Data Mania Inc.

Basics of Regular Expressions in R

Additional notes for Session Ten

Hellen Gakuruh
1/10/2017

Basics of Regular Expressions in R

Hellen Gakuruh

January 5, 2017

Introduction

Regular expressions are patterns used to match certain text. They are commonly used in find and find-and-replace software's and functions, they are also used to validate text and perform qualitative data analysis. Regular expressions are also extensively for web scraping.

A regular expression (regex) can either be a literal text or text with special characters. How a regular expression is matched to a string depends on regular expression engine used. There are two type of engines, **Deterministic Finite Automation (DFA)** which is text-directed and **Nondeterministic Finite Automation or (NFA)** which is regex-directed. R and most other programming languages uses NFA, hence we will discuss regex in relation to this engine.

Literal Regex

The simplest regular expression is a literal regex. Literals like word "cat" which are matched exactly as they appear, that is, letter "c", followed by letter "a" and finally letter "t". Note, we are not saying a match would be made for the whole word "cat" but rather the individual components of that word, so when this regex (cat) is matched against a string like ***"As a dogcatcher, I can also catch a cat in distress "***, it will get three matches. The first match is the cat in dog**cat**cher, the second is cat in **cat**ch and final one is the word **cat**. If we want to match the word "cat" then we have to use regular expression metacharacters.

All alphanumerical characters and punctuation marks can be matched as literal regex with exception of **backslash (\), *caret (^), dollar sign ($), dot (.), vertical bar (|), question mark (?), asterisk (*)*, plus sign (+), both left and right round brackets ((and))**, **left square bracket ([ )**, **and left curly bracket ({)**. These exceptions are used in regular expression as metacharacters and would have special meaning. To use these metacharacters as literal regex, they need to be escaped and in R this is done using double backslash like 1 \\+ 2. We use double backslash because single backslash are used for short-hand character classes.

Regular Expressions Metacharacters

Regular expression metacharacters are characters with special meaning. Here we look at these metacharacters and what they mean.

Character Classes

Character classes are used to match **one** character out of a list of characters. The list of characters are enclosed in square brackets *"[ ]"* to denote a character class. For example, if we want to match "categorise" or "categorize", we can write a regex categori[sz]e. This way, both variation of spelling can be matched. What is important to remember with character classes is that only one character can be matched.

Character classes can also be used to negate a match by adding a caret at the beginning. For example, a regex 2[^-]3 would match 2 followed by any single character but not a minus sign. This would therefore match "2+3", "2 3", "2*3" or "2/3" but not "2-3" or "2 + 3" (because of the spaces around +).

Character classes can be used to match literal meaning of metacharacters for example "[.]" would match a literal dot and "[+]" would match a literal plus sign. Basically this means metacharacters lose their special meaning when inside a character class (with exception of caret).

Character classes have their own metacharacter, these are **caret (^)**, **hyphen (-)**. To include a caret in a character class, place it anywhere other than the beginning of the character class. To use a hyphen, place it either at the beginning or at the end of the character class. *Question, why are these characters special in a character class?* For a caret as mentioned earlier, it is used to negate characters in a character class. Hyphens are used in character class to list a range of characters like numerical sequence or alphabets. For example, "[1-9]" would match any digit from 1 through 9, and "[a-zA-Z]" would match any lower or upper case letter. *Do remember it can only match one character (digit or alphabet), if we what to match more than one character we will need to used a quantifier.

There are a number of short hand character classes which begin with a backslash followed by an alphabet, for example, instead of writing a character class with a range of all digit like "[0-9]", we can use "\\d". Other short hand character classes include "\\w" which matches any letter or digit, "\\W" which is it's negation and matches any character other than a letter or a digit, "\\s" matches a space and "\\S" is it's negation

Dot

As noted, a **"."** is a metacharacter with a special meaning in regular expression. A dot is used to symbolize **any character**, like letters, digits, spaces, or even punctuation's.

