**[NLP] Basics: Understanding Regular Expressions**

**https://towardsdatascience.com/nlp-basics-understanding-regular-expressions-fc7c7746bc70**

**The only guide you’ll ever need**

When I started learning natural language processing, regular expressions truly felt like a foreign language. I struggled to understand the syntax and it would take me hours to write a regular expression that would return the input I was looking for. Naturally, I tried to stay away from them as long as I could.

But the truth is, as a data scientist, you’ll have to engage with regular expressions one day or the other. They form part of the basic techniques in natural language processing and learning them will go a long way to making you a more efficient programmer.

So it’s time to sit down and get to it. Think of learning regular expressions like a grammar class: they’re painful, it will seem incomprehensible at first but once you understand it and learn it, you’ll feel so relieved it’s behind you. And I promise you, it’s not that hard in the end.

*Note that the programming sotfware I will use in this article is R.*

**Regular expressions unwrapped**

Simply put, a regular expression is ”instruction” given to a function on what and how to match or replace a set of strings.

Let’s start with some basics in regular expressions, that is some basic syntax you should know.

**Brackets** [] are used to specify a disjunction of characters. For instance, using the brackets to put w or W allows me to return a capital W or small w.

/[wW]oodchuck/ --> Woodchuck or woodchuck   
/[abc]/ --> ‘a’, ‘b’, or ‘c’  
/[1234567890]/ --> any digit

If you add a **dash**, you specify a range. For instance, putting A-Z in brackets allows R to return all matches of an upper case letter.

/[A-Z]/ → machtes an upper case letter  
/[a-z]/ → matches a lower case letter  
/[0–9]/ → matches a single digit

The **caret** ^ can be used for negation or just to mean ^.

/[ˆA-Z]/ --> not an upper case letter  
/[ˆSs]/ --> neither ‘S’ nor ‘s’  
/[ˆ\.]/ --> not a period  
/[eˆ]/ --> either ‘e’ or ‘ˆ’  
/aˆb/ --> the pattern ‘aˆb’

The **question mark** ? marks optionality of the previous expression. For instance, putting a ? at the end of the woodchucks returns results for woodchuck (without an s) and woodchucks (with an s).

/woodchucks?/ --> woodchuck or woodchucks  
/colou?r/ --> color or colour

You can use the **period** . to specify any character between two expressions. For instance, putting beg.n will return you words such as begin or begun.

/beg.n/ --> Match any character between beg and n (e.g. begin, begun)

The us of **\* or +**allows you to add 1 or more of a previous character.

oo\*h! → 0 or more of a previous character (e.g. ooh!, ooooh!)  
o+h! → 1 or more of a previous character (e.g. ooh!, ooooooh!)  
baa+ → baa, baaaa, baaaaaa, baaaaaaa

**Anchors** are used to assert something about the string or the matching process. As such, they are not used in a specific word or character but help with more general queries as you can see in the examples below.

. → any character except a new line  
\\w → any word character  
\\W → anything but a word character  
\\d → any digit character  
\\D → anything but a digit character  
\\b → a word boundary  
\\B → anything but a word boundary  
\\s → any space character  
\\S → anything but a space character

**POSIX character classes** are helpful if you want to match a specific character class, e.g. digits. In other words, it makes one small sequence of characters match a larger set of characters.

[[:alpha:]] → alphabetic characters  
[[:digit:]] → digits  
[[:punct:]] → punctuation  
[[:space:]] → space, tab, newline etc.  
[[:lower:]] → lowercase alphatic characters  
[[:upper:]] → upper case alphabetic characters

**strsplit(), grep(), and gsub()**

Now this is where the action starts. When dealing with character strings, it is very likely that you’ll have to use the commands strsplit(), grep() and gsub() to activate the input you want R to return to you.

**strsplit(x, split)**

The example below shows a way of using strsplit to split the words within a sentence, in this case all the words within the dashes “ …”.

richard3 <- “now is the winter of our discontent”  
strsplit(richard3, “ “) # the second argument specifies the space

**grep(pattern, x, ignore.case = FALSE, value = FALSE)**

grep allows you to “grab” the word or set of words you want, depending on the matching pattern you set. For instance, in the code below I ask R to return me the character string that contains the word “both”.

grep.ex <- c(“some like Python”, “some prefer R”, “some like both”)grep(“both”, grep.ex) # in numerical form  
grep(“both”, grep.ex, value=TRUE) #prints the character string itself

**gsub(pattern, replacement, x, ignore.case= FALSE)**

gsub allows you to, for instance, replace a word by another one. In this case, I chose to replace Romeo by Superman in the character string below.

r\_and\_j <- “O Romeo, Romeo, wherefore art thou Romeo?”  
gsub(“Romeo”, “Superman”, r\_and\_j, ignore.case = FALSE)

**Regex applied**

Let’s start applying these commands to regular expressions. I’ll show you a couple of examples below.

