## Introduction

Text data is messy! Earlier we could match and extract the required information from the given text data using Ctrl + F, Ctrl + C, and Ctrl + V. Isn't it ? Probably, some of us still do it when the data is small. But this approach is slow and prone to lots of mistakes. In text analytics, the abundance of data makes such keyboard shortcut hacks obsolete.

We need better ways. Because of the data volume and its complicated (unstructured) nature, we require much faster, convenient, and robust ways of information extraction from text data. Just imagine, the amount of text data being generated on Twitter and Facebook every day.

Moreover, the text data we get is noisy. But, if we can learn some methods useful to extract important features from the noisy data, wouldn't that be amazing ?

In this tutorial, you'll learn all about regular expressions from scratch. At first, you might find these expressions tricky, confusing, or complicated, but after doing practical hands-on exercises (done below) you should feel quite comfortable with it. In addition, we'll also learn about string manipulation functions in R. This formidable combination of string manipulation functions and regular expressions will prepare you for text mining.

For better understanding, I've also added practice exercises on regular expressions at the end.

## Table of Contents

## What are Regular Expressions ? When to use them ?

Regular Expressions (a.k.a regex) are a set of pattern matching commands used to detect string sequences in a large text data. These commands are designed to match a family (alphanumeric, digits, words) of text which makes then versatile enough to handle any text / string class.

In short, using regular expressions you can get more out of text data while writing shorter codes.

For example, let's say you've scraped some data from the web. The data contains the log time of users. You want to extract log time. But, the data is really messy. It is contaminated with html div(s), javascript functions, and what not! In such situations, you should use regular expressions.

Other than R, regular expressions are also available in Python, Ruby, Perl, Java, Javascript, etc.

## What is String Manipulation ?

As the name suggests, string manipulation comprises a series of functions used to extract information from text variables. In machine learning, these functions are being widely used for doing feature engineering, i.e., to create new features out of existing string features. In R, we have packages such as stringr and stringi which are loaded with all string manipulation functions.

In addition, R also comprises several base functions for string manipulations. These functions are designed to complement regular expressions. The practical differences between string manipulation functions and regular expressions are

1. We use string manipulation functions to do simple tasks such as splitting a string, extracting the first three letters, etc.. We use regular expressions to do more complicated tasks such as extract email IDs or date from a set of text.
2. String manipulation functions are designed to respond in a certain way. They don't deviate from their natural behavior. Whereas, we can customize regular expressions in any way we want.

For example, suppose you are given a data set comprising the name of the customer as a variable. In this case, we can use string manipulation functions to extract and create new features as first name and last name. From the next section onward, we'll learn string manipulation functions and commands practically. So, make sure you've R installed in your machine. Also, you should install stringr R package.

## List of String Manipulation Functions

In R, a string is any value enclosed in quotes (" "). Yes, you can even have number as strings. R notifies strings under the class character. Let's see!

```
text <- "san francisco"
typeof(text)
[1] "character"
```

```
num <- c("24","34","36")
typeof(num)
[1] "character"
```

R's base paste function is used to combine (or paste) set of strings. In machine learning, it is quite frequently used in creating / re-structuring variable names. For example, let's say, you want to use two strings (Var1 and Var2) to create a new string Var3. For neatness, we'll separate the resultant values using a - (hyphen).

```
var3 <- paste("Var1","Var2",sep = "-")
var3
[1] "Var1-Var2"
```

In this paste function, you can pass a vector of values and if the vector lengths aren't equal, this function will recycle the length of the shorter vector till it matches the length of longer vector. In the example below, I've taken a vector of length 5 (1:5) and combined it with a vector of length 2 which consists of c("?","!"):

```
paste(1:5,c("?","!"),sep = "-")
[1] "1-?" "2-!" "3-?" "4-!" "5-?"
```

As you saw above, all the outputs are returned within quotes, thus making them a character class. Alternatively, R also allows you to print and concatenate strings without quotes. It is done using the cat function. In stringr package, its substitute function is str_c() or str_join().

```
cat(text,"USA",sep = "-")
san francisco-USA
```

```
cat(month.name[1:5],sep = " ")
January February March April May
```

The toString function allows you to convert any non-character value to a string.

```
toString (1:10)
[1] "1,2,3,4,5,6,7,8,9,10"
```

