A Short Introduction to Regular Expressions

Introduction

- Text data is a very common type of data. We explore regular expression, a powerful engine to search for patterns in text data.
- But before that we briefly explore some of the simpler R functions for dealing with text.

Some standard R functions for text

- In R, a string (that is a sequence of alphanumeric characters) is a variable of type character. And we can also form vectors of those string (called character vectors).
- R has some basic facilities to deal with strings and character vectors. There are functions to extract or replace substrings, to split strings into pieces, and to identify patterns.

Some standard R functions for text

- Consider the text:

```
txtvec = c("This is STAT 406",
    "at the University of Michigan.")
#This is a character vector of length 2

#We can find the length of each
#character string
nchar(txtvec)
[1] 16 30
```

Some standard R functions for text

- We can paste together two character strings using the function "paste":

```
txt=paste(txtvec[1],txtvec[2],sep=" ")
#notice the space between the quote in the argument sep
[1] "This is STAT 406 at the University of Michigan."
```

Some standard R functions for text

- We can extract substring from a given string:

```
substr(txt,start=1,stop=4)
[1] "This"
```

- We can split the string at specified characters.

```
strsplit(txt,split=" ") #split on spaces
[[1]]
[1] "This"       "is"         "STAT"
[4] "406"        "at"         "the"
[7] "University" "of"         "Michigan."
```

Some standard R functions for text

- To can convert lower case to upper case and vice versa:

  ```
  tolower(txt)
  toupper(txt)
  ```

- We can make substitutions. Consider the following example

  ```
  txt=paste(txt,"It is a core course")
  sub("406", "500",txt)
  [1] "This is STAT 500 at the University of Michigan. It
  gsub("is", "was", txt)
  The example shows the difference between
    "sub" and "gsub": "gsub" replaces all matches.
  ```

Regular expressions

- A regular expression is a sequence of characters that defines a pattern; most characters in the pattern simply "match" themselves in a target string.
- To work with regular expressions, we will first need an engine that, given a (vector of) string(s) and a pattern, can search each string, looking for the pattern.
- As we will see a whole language has been developed for doing this with maximum flexibility.

Regular expressions

Example

- Consider the following three sentences.

  ```
  txt=c('This is Stat 406 at the University of
   Michigan', 'Predictive soil mapping is
   important in soil science', 'Mathematics,
   Physics and Biology are three fundamental
     scientific disciplines')
  ```

  Which one contains the word "science"?
- Here "science" is the pattern and each entry of the vector txt is a target string.
- We need a function that can search the target, looking for the pattern.

Regular expressions

- We can achieve this using either of the function "regexpr" or "gregexpr". As follows.

  ```
  pattern="science"
  regexpr(pattern,txt)
  [1] -1 46 -1
  attr(,"match.length")
  [1] -1  7 -1
  attr(,"useBytes")
  [1] TRUE
  ```

- regexpr returns an integer vector with same length as "txt" and giving the starting position of the first match or -1 if there is none.

Regular expressions

- The integer vector returned by regexpr has an attribute called match.length which is also a vector with same length as "txt" and holds the length of the matches.
- If we care about how many times the word 'science' is used in each entry of 'txt', we will use "gregexpr".
- gregexpr(pattern,txt) returns a list with the same length as 'txt'. The *i*-th component of that list gives the occurrences of "pattern" in the *i*-th component of "txt".