**1. grep(pattern, x, ignore.case = FALSE, value = FALSE)**

dollar <- c(“I paid $15 for this book.”, “they received $50,000 in grant money”, “two dollars”)

Note that it the example above you have three different sentences, two of them use the $ sign and one of them uses the word “dollars”. Using only $ to match your pattern will yield to all three sentences being returned. However, if you add the \\ before the $ you can specify you only want the sentences using the $ sign.

grep(“$”, dollar)   
grep(“\\$”, dollar)

Here are a few other examples below of how to use grep matching your regular expressions on the words “ashes”, “shark”, “bash”:

# matches all three vector elements  
grep(“sh”, c(“ashes”, “shark”, “bash”), value=TRUE) # matches only “shark”  
grep(“^sh”, c(“ashes”, “shark”, “bash”), value=TRUE) # matches only “bash”  
grep(“sh$”, c(“ashes”, “shark”, “bash”), value=TRUE)

On the words “gaffe”, “chafe” and “chalk”:

quant.ex <- c(“gaffe”, “chafe”, “chalk”, “encyclopaedia”, “encyclopedia”)# Searching for one or more occurences of f and we want to see the value not only the index (that’s why we put value = TRUE)  
grep(“f+”, quant.ex, value=TRUE) # one or two “f”  
grep(“f{1,2}”, quant.ex, value=TRUE) # at least one “f”  
grep(“f{1,}”, quant.ex, value=TRUE)

**2. gsub(pattern, replacement, x, ignore.case= FALSE)**

In the examples below, you can play around with gsub to replace words in your sentence with anything you want.

ex.sentence <- “Act 3, scene 1. To be, or not to be, that is the Question:”

If you remember the regular expressions you learned in the first part of this article, you should be able to guess what kind of input R will return. Otherwise, I’ve added it to the code below for you to better understand all the different ways to use regular expressions to return what you want.

gsub(“.”, “\*”, ex.sentence, ignore.case=TRUE, perl=TRUE)  
[1] "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"gsub(“\\w”, “\*”, ex.sentence, ignore.case=TRUE, perl=TRUE)  
[1] "\*\*\* \*, \*\*\*\*\* \*. \*\* \*\*, \*\* \*\*\* \*\* \*\*, \*\*\*\* \*\* \*\*\* \*\*\*\*\*\*\*\*:"gsub(“\\W”, “\*”, ex.sentence, ignore.case=TRUE, perl=TRUE)  
[1] "Act\*3\*\*scene\*1\*\*To\*be\*\*or\*not\*to\*be\*\*that\*is\*the\*Question\*"gsub(“\\d”, “\*”, ex.sentence, ignore.case=TRUE, perl=TRUE)  
[1] "Act \*, scene \*. To be, or not to be, that is the Question:"gsub(“\\D”, “\*”, ex.sentence, ignore.case=TRUE, perl=TRUE)  
[1] "\*\*\*\*3\*\*\*\*\*\*\*\*1\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"gsub(“\\b”, “\*”, ex.sentence, ignore.case=TRUE, perl=TRUE)  
[1] "\*Act\* \*3\*, \*scene\* \*1\*. \*To\* \*be\*, \*or\* \*not\* \*to\* \*be\*, \*that\* \*is\* \*the\* \*Question\*:"gsub(“\\B”, “\*”, ex.sentence, ignore.case=TRUE, perl=TRUE)  
[1] "A\*c\*t 3,\* s\*c\*e\*n\*e 1.\* T\*o b\*e,\* o\*r n\*o\*t t\*o b\*e,\* t\*h\*a\*t i\*s t\*h\*e Q\*u\*e\*s\*t\*i\*o\*n:\*"gsub(“\\s”, “\*”, ex.sentence, ignore.case=TRUE, perl=TRUE)  
[1] "Act\*3,\*scene\*1.\*To\*be,\*or\*not\*to\*be,\*that\*is\*the\*Question:"gsub(“\\S”, “\*”, ex.sentence, ignore.case=TRUE, perl=TRUE)  
[1] "\*\*\* \*\* \*\*\*\*\* \*\* \*\* \*\*\* \*\* \*\*\* \*\* \*\*\* \*\*\*\* \*\* \*\*\* \*\*\*\*\*\*\*\*\*"

Otherwise, try to guess the answer to the last one:

letters.digits <- “a1 b2 c3 d4 e5 f6 g7 h8 i9”  
gsub(“(\\w)(\\d)”, “\\2\\1”, letters.digits)

That’s it! I hope you’ve enjoyed this article and that I’ve managed to make regular expression a bit more easier for you to understand.

*I regularly write articles about Data Science and Natural Language Processing. Follow me on* [*Twitter*](https://twitter.com/celine_vdr) *or* [*Medium*](https://medium.com/@celine.vdr) *to check out more articles like these or simply to keep updated about the next ones.* ***Thanks for reading!***

PS: the answer to the last one is “1a 2b 3c 4d 5e 6f 7g 8h 9i”. In other words, you’ve re-written all the letters and digits using regular expressions. Pretty cool no?