Quantitative Content Analysis: Lecture 9

Matthias Haber

12 April 2017

Today's outline

- Repetition Quantitative text analysis concepts
- Regular Expressions
- Dictionary Approaches
    - Deriving a dictionary, "Wordscores version 0.1"
    - External dictionaries
    - Application
- Text analysis workflow
- Creating a text corpus
- Preprocessing
- Regular Expressions

Set up to work along today's slides

Packages

```
# if you don't have the package, install it first
# install.package()
library(stringr)
library(dplyr)
```

Data

```
gDat <- read.csv("https://raw.githubusercontent.com/plotly/dat
```

Fake Data

Regular expressions

Regular Expressions (regex) are a language or syntax to search in texts. Regex are used by most search engines in one form or another and are part of almost any programming language. It is truly the heart and soul for string operations. In R, many string functions in `base` R as well as in `stringr` package use regular expressions, even Rstudio's search and replace allows regular expression.

You could use regex to e.g.:

- Count the occurence of certain persons/organization etc. in text
- Calculate the sums of fund discussed in legislation
- Chose your texts based on regexes

In textpreparation, regex are used to remove certain unwanted parts of text.

Regular expression syntax

Regular expressions typically specify characters to seek out, possibly with information about repeats and location within the string. This is accomplished with the help of metacharacters that have specific meaning: `$ * + . ? [ ] ^ { } | ( ) \`.

String functions related to regular expression

Regular expression is a pattern that describes a specific set of strings with a common structure. It is heavily used for string matching / replacing in all programming languages, although specific syntax may differ a bit. It is truly the heart and soul for string operations. In R, many string functions in base R as well as in `stringr` package use regular expressions, even Rstudio's search and replace allows regular expression, we will go into more details about these functions later this week:

- identify match to a pattern: `grep(..., value = FALSE)`, `grepl()`, `stringr::str_detect()`
- extract match to a pattern: `grep(..., value = TRUE)`, `stringr::str_extract()`, `stringr::str_extract_all()`
- locate pattern within a string, i.e. give the start position of matched patterns. `regexpr()`, `gregexpr()`, `stringr::str_locate()`, `string::str_locate_all()`
- replace a pattern: `sub()`, `gsub()`, `stringr::str_replace()`, `stringr::str_replace_all()`

Escape sequences

There are some special characters in R that cannot be directly coded in a string. For example, let's say you specify your pattern with single quotes and you want to find countries with the single quote `'`. You would have to "escape" the single quote in the pattern, by preceding it with \, so it's clear it is not part of the string-specifying machinery:

```
grep('\'', levels(gDat$country), value = TRUE)
## [1] "Cote d'Ivoire"
```

Escape sequences (II)

There are other characters in R that require escaping, and this rule applies to all string functions in R, including regular expressions. See here for a complete list of R esacpe sequences.

- `\'`: single quote. You don't need to escape single quote inside a double-quoted string, so we can also use `"'"` in the previous example.
- `\"`: double quote. Similarly, double quotes can be used inside a single-quoted string, i.e. `'"'`.
- `\n`: newline.
- `\r`: carriage return.
- `\t`: tab character.

Note: `cat()` and `print()` to handle escape sequences differently, if you want to print a string out with these sequences interpreted, use `cat()`.

```
print("a\nb")
```

Task 1

Find all countries with ee in the Gapminder dataset using quantifiers.

Task 1 solution

Find all countries with ee in the Gapminder dataset using quantifiers.

```
## [1] "Greece"
```

Position of pattern within the string

- ^: matches the start of the string.
- $: matches the end of the string.
- \b: matches the empty string at either edge of a *word*. Don't confuse it with ^ $ which marks the edge of a *string*.
- \B: matches the empty string provided it is not at an edge of a word.

```
(strings <- c("abcd", "cdab", "cabd", "c abd"))
## [1] "abcd"  "cdab"  "cabd"  "c abd"
grep("ab", strings, value = TRUE)
## [1] "abcd"  "cdab"  "cabd"  "c abd"
grep("^ab", strings, value = TRUE)
```

Task 2

Find all `.txt` files in the “Slides/Week9” folder

Task 2 solution

Find all `.txt` files in the "Slides/Week9" folder

```
## character(0)
```

Operators

- `.`: matches any single character
- `[...]`: a character list, matches any one of the characters inside the square brackets. We can also use `-` inside the brackets to specify a range of characters.
- `[^...]`: an inverted character list, similar to `[...]`, but matches any characters **except** those inside the square brackets.
- `\`: suppress the special meaning of metacharacters in regular expression, i.e. `$ * + . ? [ ] ^ { } | ( ) \`, similar to its usage in escape sequences. Since `\` itself needs to be escaped in R, we need to escape these metacharacters with double backslash like `\\$`.
- `|`: an "or" operator, matches patterns on either side of the `|`.

