.
D) The model will perfectly classify all the data points.

c. When applying the k-NN algorithm, we notice that the model is suffering from high variance. Which of the following could be a potential solution to this problem?

A) Increase the value of k.
B) Decrease the value of k.
C) Use a smaller dataset for training.
D) None of the above.

d. Considering the bias-variance trade-off in the k-NN algorithm, which of the following statements is true?

A) A small value of k results in a high bias, low variance model.
B) A small value of k results in a low bias, high variance model.
C) A large value of k results in a high bias, high variance model.
D) The bias-variance trade-off does not apply to the k-NN algorithm.

Q4 Miscellaneous: Yes or No ? Please explain briefly.

a. Is it possible for the assignment of observations to clusters not to change between successive iterations in K-Means?

b. In k-NN, do all features contribute equally to the calculation of distance between observations?

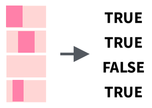
c. Are odds defined as the ratio of the probability of an event occurring to the probability of the event not occurring in logistic regression?

d. Does the use of cross-validation in machine learning algorithms help to mitigate both overfitting and underfitting?

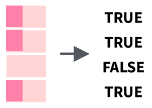
String manipulation with stringr : : **CHEAT SHEET**

The **stringr** package provides a set of internally consistent tools for working with character strings, i.e. sequences of characters surrounded by quotation marks.

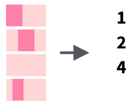
Detect Matches



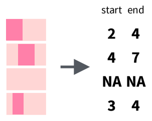
str_detect(string, **pattern**, negate = FALSE) Detect the presence of a pattern match in a string. Also **str_like()**. str_detect(fruit, "a")



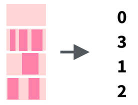
str_starts(string, **pattern**, negate = FALSE) Detect the presence of a pattern match at the beginning of a string. Also **str_ends()**. str_starts(fruit, "a")



str_which(string, **pattern**, negate = FALSE) Find the indexes of strings that contain a pattern match. str_which(fruit, "a")

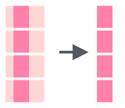


str_locate(string, **pattern**) Locate the positions of pattern matches in a string. Also **str_locate_all()**. str_locate(fruit, "a")

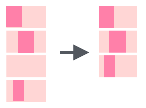


str_count(string, **pattern**) Count the number of matches in a string. str_count(fruit, "a")

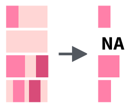
Subset Strings



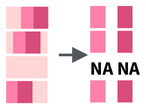
str_sub(string, start = 1L, end = -1L) Extract substrings from a character vector. str_sub(fruit, 1, 3); str_sub(fruit, -2)



str_subset(string, **pattern**, negate = FALSE) Return only the strings that contain a pattern match. str_subset(fruit, "p")

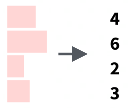


str_extract(string, **pattern**) Return the first pattern match found in each string, as a vector. Also **str_extract_all()** to return every pattern match. str_extract(fruit, "[aeiou]")

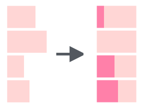


str_match(string, **pattern**) Return the first pattern match found in each string, as a matrix with a column for each ( ) group in pattern. Also **str_match_all()**. str_match(sentences, "(a|the) ([^ +])")

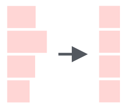
Manage Lengths



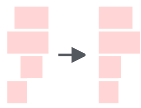
str_length(string) The width of strings (i.e. number of code points, which generally equals the number of characters). str_length(fruit)



str_pad(string, width, side = c("left", "right", "both"), pad = " ") Pad strings to constant width. str_pad(fruit, 17)



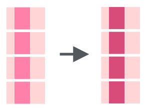
str_trunc(string, width, side = c("right", "left", "center"), ellipsis = "...") Truncate the width of strings, replacing content with ellipsis. str_trunc(sentences, 6)



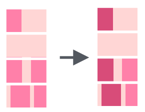
str_trim(string, side = c("both", "left", "right")) Trim whitespace from the start and/or end of a string. str_trim(str_pad(fruit, 17))

str_squish(string) Trim whitespace from each end and collapse multiple spaces into single spaces. str_squish(str_pad(fruit, 17, "both"))

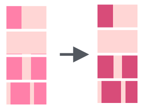
Mutate Strings



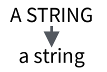
str_sub() <- value. Replace substrings by identifying the substrings with str_sub() and assigning into the results. str_sub(fruit, 1, 3) <- "str"



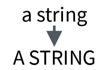
str_replace(string, **pattern**, replacement) Replace the first matched pattern in each string. Also **str_remove()**. str_replace(fruit, "p", "-")



