Quantitative Content Analysis: Lecture 9

Matthias Haber

12 April 2017

Today's outline

- Recap Quanteda
- Regular Expressions
- Dictionary Approaches

Set up to work along today's slides

Working directory

```r
wdir <- getwd()
```

Packages

```r
# if you don't have the package, install it first
# install.package()
library(stringr)
library(dplyr)
library(quanteda) # needs devtools & Matrix package
library(tm)
# you have to install the next package from github
# if you have rtools and devtools installed
# use devtools::install_github("kbenoit/readtext")
library(readtext)
```

Set up to work along today's slides (II)

Data

```r
# Gapminder dataset
gDat <- read.csv(paste0("https://raw.githubusercontent.com/",
                        "plotly/datasets/master/",
                        "gapminderDataFiveYear.csv"))
# UK manifesto texts on immigration
temp <- tempfile(fileext = ".zip")
download.file(paste0("https://github.com/kbenoit/ME414/",
                     "raw/master/day8/UKimmigTexts.zip"),
              temp)
unzip(zipfile= temp, exdir = wdir)
unlink(temp)
```

Exploring quanteda functions

Look at the Quick Start vignette, and browse the manual for quanteda. You can use `example()` function for any function in the package, to run the examples and see how the function works.

Making a corpus from a vector

`inaugTexts` is the UTF-8 encoded set of 57 presidential inaugural addresses.Try using `corpus()` on this set of texts to create a corpus. Once you have constructed this corpus, use the `summary()` method to see a brief description of the corpus. The names of the character vector `inaugTexts` should have become the document names.

```r
data_corpus_inaugural <- corpus(inaugTexts)
summary(data_corpus_inaugural)
```

Making a corpus a directory of text files

The `readtext()` function can read (almost) any set of files into an object that you can then call the `corpus()` function on, to create a corpus. For example, to read the UK manifesto files into a corpus use:

```r
# if you have readtext installed
mycorpus <- corpus(readtext(paste0(wdir, "/*.txt")))

# alternatively use VCorpus from tm
myTmCorpus <- VCorpus(DirSource(wdir, pattern = "\\.txt" ))
mycorpusTM <- corpus(myTmCorpus)
```

Explore some phrases in the text

The kwic (for “key-words-in-context”) function is easily usable and configurable to explore texts in a descriptive way.

```
kwic(data_corpus_inaugural, "terror", 3)
##
## [1797-Adams, 1327]                violence, by | terror |
## [1933-Roosevelt, 112] unreasoning, unjustified | terror |
## [1941-Roosevelt, 289]         by a fatalistic | terror |
## [1961-Kennedy, 868]       uncertain balance of | terror |
## [1981-Reagan, 821]        Americans from the | terror |
## [1997-Clinton, 1055]        the fanaticism of | terror |
## [1997-Clinton, 1655]   strong defense against | terror |
## [2009-Obama, 1646]          aims by inducing | terror |
##
## [1797-Adams, 1327]    , intrigue,
## [1933-Roosevelt, 112] which paralyzes needed
## [1941-Roosevelt, 289] , we proved
## [1961-Kennedy, 868]   that stays the
## [1981-Reagan, 821]    of runaway living
## [1997-Clinton, 1055]  . And they
```

Create a document-feature matrix

Create a document-feature matrix, using `dfm`

```
mydfm <- dfm(data_corpus_inaugural,
             remove = stopwords("english"))

topfeatures(mydfm, 10)
##          ,          .          -       will government     people
##       7026       4945       1042        911        594        575
##          ;         us        can       upon
##        565        478        471        371
```

Experiment with different `dfm` options, such as `stem=TRUE`. The function `trim()` allows to reduce the size of the dfm following its construction.

Grouping on a variable

If you want to aggregate all speeches by presidential name, you can execute

```
mydfm <- dfm(data_corpus_inaugural, groups = "President")
docnames(mydfm)[1:10]
##  [1] "Adams"      "Buchanan"   "Bush"       "Carter"     "Cleveland"
##  [6] "Clinton"    "Coolidge"   "Eisenhower" "Garfield"   "Grant"
```

Note: that this groups Theodore and Franklin D. Roosevelt together – to separate them we would have needed to add a firstname variable using `docvars()` and grouped on that as well.