Matching beginning and end of a word or string (Anchors)

Outside a character class, a caret **^** is used to mark the beginning of a string. For example, if we want to match a sentence starting with an "a", then we would use the following regex ^a. You can do this when searching for files beginning with certain letters.

A dollar sign **$** is used to match characters at the end of a string, for example file ending with a **".R"** can be searched with regex [.]R$.

Both caret and dollar sign can be used for specific matches or match exact word. For example, we can look for files that begin with letters "Epi" and end with a ".R" extension. Since we have not discussed quantifiers, we can assume the files would have four other characters after the letters "Epi", hence we have ^Epi....[.]R$.

To match a word like the "cat" from our earlier example of *"As a dogcatcher, I can also catch a **cat** in distress"*, we can use symbols "**\<**" at the beginning and "**\>**" at the end of the word, that is, "**\<cat\>**". By so doing, only one cat can be matched, this will therefore not match cat in *dogcatcher* or cat in *catch*. An alternative to **\<** and **\>** symbol is **\\b** at the beginning and end of the word.

Quantifiers

When using a dot to match any character or a character class to match a character from a list of characters, we might want to match more than one occurrence of these matches. For instance, in our earlier example of file search, instead of confining our search to four characters using four dots, we might want to leave the upper limit open. This would then mean any file starting with "epi" followed by any number of characters and ending with a ".R" extension would match.

There are four way in which a repetition can be specified; by using a **question mark (?)**, **an asterisk (*), a plus sign (+)** or **curly brackets ({})**. These are commonly referred to as **quantifiers**.

Question mark (?)

A question mark is used to **match zero or one** instance of preceding character. For example, if we want to search for files starting with "Epi" followed by one other character then ends with a ".R" extension or a file starting with "Epi" and ending with a ".R" extension, then we can use "^Epi**.?**[.]R$". Below are some files and match outcomes:

File	Outcome
"Epi.R"	A match
"Epi1.R"	A match
"Epidemiology.R"	Not a match

Using a question mark means the preceding character is optional; there will be a match whether it exists or it does not exist.

Asterisk (*)

An asterisk is used to **match zero or more** of preceding character. Like the question mark, using an asterisk means preceding character is optional, however, for an asterisk the upper bound is open, it could be one or more. So in our file example, we can use an asterisk to match zero or more characters after "Epi" but before ending with a ".R", that is **^Epi.*[.]R$**. Here are some files and match outcome:

File	Outcome

"Epi.R"	A match
"Epi1.R"	A match
"Epidemiology.R"	A match

Hope you see an asterisk never fails (as long as the other characters are matched like "Epi" and ".R"), this has it's advantages and disadvantages.

Plus sign (+)

A plus sign is used to **match one or more** of preceding character. This means preceding character is not optional, it must match at least once. Going back to our file example, using a plus sign would mean presence of at least one character character after "Epi" and before ".R" extension for a file to match. Therefore, with regex `^Epi.+[.R]$` we expect the following matches and mismatches:

File	Outcome
"Epi.R"	Not a match
"Epi1.R"	A match
"Epidemiology.R"	A match

Curly Brackets ({})

Curly brackets are used to specify an exact number of matches {n}, or a range of minimum and maximum matches {n,m}. For example `^Epi.{1}[.]R$` would match a file starting with "Epi" followed by one character and ending with ".R" extension.

File	Outcome
"Epi.R"	Not a match
"Epi1.R"	A match
"Epidemiology.R"	Not a match

Alternations and Grouping

Alternative matches can be search using a vertical bar | and encased in parentheses. Parentheses here are used to group different alternatives. For example, in our earlier example on two spelling of categorise, we can express it as `categori(s|z)e` where "s" and "z" are alternatives. Another good example is when you want to draw emphasis to certain words in a text, these words can be grouped as alternatives such that any match found is emphasised.