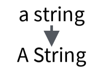
str_replace_all(string, **pattern**, replacement) Replace all matched patterns in each string. Also **str_remove_all()**. str_replace_all(fruit, "p", "-")



str_to_lower(string, locale = "en")[1] Convert strings to lower case. str_to_lower(sentences)

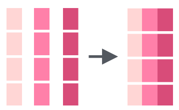


str_to_upper(string, locale = "en")[1] Convert strings to upper case. str_to_upper(sentences)

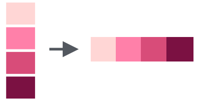


str_to_title(string, locale = "en")[1] Convert strings to title case. Also **str_to_sentence()**. str_to_title(sentences)

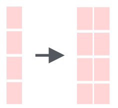
Join and Split



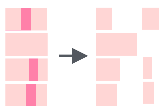
str_c(..., sep = "", collapse = NULL) Join multiple strings into a single string. str_c(letters, LETTERS)



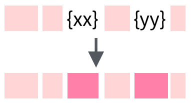
str_flatten(string, collapse = "") Combines into a single string, separated by collapse. str_flatten(fruit, ", ")



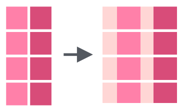
str_dup(string, times) Repeat strings times times. Also **str_unique()** to remove duplicates. str_dup(fruit, times = 2)



str_split_fixed(string, **pattern**, n) Split a vector of strings into a matrix of substrings (splitting at occurrences of a pattern match). Also **str_split()** to return a list of substrings and **str_split_n()** to return the nth substring. str_split_fixed(sentences, " ", n=3)

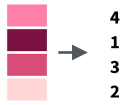


str_glue(..., .sep = "", .envir = parent.frame()) Create a string from strings and {expressions} to evaluate. str_glue("Pi is {pi}")

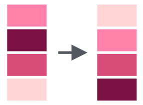


str_glue_data(.x, ..., .sep = "", .envir = parent.frame(), .na = "NA") Use a data frame, list, or environment to create a string from strings and {expressions} to evaluate. str_glue_data(mtcars, "{rownames(mtcars)} has {hp} hp")

Order Strings



str_order(x, decreasing = FALSE, na_last = TRUE, locale = "en", numeric = FALSE, ...)[1] Return the vector of indexes that sorts a character vector. fruit[str_order(fruit)]



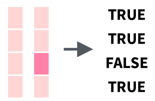
str_sort(x, decreasing = FALSE, na_last = TRUE, locale = "en", numeric = FALSE, ...)[1] Sort a character vector. str_sort(fruit)

Helpers

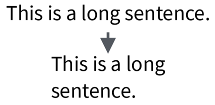
str_conv(string, encoding) Override the encoding of a string. str_conv(fruit,"ISO-8859-1")

appl<e>
banana
p<e>ar

str_view_all(string, **pattern**, match = NA) View HTML rendering of all regex matches. Also **str_view()** to see only the first match. str_view_all(sentences, "[aeiou]")



str_equal(x, y, locale = "en", ignore_case = FALSE, ...)[1] Determine if two strings are equivalent. str_equal(c("a", "b"), c("a", "c"))



str_wrap(string, width = 80, indent = 0, exdent = 0) Wrap strings into nicely formatted paragraphs. str_wrap(sentences, 20)

[1] See **bit.ly/ISO639-1** for a complete list of locales.

RStudio® is a trademark of RStudio, PBC • CC BY SA RStudio • info@rstudio.com • 844-448-1212 • rstudio.com • Learn more at **stringr.tidyverse.org** • Diagrams from **@LVaudor** on Twitter • stringr 1.4.0+ • Updated: 2021-08

Need to Know

Pattern arguments in stringr are interpreted as regular expressions *after any special characters have been parsed.*

In R, you write regular expressions as *strings*, sequences of characters surrounded by quotes (**""**) or single quotes(**''**).

Some characters cannot be represented directly in an R string . These must be represented as **special characters**, sequences of characters that have a specific meaning., e.g.

Special Character	Represents
\\\\	\\
\\"	"
\n	new line

Run **?"'"** to see a complete list

Because of this, whenever a \ appears in a regular expression, you must write it as \\\\ in the string that represents the regular expression.