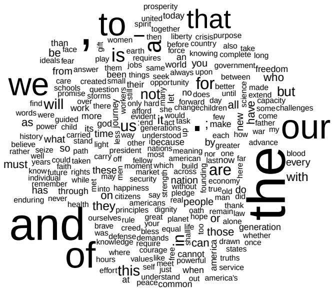
Subset a corpus

There is a corpus_subset method defined for a corpus, which works just like R's normal subset() command. For instance if you want a wordcloud of just Obama's two inagural addresses, you would need to subset the corpus first:

```
obamadfm <- dfm(corpus_subset(data_corpus_inaugural,
                              President=="Obama"))
```

Subset a corpus (II)

```r
plot(obamadfm)
```



Task 1

Create a wordcloud of just Obama's two inagural addresses with the stopwords.

Task 1 solution

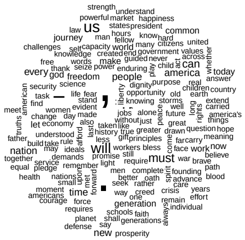
Create a wordcloud of just Obama's two inagural addresses with the stopwords.

```r
obamadfm <- dfm(corpus_subset(data_corpus_inaugural,
                              President=="Obama"),
                remove = stopwords("english"))
```

Task 1 solution (II)

Create a wordcloud of just Obama's two inagural addresses with the stopwords.

```r
plot(obamadfm)
```



Descriptive statistics

Quanteda provides a number of functions to create descriptive summaries. For example, you can examine the most frequent word features using `topfeatures()` and use `nsyllable` to count the number of syllables.

```
ibdfm <- dfm(data_corpus_irishbudget2010)
topfeatures(ibdfm)
##  the    .   to    ,   of  and   in    a   is that
## 3600 2371 1639 1548 1537 1360 1233 1013  868  804
nsyllable(texts(data_corpus_irishbudget2010))
##  [1] 13370  6313  9126 10569 10260  6172  3155  6081  1869  1842  1517
## [12]  2293  1793  5924
```

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. 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

- `grep(..., value = FALSE)`, `grepl()`, `stringr::str_detect()` to identify match to a pattern
- `grep(..., value = TRUE)`, `stringr::str_extract()`, `stringr::str_extract_all()` to extract match to a pattern
- `regexpr()`, `gregexpr()`, `stringr::str_locate()`, `string::str_locate_all()` to locate pattern within a string
- `sub()`, `gsub()`, `stringr::str_replace()`, `stringr::str_replace_all()` to replace a pattern
- `strsplit()`, `stringr::str_split()` to split a string using a pattern

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.

Quantifiers

Quantifiers specify the number of repetitions of the pattern.

- `*`: matches at least 0 times.
- `+`: matches at least 1 times.
- `?`: matches at most 1 times.
- `{n}`: matches exactly n times.
- `{n,}`: matches at least n times.
- `{n,m}`: matches between n and m times.

Quantifiers (II)

```
strings <- c("a", "ab", "acb", "accb", "acccb", "accccb")
grep("ac*b", strings, value = TRUE)
## [1] "ab"     "acb"    "accb"   "acccb"  "accccb"
grep("ac+b", strings, value = TRUE)
## [1] "acb"    "accb"   "acccb"  "accccb"
grep("ac?b", strings, value = TRUE)
## [1] "ab"  "acb"
grep("ac{2}b", strings, value = TRUE)
## [1] "accb"
grep("ac{2,}b", strings, value = TRUE)
## [1] "accb"   "acccb"  "accccb"
grep("ac{2,3}b", strings, value = TRUE)
## [1] "accb"  "acccb"
```

Task 2

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

Task 2 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.

```r
(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)
## [1] "abcd"
grep("ab$", strings, value = TRUE)
## [1] "cdab"
grep("\\bab", strings, value = TRUE)
## [1] "abcd"  "c abd"
```

Task 3

Find all `.txt` files in your working directory

Task 3 solution

Find all `.txt` files in your working directory

```
files <- list.files(wdir)
grep("\\.txt", files, value = TRUE)
## [1] "BNP.txt"            "Coalition.txt"     "Conservative.txt"
## [4] "Greens.txt"         "Labour.txt"        "LibDem.txt"
## [7] "PC.txt"             "SNP.txt"           "UKIP.txt"
```

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 `|`.
- `(...)`: grouping in regular expressions which allows to retrieve the bits that matched various parts of your regular expression. Each group can than be refer using `\\N`, with N being the No. of `(...)` used. This is called **backreference**.