Now, let's look at some of the commonly used base R functions (also available in stringr) to modify strings:

| Functions | Description |
|---|---|
| nchar() | It counts the number of characters in a string or vector. In the stringr package, it's substitute function is str_length() |

| Functions | Description |
|---|---|
| tolower() | It converts a string to the lower case. Alternatively, you can also use the str_to_lower() function |
| toupper() | It converts a string to the upper case. Alternatively, you can also use the str_to_upper() function |
| chartr() | It is used to replace each character in a string. Alternatively, you can use str_replace() function to replace a complete string |
| substr() | It is used to extract parts of a string. Start and end positions need to be specified. Alternatively, you can use the str_sub() function |
| setdiff() | It is used to determine the difference between two vectors |
| setequal() | It is used to check if the two vectors have the same string values |
| abbreviate() | It is used to abbreviate strings. The length of abbreviated string needs to be specified |
| strsplit() | It is used to split a string based on a criterion. It returns a list. Alternatively, you can use the str_split() function. This function lets you convert your list output to a character matrix |
| sub() | It is used to find and replace the first match in a string |
| gsub() | It is used to find and replace all the matches in a string / vector. Alternatively, you can use the str_replace() function |

To look at the list of all functions contained in the stringr package, go here.

Now, let's write these functions and understand their effect on strings. I haven't shown their outputs. You are expected to run these commands locally and understand the difference.

```
library(stringr)
string <- "Los Angeles, officially the City of Los Angeles and often known by its initials L.A., is the second-most populous city in the United States (after New York City), the most populous city in California and the county seat of Los Angeles County. Situated in Southern California, Los Angeles is known for its Mediterranean climate, ethnic diversity, sprawling metropolis, and as a major center of the American entertainment industry."

strwrap(string)

#count number of characters
nchar(string)
str_length(string)

#convert to lower
tolower(string)
str_to_lower(string)

#convert to upper
toupper(string)
str_to_upper(string)

#replace strings
chartr("and","for",x = string) #letters a,n,d get replaced by f,o,r
str_replace_all(string = string, pattern = c("City"),replacement = "state") #this is case sentitive

#extract parts of string
`substr(x = string,start = 5,stop = 11)

#extract angeles str_sub(string = string, start = 5, end = 11)

#get difference between two vectors
setdiff(c("monday","tuesday","wednesday"),c("monday","thursday","friday"))

#check if strings are equal
setequal(c("monday","tuesday","wednesday"),c("monday","tuesday","wednesday"))
setequal(c("monday","tuesday","thursday"),c("monday","tuesday","wednesday"))
```

```
#abbreviate strings
abbreviate(c("monday","tuesday","wednesday"),minlength = 3)

#split strings
strsplit(x = c("ID-101","ID-102","ID-103","ID-104"),split = "-")
str_split(string = c("ID-101","ID-102","ID-103","ID-104"),pattern = "-",simplify = T)

#find and replace first match
sub(pattern = "L",replacement = "B",x = string,ignore.case = T)

#find and replace all matches
gsub(pattern = "Los",replacement = "Bos",x = string,ignore.case = T)
```

The pattern parameter in the functions above also accept regular expressions. These functions when combined with regular expressions can do highly complex search operations. Now, let's learn about regular expressions.

## List of Regular Expression Commands

Apart from the function listed above, there are several other functions specially designed to deal with regular expressions (a.k.a regex). Yes, R is equally powerful when it comes to parsing text data. In regex, there are multiple ways of doing a certain task. Therefore, while learning, it's essential for you to stick to a particular method to avoid confusion.

For using regular expressions, the available base regex functions are grep(), grepl(), regexpr(), gregexpr(), regexec(), and regmatches(). Here's a quick preview of these functions:

| Function | Description |
|---|---|
| grep | returns the index or value of the matched string |
| grepl | returns the Boolean value (True or False) of the matched string |
| regexpr | return the index of the first match |
| gregexpr | returns the index of all matches |
| regexec | is a hybrid of regexpr and gregexpr |
| regmatches | returns the matched string at a specified index. It is used in conjunction with regexpr and gregexpr. |

Regular expressions in R can be divided into 5 categories:

1. Metacharacters
2. Sequences
3. Quantifiers
4. Character Classes
5. POSIX character classes

Let's understand them in detail.