Use **writeLines()** to see how R views your string after all special characters have been parsed.

writeLines("\\\\.")
# \\.

writeLines("\\\\ is a backslash")
# \\ is a backslash

INTERPRETATION

Patterns in stringr are interpreted as regexs. To change this default, wrap the pattern in one of:

regex(pattern, ignore_case = FALSE, multiline = FALSE, comments = FALSE, dotall = FALSE, ...**)** Modifies a regex to ignore cases, match end of lines as well of end of strings, allow R comments within regex's , and/or to have . match everything including \n.
str_detect("I", regex("i", TRUE))

fixed() Matches raw bytes but will miss some characters that can be represented in multiple ways (fast). str_detect("\u0130", fixed("i"))

coll() Matches raw bytes and will use locale specific collation rules to recognize characters that can be represented in multiple ways (slow). str_detect("\u0130", coll("i", TRUE, locale = "tr"))

boundary() Matches boundaries between characters, line_breaks, sentences, or words. str_split(sentences, boundary("word"))

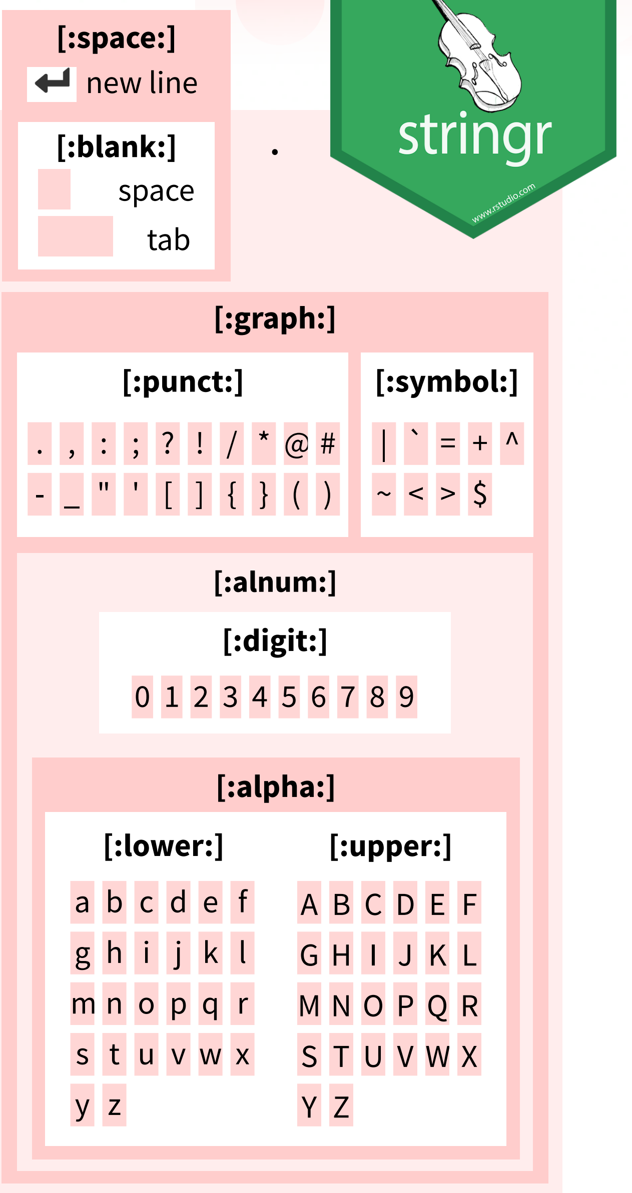
Regular Expressions - Regular expressions, or *regexps*, are a concise language for describing patterns in strings.

MATCH CHARACTERS

see <- function(rx) str_view_all("abc ABC 123\t.!?\\\\(){}\n", rx)