Regular expressions

```
pattern="science"
gregexpr(pattern,txt)
[[1]]
[1] -1
attr(,"match.length")
[1] -1
attr(,"useBytes")
[1] TRUE

[[2]]
[1] 46
attr(,"match.length")
[1] 7
attr(,"useBytes")
[1] TRUE

[[3]]
[1] -1
attr(,"match.length")
[1] -1
attr(,"useBytes")
```

Regular expressions

- The above example is limited. For example someone could write "Science" with an upper case "S". Or "Sciences" or scientific.
- We need a tool that has flexibility in searching for patterns. Regular expression provides such a tool. The language is built around meta-characters, which are characters with special meaning. The following are the meta-characters:

  `.   \   |   (   )   [   ] {   ^   $   *   +   ?`

- An example of regular expression will look like this:

  ```
  pattern="\\$[0-9]*(\\.[0-9]+)?"
  ```

Metacharacters

The metacharacters

```
 ^
```

represents the start of a line. For example:

```
pattern="^This"
#looking for start of the line followed by 'This'.
regexpr(pattern,txt)
[1]  1 -1 -1
attr(,"match.length")
[1]  4 -1 -1
attr(,"useBytes")
[1] TRUE
```

Metacharacters

- Sometimes we are searching for a pattern that contains a metacharacter. To prevent the regular expression engine from interpreting the character as a metacharacter, we use the backslash

  \

  before that special characters. In R we double the backslash and write \\

Metacharacters

- In technical terms, we have "escaped" the special meaning of the metacharacters

  ```
  pattern="\\+"
  regexpr(pattern,txt)
  ```

- In the above pattern, "+" is not treated as a metacharacter but as the usual character "+". We will talk more about the metacharater "+" below.

Metacharacters

Similarly, the metacharacters $ represents the end of a line. For example:

```
pattern="soil\\.$"
regexpr(pattern,txt)
[1] -1 -1 -1
attr(,"match.length")
[1] -1 -1 -1
attr(,"useBytes")
[1] TRUE
```

will match all components ending in "soil.". There is none. Notice how we escape the dot ·.

Metacharacters

The symbol

```
[ ]
```

represents a character class. A character class matches a single character out of all the possibilities contained in the brackets. Here is an example

```
pattern="[sS]cience"
regexpr(pattern,txt)
[1] -1 46 -1
```

Metacharacters

There are few rules for character classes. You can specify a range of letters

```
[a-z] or [a-zA-Z] # a short for [abc...z] or [abc..zABC...Z
```

where the order within the class doesn't matter. Consider the following example.

```
pattern="Stat [0-9][0-9][0-9]"
regexpr(pattern,txt)
[1]  9 -1 -1
```

Metacharacters

Remark: You can also combine ranges with literal characters as in

`[a-z0-9_!.?]`

- Notice here that "_", "!", "." and "?" are all regular characters, not meta-characters, because they appear in the bracket "[]".
- Notice also that the character "-" is a meta-character when it appears in brackets, and specifies a range. The only exception is when "-" is at the beginning of the character class as in

  `[-a-z0-9_!.?]`

Metacharacters

Remark: You can also set up a character class by negation. For example

```
[^1-6]
```

will match any character that is not in $\{1, \ldots, 6\}$. For example searching a Stat course that does not start with $1, 2$ or 3.

```
pattern="Stat [^1-3][0-9][0-9]"
regexpr(pattern,txtnyt)
[1]  9 -1 -1
```

But notice that this search will also match something like "Stat x23" or "Stat q00".

Metacharacters

Here is another example. In a list of words search for words with a "q" not followed by "u".

Example

consider the list of words

```
wds=c('Iraq','Iraqi','quest','Iraqian','miqra',
     'Qantas','qasida','qintar','qoph')
pattern="q[^u]"
regexpr(pattern,wds)
[1] -1  4 -1  4  3 -1  1  1  1
```

Metacharacters

- The metacharacter · matches any character. We also call it a wildcard.
- For example, suppose we search the date "04-23-06". It could also be 04/23/06" or even "04.23.06". How to search?

  ```
  pattern = "04.23.06"
  ```

  Here the · is a metacharacter and means "any character".
- Another way for searching is

  ```
  pattern = "04[-/.]23[-/.]06"
  ```

Metacharacters

- In fact the second expression is more accurate. Consider the example

```
txt2=c("04/23/06","19 204523 06789")
pattern1="23.04.06"
regexpr(pattern1,txt2)
[1] 1 5
pattern2="23[-/.]04[-/.]06"
regexpr(pattern2,txt2)
[1]  1 -1
```