### 1. Metacharacters

Metacharacters comprises a set of special operators which regex doesn't capture. Yes, regex work by its own rules. These operators are most common in every line of text you would come across. These characters include: . \ | ( ) [ ] { } $ * + ?

If any of these characters are available in a string, regex won't detect them unless they are prefixed with double backslash (\) in R. Let's see how to escape these characters in R.

From a given vector, we want to detect the string "percent%." We'll use the base grep() function used to detect strings given a pattern. Also. we'll use the gsub() function to make the replacements. We can do it like this:

```
dt <- c("percent%","percent")
grep(pattern = "percent\\%",x = dt,value = T)
[1] "percent%"
```

```
#detect all strings
dt <- c("may?","money$","and&")
grep(pattern = "[a-z][\\?-\\$-\\&]",x = dt,value = T)
[1] "may?" "money$" "and&"
```

```
gsub(pattern = "[\\?-\\$-\\&]",replacement = "",x = dt)
[1] "may" "money" "and"
```

In fact, if you find a double backslash in a string, you'll need to prefix it with another double backslash to get detected. Following is an example:

```
gsub(pattern = "\\\\",replacement = "-",x = "Barcelona\\Spain")
[1] "Barcelona-Spain"
```

## 2. Quantifiers

Quantifiers are the shortest to type, but these tiny atoms are immensely powerful. One position here and there can change the entire output value. Quantifiers are mainly used to determine the length of the resulting match. Always remember, that quantifiers exercise their power on items to the immediate left of it. Following is the list of quantifiers commonly used in detecting patterns in text: It matches everything except a newline.

| Quantifier | Description |
|---|---|
| . | It matches everything except a newline. |
| ? | The item to its left is optional and is matched at most once. |
| * | The item to its left will be matched zero or more times. |
| + | The item to its left is matched one or more times. |
| {n} | The item to its left is matched exactly n times. The item must have a consecutive repetition at place. e.g. Anna |
| {n, } | The item to its left is matched n or more times. |
| {n,m} | The item to its left is matched at least n times but not more than m times. |

These quantifiers can be used with metacharacters, sequences, and character classes to return complex patterns. Combinations of these quantifiers help us match a pattern. The nature of these quantifiers is better known in two ways:

- Greedy Quantifiers : The symbol .* is known as a greedy quantifier. It says that for a particular pattern to be matched, it will try to match the pattern as many times as its repetition are available.
- Non-Greedy Quantifiers : The symbol .? is known as a non-greedy quantifier. Being non-greedy, for a particular pattern to be matched, it will stop at the first match.

Let's look at an example of greedy vs. non-greedy quantifier. From the given number, apart from the starting digit, we want to extract this number till the next digit '1' is detected. The desired result is 101.

```
number <- "101000000000100"
```

```
#greedy
regmatches(number, gregexpr(pattern = "1.*1",text = number))
[1] "1010000000001"
```

```
#non greedy
regmatches(number, gregexpr(pattern = "1.?1",text = number))
[1] "101"
```

It works like this: the greedy match starts from the first digit, moves ahead, and stumbles on the second '1' digit. Being greedy, it continues to search for '1' and stumbles on the third '1' in the number. Then, it continues to check further but couldn't find more. Hence, it returns the result as "1010000000001." On the other hand, the non-greedy quantifier, stops at the first match, thus returning "101."

Let's look at a few more examples of quantifiers:

```
names <- c("anna","crissy","puerto","cristian","garcia","steven","alex","rudy")

#doesn't matter if e is a match
grep(pattern = "e*",x = names,value = T)
[1] "anna" "crissy" "puerto" "cristian" "garcia" "steven" "alex" "rudy"

#must match t one or more times
grep(pattern = "t+",x = names,value = T)
[1] "puerto" "cristian" "steven"

#must match n two times
grep(pattern = "n{2}",x = names,value = T)
[1] "anna"
```

## 3. Sequences

As the name suggests, sequences contain special characters used to describe a pattern in a given string. Following are the commonly used sequences in R:

| Sequences | Description |
|---|---|
| \d | matches a digit character |
| \D | matches a non-digit character |
| \s | matches a space character |
| \S | matches a non-space character |
| \w | matches a word character |
| \W | matches a non-word character |
| \b | matches a word boundary |
| \B | matches a non-word boundary |