Capturing and Backreferences

Other than grouping alternatives, parenthesis in regular expression are also used to capture specific pattern. This is quite useful when doing "find-and-replace" pattern search. For example, if we had a date "1/6/2017" and we only need the year part, we can construct a regular expression to match the whole date but capture the year part which will then

replace the date. For this date we can write `\d/\d/(\d{4})` which mean a digit followed by a forward slash, then a digit, then a forward slash and finally four digits. The last four digits are captured with a parentheses.

Captured patterns can be re-used either in the same regular expression or as a replacement of matched pattern. When using captured pattern, means we are doing a **back reference** to a matched pattern. Each captured pattern is numbered for example our four digits would be "1" as it is the first and only captured group, to back reference to this captured pattern we write `\\1` which is the first back reference. We shall see how these backreference are used in R when discussing Regular Expressions in R

Regular Expression in R

R implements two types of regular expressions, "extended regular expression" which is the default and "perl-like" regular expressions. Both of these types use NFA regular expression engine, so they are similar in a number of ways but with subtle differences. Since this is a basic introduction to regular expression, we might not go into the nitty gritty of these differences, but where necessary we shall discuss specific differences.

R's Functions for Regular Expression

R has a number of functions dedicated to pattern searching, these include *`grep()`*, *`grepl()`*, *`regexpr()`*, *`gregexpr()`*, *`sub()`* and *`gsub`*(). In all these functions, default regular expression used is "extended regular expression", but perl-like regular expressions can be turned on with "perl = TRUE". It is also possible to use literal regular expression by using "fixed = TRUE".

grep and grepl

"grep()" and "grepl()" are two commonly used regular expression functions in R. They work the same way but differ in their output. "grep" returns indices of matches or values of matches while grepl returns TRUE for matches and FALSE otherwise.

As an example, let's look at the following text.

```
string1 <- "My students performed exceptionally well this time with the first
student scoring 98%. The second student followed closely with 97%. The third
top student scored 95%."
length(string1)

## [1] 1
```

Let's assume it's a typical after exams report for which we are interested in top three (numerical) scores. We know scores can either be one or two digits followed by a percentage sign, we can therefore use the following regex.

```
pattern <- "\\d{1,2}%"
```

Since our text is one continuous string or a vector of one and contains pattern (it has digits and % after it), calling "grep" with default values will "output 1" or indices of matched

vector, but if "value = TRUE" it will output content of matched vector. For "grepl" output will be TRUE.

```
grep(pattern, string1)

## [1] 1

grep(pattern, string1, value = TRUE)

## [1] "My students performed exceptionally well this time with the first
student scoring 98%. The second student followed closely with 97%. The third
top student scored 95%."

grepl(pattern, string1)

## [1] TRUE
```

If our string was a vector of length three, then search for pattern match will be done for each index.

```
string2 <- c("My students performed exceptionally well this time with the
first student scoring 98%.", "The second student followed closely with 97%.",
"The third top student scored 95%.")
length(string2)

## [1] 3

grep(pattern, string2)

## [1] 1 2 3

grep(pattern, string2, value = TRUE)

## [1] "My students performed exceptionally well this time with the first
student scoring 98%."
## [2] "The second student followed closely with 97%."
## [3] "The third top student scored 95%."

grepl(pattern, string2)

## [1] TRUE TRUE TRUE
```

regexpr and gregexpr

While "grep" and "grepl" check for possible match, "regexpr" and "gregexpr" locate position and length of matches. "regexpr" outputs first match while "gregexpr" output all matches.

```
regexpr(pattern, string1)

## [1] 83
## attr(,"match.length")
## [1] 3
## attr(,"useBytes")
## [1] TRUE
```

```
gregexpr(pattern, string1)

## [[1]]
## [1]  83 129 163
## attr(,"match.length")
## [1] 3 3 3
## attr(,"useBytes")
## [1] TRUE
```