Metacharacters

The metacharacter

```
|
```

is used to formulate alternatives. Here is an example.

```
pattern = "soil|Biology"
regexpr(pattern,txt)
[1] -1 12 26
```

This will match expression containing either "soil" or "Biology"

Metacharacters

Parenthesis are typically used to limit the scope of the |
metacharacter. For example

```
pattern = "(s|S)cience" # is same as "science" or "Science"
pattern ="s|Science" # same as "s" or "Science".
```

Metacharacters

The metacharacter ? indicates that the so-flagged expression is optional. Here is an example.

```
pattern="colou?r"  #the "u" is optional
pattern = "George( W)? Bush"  #"W" is optional
```

As another example, suppose that we want to search for "July 4th". One could write "July" or "Jul" or "Jul." and "4th" or "Fourth"

```
pattern="(July|Jul|Jul\\.) (Fourth|4th|4)"
```

Can we do better? Yes. Here is one way.

```
pattern="Jul(\\.|y)? (Fourth|4(th)?)"
```

Metacharacters

- The metacharacters

  `* and +`

  are used to indicate repetitions. The "star" metacharacter means "any number of occurrence, including none" and the"plus" metacharacter means "at least one occurrence".
- Suppose that we want to find expressions with a word put between parenthesis. Remember that "(" and ")" are meta-characters.

  ```
  pattern = "\\(.*\\)"
  ```

  This will match anything written in parenthesis.

Metacharacters

- The last metacharacter we study is { }. We use this to specify the minimum and maximum number of matches of an expression.
- Here is an example. Suppose we want to find expressions having a word within parenthesis of length between 2 and 6.

```
pattern = "\\(.{2,6}\\)"
txt4="This (course) is fun"
regexpr(pattern,txt4)
```

Metacharacters

Example

Write a regular expression to match a dollar amount with optional cents.

```
pattern="\\$[0-9]+(\\.[0-9]+)?"
txt=c('my number is 123', 'my number is $123',
'my number is $12.45', 'my number is $1.125')
regexpr(pattern,txt)
 [1] -1 14 14 14
```

Metacharacters

Example

Write a regular expression to match an email address.

```
pattern="[a-zA-Z0-9_]+@[a-zA-Z0-9_]+\\.[a-z]{3}"
txt=c('smith@umich.edu','my number is 123',
   'smith_12@yahoo.com', 'smith@ong.fr')
regexpr(pattern,txt)
[1]  1 -1  1  1
```

An Example

- We consider a more interesting example. How much of the US Federal research funding go to various topics in Statistics.
- The data is the same we used in Chapter 6. It is in XML format obtained from

  `http://www.nsf.gov/awardsearch/download.jsp`
- Each file reports on one award. And has the following structure.

```
xmlChildren(node1)
$AwardTitle
<AwardTitle>Kent State Informal Analysis Seminar</Awarc

$AwardEffectiveDate
<AwardEffectiveDate>01/01/2014</AwardEffectiveDate>
...
```

An Example

- Suppose we wish to know how many projects dealing with Markov chain Monte Carlo, or Bayesian statsitics, or High-dimensional (big data) problems were funded over the years.
- I downloaded the files from 1990 to 2013, and extracted all the files in separate folders.
- We could choose to look for the following simple patterns

```
pattern1="(MCMC|[Mm]arkov [Cc]hain [Mm]onte [Cc]arlo
    |MARKOV CHAIN MONTE CARLO)"
pattern2="[bB]ayesian"
pattern3="([bB]ig [dD]ata|[Dd]ata [Ss]cience
    |[Hh]igh [Dd]imensional)"
```

An Example

```
years=seq(from=1990,to=2013,by=1)
nb_years=length(years);
count_proj1=0;count_proj2=0;count_proj3=0;
funding1=0; funding2=0;funding3=0;
```