Let's look at some examples:

```
string <- "I have been to Paris 20 times"

#match a digit
gsub(pattern = "\\d+",replacement = "_",x = string)
regmatches(string,regexpr(pattern = "\\d+",text = string))

#match a non-digit
gsub(pattern = "\\D+",replacement = "_",x = string)
regmatches(string,regexpr(pattern = "\\D+",text = string))

#match a space - returns positions
gregexpr(pattern = "\\s+",text = string)

#match a non space
gsub(pattern = "\\S+",replacement = "app",x = string)

#match a word character
gsub(pattern = "\\w",replacement = "k",x = string)

#match a non-word character
gsub(pattern = "\\W",replacement = "k",x = string)
```

## 4. Character Classes

Character classes refer to a set of characters enclosed in a square bracket [ ]. These classes match only the characters enclosed in the bracket. These classes can also be used in conjunction with quantifiers. The use of the caret (^) symbol in character classes is interesting. It negates the expression and searches for everything except the specified pattern. Following are the types of character classes used in regex:

| Characters | Description |
|---|---|
| [aeiou] | matches lower case vowels |
| [AEIOU] | matches upper case vowels |
| [0123456789] | matches any digit |
| [0-9] | same as the previous class |
| [a-z] | match any lower case letter |
| [A-Z] | match any upper case letter |
| [a-zA-Z0-9] | match any of the above classes |
| [^aeiou] | matches everything except letters |
| [^0-9] | matches everything except digits |

Let's look at some examples using character classes:

```
string <- "20 people got killed in the mob attack. 14 got severely injured"

#extract numbers
regmatches(x = string,gregexpr("[0-9]+",text = string))

#extract without digits
regmatches(x = string,gregexpr("[^0-9]+",text = string))
```

5. POSIX Character Classes

In R, these classes can be identified as enclosed within a double square bracket ([[ ]]). They work like character classes. A caret ahead of an expression negates the expression value. I find these classes more intuitive than the rest, and hence easier to learn. Following are the posix character classes available in R:

| POSIX Characters | Description |
|---|---|
| [[:lower:]] | matches lower case letter |
| [[:upper:]] | matches upper case letter |
| [[:alpha:]] | matches letters |
| [[:digit:]] | matches digits |
| [[:space:]] | matches space characters eg. tab, newline, vertical tab, space, etc |
| [[:blank:]] | matches blank characters (same as previous) such as space, tab |
| [[:alnum:]] | matches alphanumeric characters, e.g. AB12, ID101, etc |
| [[:cntrl:]] | matches control characters. Control characters are non-printable characters such as \t (tab), \n (new line), \e (escape), \f (form feed), etc |
| [[:punct:]] | matches punctuation characters |
| [[:xdigit:]] | matches hexadecimal digits (0 - 9 A - E) |
| [[:print:]] | matches printable characters ([[:alpha:]] [[:punct:]] and space) |
| [[:graph:]] | matches graphical characters. Graphical characters comprise [[:alpha:]] and [[:punct:]] |

Let's look at some of the examples of this regex class:

```
string <- c("I sleep 16 hours\n, a day","I sleep 8 hours\n a day.","You sleep how many\t hours ?")
```

```
#get digits
unlist(regmatches(string,gregexpr("[[:digit:]]+",text = string)))

#remove punctuations
gsub(pattern = "[[:punct:]]+",replacement = "",x = string)

#remove spaces
gsub(pattern = "[[:blank:]]",replacement = "-",x = string)

#remove control characters
gsub(pattern = "[[:cntrl:]]+",replacement = " ",x = string)

#remove non graphical characters
gsub(pattern = "[^[:graph:]]+",replacement = "",x = string)
```

Until here, we've learned the basics of regular expressions and string manipulations. I hope you've got enough insights and hands on experience of writing these pattern recognition string commands. In the next section, we'll practice everything we have learned till here using some real-life problems. The questions listed below are the ones you might face while working on text data.

Hence, make sure you understand them well. For best results, I would suggest you code them word by word for better understanding.

