Regular Expressions

Mehmet Ali Erkan

Introduction

Regular Expression, a concise and powerful language for describing patterns within strings.

```
[^ ]*?@[^ ]*?\.[^ ]*
```

The chapter initiates with the basics of regular expressions.

The most helpful **stringr** functions for data analysis and the fundamentals of regular expressions are covered in the chapter.

Seven important new topics will be explained (escaping, anchoring, character classes, shorthand classes, quantifiers, precedence, and grouping)

Next, some of the other types of patterns that **stringr** functions can work with and the various “flags” that allow us to tweak the operation of regular expressions can be explained.

Prerequisites

Pattern Basics

str_view() will be used to learn how **regex** patterns work.

Let's create some strings.

```
my_name <- "mehmet ali"
my_surname <- "erkan"
favorite_song <- "burning of the midnight lamp"
favorite_film <- "stalker"
favorite_team <- "fenerbahce"
favorite_city <- "antalya"

my_favorite <- c(my_name, my_surname, favorite_song,
favorite_film,favorite_team,
favorite_city)
```

Let's find the words that Includes "er"

```
str_view(my_favorite, "er") # includes "er"

## [2] │ <er>kan
## [4] │ stalk<er>
## [5] │ fen<er>bahce
```

*Letters and numbers match exactly and are called **literal characters**.*

*Most punctuation characters, like ., +, *, [,], and ? .*

*These are called **metacharacters**.*

For example, . will match any character , so "a." will match any string that contains an "a" followed by another character :

Starts with a and continue with another character.

```
str_view(my_favorite, "a.") # starts with "a"

## [1] │ mehmet <al>i
## [2] │ erk<an>
## [3] │ burning of the midnight l<am>p
## [4] │ st<al>ker
## [5] │ fenerb<ah>ce
## [6] │ <an>t<al>ya
```

*Letters and numbers match exactly and are called **literal characters**.*

Or we could find all the name that contain an "i", followed by letters, followed by an "g":

```
str_view(my_favorite, "i.g")

## [3] │ burn<ing> of the midnight lamp
```

Quantifiers

Control how many times a pattern can match:

? makes a pattern optional.

+ lets a pattern repeat.

* lets a pattern be optional or repeat.

Examples

dc? matches an "d", optionally followed by d "c"

```
str_view(c("d", "dc", "dcc","ddcc"), "dc?")

## [1] │ <d>
## [2] │ <dc>
## [3] │ <dc>c
## [4] │ <d><dc>c
```

dc+ matches an "d", followed by at least one "c".

```
str_view(c("d", "dc", "dcc"), "dc+")

## [2] │ <dc>
## [3] │ <dcc>
```

dc matches an "d", followed by any number of "c"s.*

```
str_view(c("d", "dc", "dcc"), "dc*")

## [1] │ <d>
## [2] │ <dc>
## [3] │ <dcc>
```

***Character classes** are defined by [ ] and match a set of characters,*

e.g. [aei]matches "a", "e", "i".

*Also, invert the match by starting with [^aeifg] matches anything **except** "a", "e", "i", "f" or"g".*

We can use this idea to find the words containing a "n" surrounded by vowels, or a "d" surrounded by consonants:

```
str_view(my_favorite, "[aei]n[aei]")

## [5] │ f<ene>rbahce

str_view(my_favorite, "[^mei]l[^mei]")

## [3] │ burning of the midnight< la>mp
## [4] │ st<alk>er
## [6] │ ant<aly>a
```

Alternation

Alternation can be used to pick between one or more alternative patterns. For example, the following patterns look for some meaningful words in our set.

```
str_view(my_favorite, "ali|burn|night")

## [1] │ mehmet <ali>
## [3] │ <burn>ing of the mid<night> lamp
```

Regular expressions are very intensive and use a lot of punctuation characters, so they can seem overwhelming and hard to read at first. However, with practice, it can be better.

For the practice, please use the below datasets:

https://www.kaggle.com/datasets/ayessa/salary-prediction-classification

```
salary <- read.csv("salaryy.csv")
```

Example: Please find the name of countries that includes d between letters "a", "e", "e", "f", and "g" .

```
str_view(unique(salary$native.country), "[aeifg]d[aeifg]")

## [11] │  Can<ada>
## [35] │  Trin<ada>d&Tobago
```