Task 3

Find countries in the Gapminder data with letter `i` or `t`, and ends with `land`, and replace `land` with `LAND` using backreference.

Task 3 solution

Find countries in the Gapminder data with letter `i` or `t`, and ends with `land`, and replace `land` with `LAND` using backreference.

```
## [1] "FinLAND"     "IceLAND"     "IreLAND"     "SwaziLAND"
## [6] "ThaiLAND"
```

Character classes

Character classes allow to specify entire classes of characters, such as numbers, letters, etc. There are two flavors of character classes, one uses `[:` and `:]` around a predefined name inside square brackets and the other uses \ and a special character. They are sometimes interchangeable.

- `[:digit:]` or `\d`: digits, 0 1 2 3 4 5 6 7 8 9, equivalent to `[0-9]`.
- `\D`: non-digits, equivalent to `[^0-9]`.
- `[:lower:]`: lower-case letters, equivalent to `[a-z]`.
- `[:upper:]`: upper-case letters, equivalent to `[A-Z]`.
- `[:alpha:]`: alphabetic characters, equivalent to `[[:lower:][:upper:]]` or `[A-z]`.
- `[:alnum:]`: alphanumeric characters, equivalent to `[[:alpha:][:digit:]]` or `[A-z0-9]`.
- `\w`: word characters, equivalent to `[[:alnum:]_]` or `[A-z0-9_]`.
- `\W`: not word, equivalent to `[^A-z0-9_]`.
- `[:xdigit:]`: hexadecimal digits (base 16), 0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f, equivalent to `[0-9A-Fa-f]`

General modes for patterns

There are different syntax standards for regular expressions, and R offers two:

- POSIX extended regular expressions (default)
- Perl-like regular expressions.

You can easily switch between by specifying `perl = FALSE/TRUE` in base R functions, such as `grep()` and `sub()`. For functions in the `stringr` package, wrap the pattern with `perl()`. The syntax between these two standards are a bit different sometimes, see an example here. If you had previous experience with Python or Java, you are probably more familiar with the Perl-like mode. But for this tutorial, we will only use R's default POSIX standard.

General modes for patterns (II)

There's one last type of regular expression – "fixed", meaning that the pattern should be taken literally. Specify this via `fixed = TRUE` (base R functions) or wrapping with `fixed()` (`stringr` functions). For example, `"A.b"` as a regular expression will match a string with "A" followed by any single character followed by "b", but as a fixed pattern, it will only match a literal "A.b".

```r
(strings <- c("Axbc", "A.bc"))
## [1] "Axbc" "A.bc"
pattern <- "A.b"
grep(pattern, strings, value = TRUE)
## [1] "Axbc" "A.bc"
grep(pattern, strings, value = TRUE, fixed = TRUE)
## [1] "A.bc"
```

By default, pattern matching is case sensitive in R, but you can turn it off with `ignore.case = TRUE` (base R functions) or wrapping with

Task 4

Find continents in Gapminder with letter o in it.

Task 4 solution

Find continents in Gapminder with letter o in it.

```
## [1] "Europe"  "Oceania"
```

Regular expression vs shell globbing

The term globbing in shell or Unix-like environment refers to pattern matching based on wildcard characters. A wildcard character can be used to substitute for any other character or characters in a string. Globbing is commonly used for matching file names or paths, and has a much simpler syntax. It is somewhat similar to regular expressions, and that's why people are often confused between them. Here is a list of globbing syntax and their comparisons to regular expression:

- `*`: matches any number of unknown characters, same as `.*` in regular expression.
- `?`: matches one unknown character, same as `.` in regular expression.
- `\`: same as regular expression.
- `[...]`: same as regular expression.
- `[!...]`: same as `[^...]` in regular expression.