One handy function for extracting matches found is "regmatches".

```
regmatches(string1, regexpr(pattern, string1))

## [1] "98%"

regmatches(string1, gregexpr(pattern, string1))

## [[1]]
## [1] "98%" "97%" "95%"
```

regexec

This function (regexec) is similar to "regexpr" with exception to output produced. "regexpr" outputs a vector while "regexec" outputs a list.

```
regexec(pattern, string1)

## [[1]]
## [1] 83
## attr(,"match.length")
## [1] 3
## attr(,"useBytes")
## [1] TRUE

regmatches(string1, regexec(pattern, string1))

## [[1]]
## [1] "98%"
```

sub and gsub

"sub" and "gsub" are replacement functions, "sub" replaces only the first match while "gsub" makes a global replacement.

```
# Replace first % with percentage
sub("%", " percentage", string1)

## [1] "My students performed exceptionally well this time with the first
student scoring 98 percentage. The second student followed closely with 97%.
The third top student scored 95%."

# Replace all % with percentage
gsub("%", " percentage", string1)
```

```
## [1] "My students performed exceptionally well this time with the first
student scoring 98 percentage. The second student followed closely with 97
percentage. The third top student scored 95 percentage."
```

Backreference can be quite useful for replacement. For example here we will extract an HTML element name from it's opening tag (basically extracting "h1").

```
# HTML opening tag
string3 <- "<h1>"
# Extracting "h1"
sub("<(\\w+)>", "\\1", string3)

## [1] "h1"
```

Under the hood

Up to this point we have basic knowledge of regular expression which can enable us carryout pattern search in R. However, crafting good regular expressions requires two things, first, an understanding of regular expression (how it works) and second, lots of practice (just like every new skill).

Understanding how regular expression works can prevent a lot of unexpected output/results. To this end, we will discuss how regular expression works, with an aim of opening our mind to **think like a regular expression engine**. This will certainly not be a comprehensive discussion but rather build a sound foundation.

The whole idea of "thinking like a regular expression engine" basically means thinking like the programmer who developed regular expression or think of how a regular expression is programmed. This is important in regular expression as our expectations usually differ from how regex is programmed. For instance, when doing literal search, we think of complete words like "cat", "epi", "computer", "sky", "paper" and the like but for regex engine (programme), it sees the individual parts of the word. For the word "cat", regex engine sees it as "c" followed by an "a" and finally "t", this means any series of these individual parts would be considered a match even though they occur in the middle of another word.

There are two core issues we will discuss here, these are implications of quantifiers and how alternations work.

Greedy and Lazy Quantifiers (repetition)

One of the most commonly used regex metacharacters are quantifiers. As you start using these metacharacters, you will soon realise that these metacharacters often do not work to our expectation.

For example, suppose we are interested with the first sentence in "string1" that is, *"My students performed exceptionally well this time with the first student scoring 98%."*. We

might write a regex that matches all the characters from "M" all the way to first space right before score "98" using "." *or ".+" then match 98%. Our regex would therefore be .+%[.]. Here we have opted to use a plus to indicate that at least one character should be matched before the digits although* "" quantifier would have worked equally well. With this regular expression, we get the following output.

```
regmatches(string1, regexpr(pattern2, string1))

## [1] "My students performed exceptionally well this time with the first
student scoring 98%. The second student followed closely with 97%. The third
top student scored 95%."
```

Question, why did we get that output?