## Practice Examples on Regular Expressions

### 1. Extract digits from a string of characters

```
#extract digits - all 4 works
string <- "My roll number is 1006781"
gsub(pattern = "[^0-9]",replacement = "",x = string)
stringi::stri_extract_all_regex(str = string,pattern = "\\d+") #list
regmatches(string, regexpr("[0-9]+",string))
regmatches(string, regexpr("[[:digit:]]+",string))
```

### 2. Remove spaces from a line of strings

```
#remove space
gsub(pattern = "[[:space:]]",replacement = "",x = "and going there today tomorrow")
gsub(pattern = "[[:blank:]]",replacement = "",x = "and going there today tomorrow")
gsub(pattern = "\\s",replacement = "",x = "and going there today tomorrow")
```

### 3. Return if a value is present in a vector

```
#match values
det <- c("A1","A2","A3","A4","A5","A6","A7")
grep(pattern = "A1|A4",x = det,value =T)
```

### 4. Extract strings which are available in key value pairs

```
d <- c("(monday :: 0.1231313213)","tomorrow","(tuesday :: 0.1434343412)")
grep(pattern = "\\([a-z]+ :: (0\\.[0-9]+)\\)",x = d,value = T)
regmatches(d,regexpr(pattern = "\\((.*) :: (0\\.[0-9]+)\\)",text = d))
```

**Explanation:** You might find it complicated to understand, so let's look at it bit by bit. "\(" is used to escape the metacharacter. "[a-z]+" matches letters one or more times. "(0\.[0-9]+)" matches the decimal value, where the metacharacter (.) is escaped using double backslash, so is the period. The numbers are matched using "[0-9]+."

### 5. In a key value pair, extract the values

```
string = c("G1:E001", "G2:E002", "G3:E003")
gsub(pattern = ".*:",replacement = "",x = string)
```

**Explanation:** In the regex above, ".*:" matches everything (except newspace) it can until it reaches colon (:), then gsub() function replaces it with blank. Hence, we get the desired output.

### 6. Remove punctuation from a line of text

```
going <- "a1~!@#$%^&*bcd(){}_+:efg\"<>?,./;'[]-="
gsub(pattern = "[[:punct:]]+",replacement = "",x = going)
```

### 7. Remove digits from a string which contains alphanumeric characters

```
c2 <- "day of 2nd ID5 Conference 19 12 2005"
```

From the string above, the desired output is "day of 2nd ID5 Conference." You can't do it with simple "[[:digit:]]+" regex as it will match all the digits available in the given string. Instead, in such case, we'll detect the digit boundary to get the desired result.

```
gsub(pattern = "\\b\\d+\\b",replacement = "",x = c2)
```

### 8. Find the location of digits in a string

```
string <- "there were 2 players each in 8 teams"
gregexpr(pattern = '\\d',text = string) #or
unlist(gregexpr(pattern = '\\d',text = "there were 2 players each in 8 teams"))
```

### 9. Extract information available inside parentheses (brackets) in a string

```
string <- "What are we doing tomorrow ? (laugh) Play soccer (groans) (cries)"
gsub("[\\(\\)]","",regmatches(string, gregexpr("\\(.*?\\)", string))[[1]])
```

**Explanation:** In this solution, we've used the lazy matching technique. First, using regmatches, we've extracted the parentheses with words such as (cries) (groans) (laugh). Then, we've simply removed the brackets using the gsub() function.

### 10. Extract only the first digit in a range

```
x <- c("75 to 79", "80 to 84", "85 to 89")
gsub(" .*\\d+", "", x)
```

### 11. Extract email addresses from a given string

```
string <- c("My email address is abc@boeing.com","my email address is def@jobs.com","aescher koeif","paul renne")
unlist(regmatches(x = string, gregexpr(pattern = "[[:alnum:]]+\\@[[:alpha:]]+\\.com",text = string)))
```

### Summary

Regular Expressions are a crucial aspect of text mining and natural language processing. In fact, you would often use regular expressions for doing feature engineering on text data. So, make sure you understand them well. After all, we need to learn ways to overcome messy text data.

In this tutorial, you learned about the basics of string manipulations and regular expressions. You learned about several base R functions used to handle strings and regex. I find R language to be immensely powerful for data cleaning and text data manipulations. In the next tutorial, we'll go a step ahead and learn to do text mining in R.