Resources

- Regular expression in R official document.
- Perl-like regular expression: regular expression in perl manual.
- `qdapRegex` package: a collection of handy regular expression tools, including handling abbreviations, dates, email addresses, hash tags, phone numbers, times, emoticons, and URL etc.
- Recently, there are some attemps to create human readable regular expression packages, Regularity in Ruby is a very successful one. Unfortunately, its implementation in R is still quite beta at this stage, not as friendly as Regularity yet. But keep an eye out, better packages may become available in the near future!
- There are some online tools to help learn, build and test regular expressions. On these websites, you can simply paste your test data and write regular expression, and matches will be highlighted.
- regexpal
- RegExr

Dictionary approaches

Dictionaries help classifying texts to categories or determine their content of a known concept.

- Which text pertain to which categories?
- Which texts contain how much of a concept?
- Compared to e.g. CMP
    - Dictionaries require knowing the semantic form of the concept
    - i.e. one would need a complete dictionary of left or right statements

Rational for dictionaries

- Rather than count words that occur, pre-define words associated with specific meanings
- Two components:
    - **key** the label for the equivalence class for the concept or canonical term
    - **values** (multiple) terms or patterns that are declared equivalent occurences of the key class
- Frequently involves lemmatization: transformation of all in ected word forms to their "dictionary look-up form" – more powerful than stemming

Creating dictionaries

Creating Dictionaries

- Scheme of classification
- Documents with known properties or classification
    - Training Set: Used to construct a dictionary
    - Test Set: Used to test dictionary (properties/classification is known)
    - Classification Set: Text to be classified/scaled with the dictionary

Creating dictionaries (II)

Sequence of steps

- Collect the words that discriminate between categories/concepts, i.e. create a dictionary
    - Existing dictionaries
    - Creating a dictionary
- Quantify the occurence of these words in texts
- Validate

Creating dictionaries (III)

Methods (though not exhaustive)

- By hand
  - Based on a Training Set (Laver & Garry)
  - Based on a previously existing list or external Sources (Dodds & Danforth)
- Automatically (Wordscores)
  - Replaces the creation of a dictionary as in Laver and Garry 2000

Estimating Policy Positions from Political Texts

Laver & Garry

- Goal: Generating party positions for British and Irish manifestos
- Coding scheme similar to the CMP's
    - More hierachical, larger number of categories
    - Each category has a pro-, con- and neutral variant

Estimating Policy Positions from Political Texts (II)

Training Set

- Manifestos of Labour and Cons (UK) in 1992
  - Pool of ‘keywords’
  - $N_L \geq 2N_R =>$ Dictionary element left
  - $N_R \geq 2N_L =>$ Dictionary element right
- Allocate selected words to the coding scheme’s categories

Estimating Policy Positions from Political Texts (III)

- Count the occurence of the elements in the dictionary in manifestos
    - Britain (1992 & 1997)
    - Ireland (1992 & 1997)
- Left-right-scaling: $\frac{R-L}{R+L}$ (see Session 7 and assignment 2)
    - "Updating process"
    - $Econ_{LR}$
    - Soc_{LR}

Estimating Policy Positions from Political Texts (III)

- Test-Set: Crossvalidation
  - Expert Surveys
  - CMP Coding/Revised CMP Coding

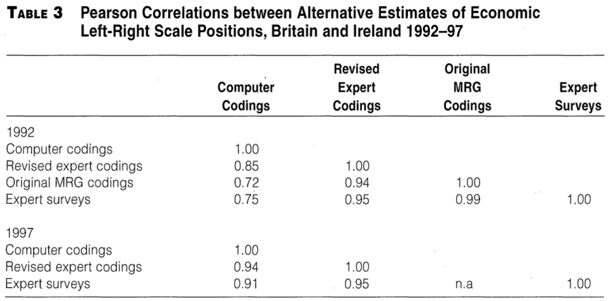

TABLE 3 **Pearson Correlations between Alternative Estimates of Economic Left-Right Scale Positions, Britain and Ireland 1992–97**

	Computer Codings	Revised Expert Codings	Original MRG Codings	Expert Surveys
1992				
Computer codings	1.00			
Revised expert codings	0.85	1.00		
Original MRG codings	0.72	0.94	1.00	
Expert surveys	0.75	0.95	0.99	1.00
1997				
Computer codings	1.00			
Revised expert codings	0.94	1.00		
Expert surveys	0.91	0.95	n.a	1.00

Next Session

- Regular expressions
- Dictionary-based content analysis