Well, let's take another look at our regex and how it was matched to our string. First recall we mentioned R uses a regular expression engine called Non-deterministic Finite Automation (NFA) and that this machine is a **regex-directed engine**. Regex-directed engines begin with a component of a regular expression and match it a given string; for us, we are matching **.+%[.]** (our regex) against "***My students performed exceptionally well this time with the first student scoring 98%. The second student followed closely with 97%. The third top student scored 95%."*** (our string). So beginning with "." in our regex, a match will occur at letter "M", next component is a quantifier "+" whose lower bound requires at least one character to be matched, this was met when we matched "M", so so far so good, we have a match for ".+", but hold on, "+" has an infinite upper boundary and by default it will attempt to match preceding character as may times as possible before it declares a failure. In this case our preceding character is "any character", therefore all characters after "M" are marched (any character includes spaces and punctuation but not new line). If you think about it, ".+" will match everything from "My" to "95%", basically our original string. At this point we are just after ".+" in our regex and at the end of our string (".+" has consumed all characters), question now is, what happens when machine attempts to match "\d" in our regex with our string. You do realise there are no more characters to be matched in our string we are literary at the end of our string. This is the interesting part, the engine (for basic introduction, let's leave distinction of engine and transmission for later discussions) will do what we call **backtracking** which means it will go backwards until it finds "\d" or a digit. So, going backwards we find a full stop, this is not a digit, hence we have to go backwards one more step and find a "5", this is a digit, hence our "\d" from our regex finds a match. From there, our next regex component which is a range quantifier tries to match another digit (it's upper bound) but fails, however, since there was at least one digit match "5", the whole of "\d{1,2}" is a success and we move to the next regex component which is a "%", this finds a match in the string so does the next component which is a literal full stop therefore the engine declares a complete success.

You can now see using ".+" or even "." means regex engine would consume every other character as long as it is not a new line (actually upto the end of a line), and only backtracks to match other components. If what you want to backtrack to have multiple instances like in our example which had multiple digits located at various places in the string, then using "." or ".+" can prove to be difficult.

This behavior of quantifiers matching the most characters (until it's upper bound), is often referred to as greedy matching and since it is the default, all quantifiers they are called **greedy quantifiers**. To make quantifiers not to be greedy, a question mark is added for example **.+?%[.]**. What this means is that a quantifier will only attempt to match its lower bound before attempting to match next regex component. In our example, when we added "?" after "+" meant the engine began by matching only one character before matching a digit, when it did not find it, it matched another character before making a second attempt at matching a digit, this process continue for the next 80 characters until engine reached the first space before score 98%, only after this space did it match a digit. This process of matching one character before attempting to match regex component is referred as **lazy matching**.

```
regmatches(string1, regexpr(pattern2, string1))

## [1] "My students performed exceptionally well this time with the first
student scoring 98%."
```

Of course, we could have been more specific and not used ".+" at all by using regex **(\\w+\\s)+%[.]**. This regex means *"match any multiple words and spaces until a digit is reached followed by a "%" and a fullstop"*. In this case `(\\w+\\s)+` will not consume everything as it will only match words and spaces, this match will eventually fail once it reaches digit "9", but since next regex component is "\\d", "9" will then be matched and quantifier "+" will match "8" before next regex component "%" is matched as well as literal ".". The point here is that, if there is another more specific regex than ".*" or ".+", use it; dot metacharacter is usually very general.

Another point to note here is that a **"*"** never fails. Think about it, "*" means *preceding character can match **zero** or more times,* so a zero match is still a match. Suppose we want to verify all responses from our teachers contain numerical values. We can easily use **"\\d*"** , but this would allow text without numerical values to match as "*" considers a zero match as a success. Using "+" instead of "*" would require at least one digit to be matched for it to be successful. Hence, be extra cautious of ***the unfailing "*"***.

```
string5 <- "My students performed exceptionally well with all leading
students scoring over ninty percent"
grepl(pattern4, string5)

## [1] TRUE

pattern5 <- "\\d+"
grepl(pattern5, string5)

## [1] FALSE
```

It is also worth noting, backtracking consumes a lot of time, and using quantifiers most definitely will result in multiple backtracking. So for efficiency purposes, craft regex that use fewer quantifiers; *if possible*.