Operators (II)

```
strings <- c("^ab", "ab", "abc", "abd", "abe", "ab 12")
grep("ab.", strings, value = TRUE)
## [1] "abc"   "abd"   "abe"   "ab 12"
grep("ab[c-e]", strings, value = TRUE)
## [1] "abc" "abd" "abe"
grep("ab[^c]", strings, value = TRUE)
## [1] "abd"   "abe"   "ab 12"
grep("^ab", strings, value = TRUE)
## [1] "ab"    "abc"   "abd"   "abe"   "ab 12"
grep("\\^ab", strings, value = TRUE)
## [1] "^ab"
grep("abc|abd", strings, value = TRUE)
## [1] "abc" "abd"
gsub("(ab) 12", "\\1 34", strings)
## [1] "^ab"   "ab"    "abc"   "abd"   "abe"   "ab 34"
```

Task 4

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

Task 4 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"   "SwitzerLAND
## [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]`.

Character classes (II)

- `\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]`.
- `[:blank:]`: blank characters, i.e. space and tab.
- `[:space:]`: space characters: tab, newline, vertical tab, form feed, carriage return, space.
- `\s`: space, ‘ ‘.
- `\S`: not space.
- `[:punct:]`: punctuation characters, ! " # $ % & ’ ( ) - + , - . / : ; < = > ? @ [ ] ^ _ ‘ { | } ~.

Character classes (III)

- `[:graph:]`: graphical (human readable) characters: equivalent to `[[:alnum:][:punct:]]`.
- `[:print:]`: printable characters, equivalent to `[[:alnum:][:punct:]\\s]`.
- `[:cntrl:]`: control characters, like `\n` or `\r`, `[\x00-\x1F\x7F]`.

Note:

- `[:...:]` has to be used inside square brackets, e.g. `[[:digit:]]`.
- \ itself is a special character that needs escape, e.g. `\\d`. Do not confuse these regular expressions with R escape sequences such as `\t`.

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()`.

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".

```
strings <- c("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"
```

General modes for patterns (III)

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 `ignore.case()` (`stringr` functions). Alternatively, you can use `tolower()` and `toupper()` functions to convert everything to lower or upper case. Take the same example above:

```
pattern <- "a.b"
grep(pattern, strings, value = TRUE)
## character(0)
grep(pattern, strings, value = TRUE, ignore.case = TRUE)
## [1] "Axbc" "A.bc"
```

Task 5

Find continents in Gapminder with letter o in it.

Task 5 solution

Find continents in Gapminder with letter o in it.

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

Regular expression vs shell globbing

The term globbing 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. Below 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.
- 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. They are a hybrid procedure between qualitative and quantitative classification. Dictionary construction involves a lot of contextual interpretation and qualitative judgment.

- 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
- Perfect reliability because there is no human decision making as part of the text analysis procedure