An Example

```
for (i in 1:nb_years){  #for each year
   Files=list.files(paste("./", years[i], "/",sep=""))
   L=length(Files)
   count_local1=0;count_local2=0;count_local3=0;
   funding_local1=0;funding_local2=0;funding_local3=0;
   for (l in 1:L){ #For each project
      filename=paste("./", years[i], "/", Files[l],sep="")
      doc=xmlTreeParse(filename)
      root=xmlRoot(doc)
      lst_chld=xmlChildren(root[[1]])
      V1=xmlValue(lst_chld$AbstractNarration)
      V2=as.numeric(xmlValue(lst_chld$AwardAmount))
```

An Example

```
if (length(V1)>0 & length(V2)>0){
   search1=regexpr(pattern1,V1)
   if (search1!=-1){
      count_local1=count_local1+1;
      funding_local1=funding_local1+V2;
   }
   search2=regexpr(pattern2,V1)
   if (search2!=-1){
      count_local2=count_local2+1;
      funding_local2=funding_local2+V2;
   }
```

An Example

```
            search3=regexpr(pattern3,V1)
            if (search3!=-1){
                count_local3=count_local3+1;
                funding_local3=funding_local3+V2;
            }
        }
    }
    count_proj1[i]=count_local1;count_proj2[i]=count_local2;
    count_proj3[i]=count_local3;
    funding1[i]=funding_local1;funding2[i]=funding_local2;
    funding3[i]=funding_local3;
    print(years[i])
}
```

An Example

Do some plots

```
par(mfrow=c(1,2))
plot(years,count_proj1,type='l',lty=1,col='blue',
   ylim=c(0,170),main='Number of projects
      funded by NSF with key words')
par(new=T)
plot(years,count_proj2,type='l',lty=2,col='black',
        ylim=c(0,170))
par(new=T)
plot(years,count_proj3,type='l',lty=3,col='red',
     ylim=c(0,170))
legend(x='topleft',legend=c('MCMC','Bayesian',
'High-dimensional/Big data/Data Science'),
     col=c('blue','black','red'),lty=c(1,2,3))
```

An Example

```
f1=funding1/10^6;f2=funding2/10^6;f3=funding3/10^6;
plot(years,f1,type='l',lty=1,col='blue',ylim=c(0,65),
    main='NSF Dollar Amount (in mil.)
         to projects with key words')
par(new=T)
plot(years,f2,type='l',lty=2,col='black',ylim=c(0,65))
par(new=T)
plot(years,f3,type='l',lty=3,col='red',ylim=c(0,65))
legend(x='topleft',legend=c('MCMC','Bayesian',
'High-dimensional/Big data/Data Science'),
    col=c('blue','black','red'),lty=c(1,2,3))
```

An Example

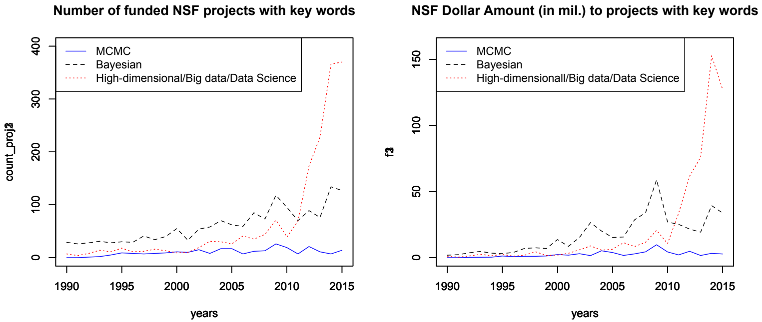

Figure 1: NSF Research Funding

Conclusion

Conclusion:

- We have seen a brief introduction to regular expressions.
- A regular expression engine is a software embedded in many languages and operating systems that offers the capability to search for patterns in text documents.
- The patterns are constructed using a small regular expression language based on meta-characters.
- These notes were prepared using the book "Mastering regular expressions" 3rd Edition by Jeffrey Friedl which covers the topic in great detail.