Alternations

As mentioned before, alternations allow multiple different patterns to be searched. This can be quite handy when dealing with multiple possible matches however, **order of these alternative do not matter** as **earliest match wins**. What do we mean by this, well, take an example where we want to categorize qualitative epidemiology data, our interest is to extract reported diagnosis. Diagnosis is classified as either "mild", "moderate" or "severe", which form our alternatives for pattern search. If a report has more than one reported diagnosis, then the earliest match from the string becomes the successful match.

```
# Patient report
patient1 <- "Patient was initially presented with mild influenza, but later
was diagnosed with severe influenza."

# Order of alternatives do not matter, as earliest match from string wins
regmatches(patient1, regexpr("(mild|moderate|severe)", patient1))

## [1] "mild"

regmatches(patient1, regexpr("(severe|moderate|mild)", patient1))

## [1] "mild"
```

Let's discuss this a little more, this time we are going to use **"regex transmission and engine analogy"**. We will use "transmission" move us along our regex and "engine" to attempt a match. Our regex is **"mild|moderate|severe"** which is a list of alternatives, with the first alternative being "mild", engine makes it's first attempt right before the first character in the string "***Patient was initially presented with mild influenza, but later was diagnosed with severe influenza.***". Here the first alternative fails and transmission moves attempt to begin right before the second character, here again first alternative fails, it continues failing until it reaches the space before mild or right after the 37th character, here the first alternative matches "mild" in the string. Before this match is made, transmission actually "bumped along" 37 times each time the engine declaring a failure. With first alternative matched, second alternative "moderate" is attempted and this is done right from the beginning just like the first alternative. "moderate" fails to match the string at all positions, the transmission reaches end of string and engine declares a complete failure for this alternative. Next, "severe" is matched at the beginning of the string, this fails and continues to fail until after the eighty second character (right after the space before severe) where if finds a match. Since there are two matches, when using "regexpr" (ideal for reporting one match), first match "mild" becomes overall match as it was the earliest match.

The point here, is that order of alternative have no influence on overall match. So for a case like this epidemiology report, it would be good if diagnosis of interest is written first otherwise it could hamper final output.

References

1. R's help pages

- ?grep
- ?regex

2. Mastering Regular Expressions, by Jeffrey E.F. Friedl (O'Reilly)
3. https://www.rstudio.com/wp-content/uploads/2016/09/RegExCheetsheets.pdf
4. https://www.r-bloggers.com/regular-expressions-exercises-part-1/

Exercises

1. Write a regular expression to delete double words in the following string

```
txt1 <- "There are are so many things you you can do with with regular
expression.
From basic basic find or find-and-replace to complex data data analysis, the
list is endless. It's a a good thing to to learn."
```

2. Using regular expression, capitalise the following sentence (read ?grep, use perl = TRUE)

```
txt2 <- "capitalise me."
```

3. Find indices containing digits only

```
txt3 <- c("99", "All 99", "99a2", " 99", "99 ", "22")
```

4. Write a regex to validate the following solution; expected solution can either be "True" or "False". Allow for spaces at the beginning or at the end of the solution, also allow for upper or lower case.

```
solution <- "That's true"
```

5. Extract the first "span" from this element using "regexpr()"

```
txt4 <- '<span class="center"></span>'
```

Solutions

* There are many ways to achieve a match, hence these regex should not be considered as the only ones to answer question asked.
* Purpose of this exercise is more of learning how to craft successful regex and using ideal R regex function.

1 . Finding and deleting double words

```
gsub("(\\w+)\\s\\1", "\\1", txt1)

## [1] "There are so many things you can do with regular expression. \nFrom
basic find or find-and-replace to complex data analysis, the list is endless.
It's a good thing to learn."
```

2. Capitalising sentences using regular expression

```
txt2

## [1] "capitalise me."

gsub("^(\\w)", "\\U\\1", txt2, perl = TRUE)
```

```
## [1] "Capitalise me."
```

3. Indices containing digits only

```
grep("^\\d+$", txt3)
```

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

4. Validating text

```
grepl("^\\s?(true|false)\\s?$", solution, ignore.case = TRUE)
```

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

5. Extracting matches

```
pattern6 <- "span"
regmatches(txt4, regexpr(pattern6, txt4))
```

```
## [1] "span"
```