string (type this)	regexp (to mean this)	matches (which matches this)	example	
	a (etc.)	a (etc.)	see("a")	abc ABC 123 .!?\(){}
\\\\.	**\\.**	.	see("\\\\.")	abc ABC 123 .!?\(){}
\\\\!	**\\!**	!	see("\\\\!")	abc ABC 123 .!?\(){}
\\\\?	**\\?**	?	see("\\\\?")	abc ABC 123 .!?\(){}
\\\\\\\\	\\	\\	see("\\\\\\\\")	abc ABC 123 .!?\(){}
\\\\(	**\\(**	(	see("\\\\(")	abc ABC 123 .!?\(){}
\\\\)	**\\)**	)	see("\\\\)")	abc ABC 123 .!?\(){}
\\\\{	**\\{**	{	see("\\\\{")	abc ABC 123 .!?\(){}
\\\\}	**\\}**	}	see("\\\\}")	abc ABC 123 .!?\(){}
\\\\n	**\n**	new line (return)	see("\\\\n")	abc ABC 123 .!?\(){}
\\\\t	**\t**	tab	see("\\\\t")	abc ABC 123 .!?\(){}
\\\\s	**\s**	any whitespace (**\S** *for non-whitespaces*)	see("\\\\s")	abc ABC 123 .!?\(){}
\\\\d	**\d**	any digit (**\D** *for non-digits*)	see("\\\\d")	abc ABC 123 .!?\(){}
\\\\w	**\w**	any word character (**\W** *for non-word chars*)	see("\\\\w")	abc ABC 123 .!?\(){}
\\\\b	**\b**	word boundaries	see("\\\\b")	abc ABC 123 .!?\(){}
	[:digit:][1]	digits	see("[:digit:]")	abc ABC 123 .!?\(){}
	[:alpha:][1]	letters	see("[:alpha:]")	abc ABC 123 .!?\(){}
	[:lower:][1]	lowercase letters	see("[:lower:]")	abc ABC 123 .!?\(){}
	[:upper:][1]	uppercase letters	see("[:upper:]")	abc ABC 123 .!?\(){}
	[:alnum:][1]	letters and numbers	see("[:alnum:]")	abc ABC 123 .!?\(){}
	[:punct:][1]	punctuation	see("[:punct:]")	abc ABC 123 .!?\(){}
	[:graph:][1]	letters, numbers, and punctuation	see("[:graph:]")	abc ABC 123 .!?\(){}
	[:space:][1]	space characters (i.e. \s)	see("[:space:]")	abc ABC 123 .!?\(){}
	[:blank:][1]	space and tab (but not new line)	see("[:blank:]")	abc ABC 123 .!?\(){}
	.	every character except a new line	see(".")	abc ABC 123 .!?\(){}

[1] Many base R functions require classes to be wrapped in a second set of [ ], e.g. **[[:digit:]]**


[:space:]
↵ new line
[:blank:]
space
tab
.
[:graph:]
[:punct:]
. , : ; ? ! / * @ #
- _ " ' [ ] { } ( )
[:symbol:]
| ` = + ^
~ < > $
[:alnum:]
[:digit:]
0 1 2 3 4 5 6 7 8 9
[:alpha:]
[:lower:]
a b c d e f
g h i j k l
m n o p q r
s t u v w x
y z
[:upper:]
A B C D E F
G H I J K L
M N O P Q R
S T U V W X
Y Z

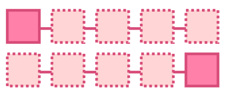
ALTERNATES

alt <- function(rx) str_view_all("abcde", rx)

regexp	matches	example	
ab|d	or	alt("ab|d")	abcde
[abe]	one of	alt("[abe]")	abcde
[^abe]	anything but	alt("[^abe]")	abcde
[a-c]	range	alt("[a-c]")	abcde

ANCHORS

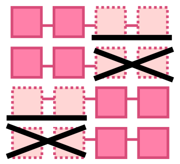
anchor <- function(rx) str_view_all("aaa", rx)



regexp	matches	example	
^a	start of string	anchor("^a")	aaa
a$	end of string	anchor("a$")	aaa

LOOK AROUNDS

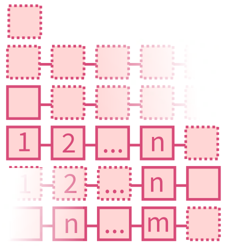
look <- function(rx) str_view_all("bacad", rx)



regexp	matches	example	
a(?=c)	followed by	look("a(?=c)")	bacad
a(?!c)	not followed by	look("a(?!c)")	bacad
(?<=b)a	preceded by	look("(?<=b)a")	bacad
(?<!b)a	not preceded by	look("(?<!b)a")	bacad

QUANTIFIERS

quant <- function(rx) str_view_all(".a.aa.aaa", rx)



regexp	matches	example	
a?	zero or one	quant("a?")	.a.aa.aaa
a*	zero or more	quant("a*")	.a.aa.aaa
a+	one or more	quant("a+")	.a.aa.aaa
a{n}	exactly **n**	quant("a{2}")	.a.aa.aaa
a{n, }	**n** or more	quant("a{2,}")	.a.aa.aaa
a{n, m}	between **n** and **m**	quant("a{2,4}")	.a.aa.aaa

GROUPS

ref <- function(rx) str_view_all("abbaab", rx)

Use parentheses to set precedent (order of evaluation) and create groups

regexp	matches	example	
(ab|d)e	sets precedence	alt("(ab|d)e")	abcde

Use an escaped number to refer to and duplicate parentheses groups that occur earlier in a pattern. Refer to each group by its order of appearance

string (type this)	regexp (to mean this)	matches (which matches this)	example (the result is the same as ref("abba"))	
\\\\1	**\1** (etc.)	first () group, etc.	ref("(a)(b)\\\\2\\\\1")	abbaab

RStudio