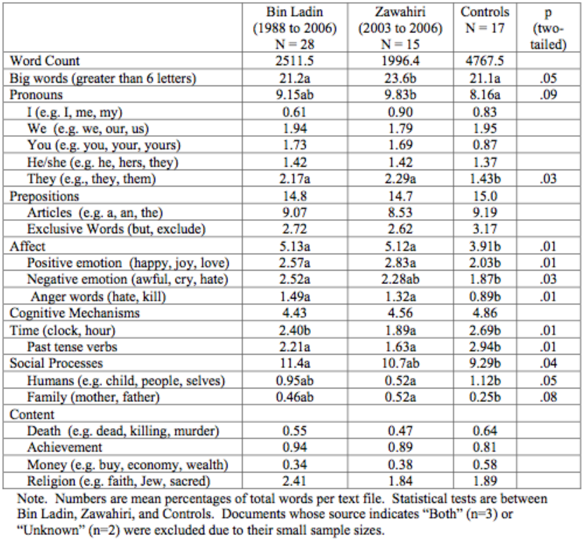
Rational for dictionaries

- 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

Example 1: Linquistic inquiry

- Craeted by Pennebaker et al: `http://www.liwc.net`
- uses a dictionary to calculate the percentage of words in the text that match each of up to 82 language dimensions
- Consists of about 4,500 words and word stems, each defining one or more word categories or subdictionaries
- For example, the word *cried* is part of five word categories: sadness, negative emotion, overall affect, verb, and past tense verb
- Hierarchical: so "anger" are part of an emotion category and a negative emotion subcategory
- You can buy it here: `http://www.liwc.net/descriptiontable1.php`

Example 2: Terrorist speech

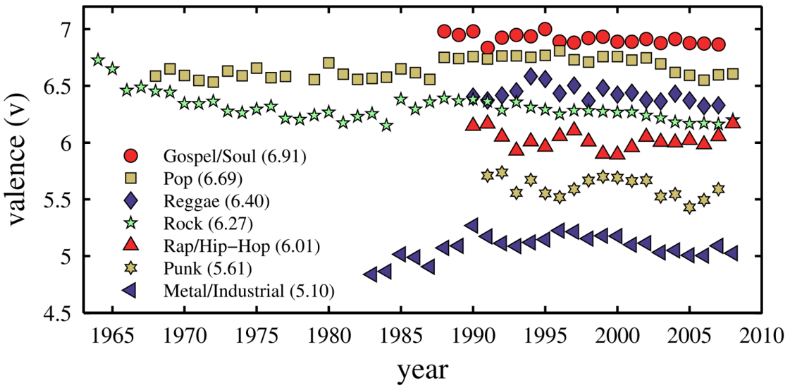


	Bin Ladin (1988 to 2006) N = 28	Zawahiri (2003 to 2006) N = 15	Controls N = 17	p (two-tailed)
Word Count	2511.5	1996.4	4767.5	
Big words (greater than 6 letters)	21.2a	23.6b	21.1a	.05
Pronouns	9.15ab	9.83b	8.16a	.09
I (e.g. I, me, my)	0.61	0.90	0.83	
We (e.g. we, our, us)	1.94	1.79	1.95	
You (e.g. you, your, yours)	1.73	1.69	0.87	
He/she (e.g. he, hers, they)	1.42	1.42	1.37	
They (e.g., they, them)	2.17a	2.29a	1.43b	.03
Prepositions	14.8	14.7	15.0	
Articles (e.g. a, an, the)	9.07	8.53	9.19	
Exclusive Words (but, exclude)	2.72	2.62	3.17	
Affect	5.13a	5.12a	3.91b	.01
Positive emotion (happy, joy, love)	2.57a	2.83a	2.03b	.01
Negative emotion (awful, cry, hate)	2.52a	2.28ab	1.87b	.03
Anger words (hate, kill)	1.49a	1.32a	0.89b	.01
Cognitive Mechanisms	4.43	4.56	4.86	
Time (clock, hour)	2.40b	1.89a	2.69b	.01
Past tense verbs	2.21a	1.63a	2.94b	.01
Social Processes	11.4a	10.7ab	9.29b	.04
Humans (e.g. child, people, selves)	0.95ab	0.52a	1.12b	.05
Family (mother, father)	0.46ab	0.52a	0.25b	.08
Content				
Death (e.g. dead, killing, murder)	0.55	0.47	0.64	
Achievement	0.94	0.89	0.81	
Money (e.g. buy, economy, wealth)	0.34	0.38	0.58	
Religion (e.g. faith, Jew, sacred)	2.41	1.84	1.89	

Note. Numbers are mean percentages of total words per text file. Statistical tests are between Bin Ladin, Zawahiri, and Controls. Documents whose source indicates "Both" (n=3) or "Unknown" (n=2) were excluded due to their small sample sizes.

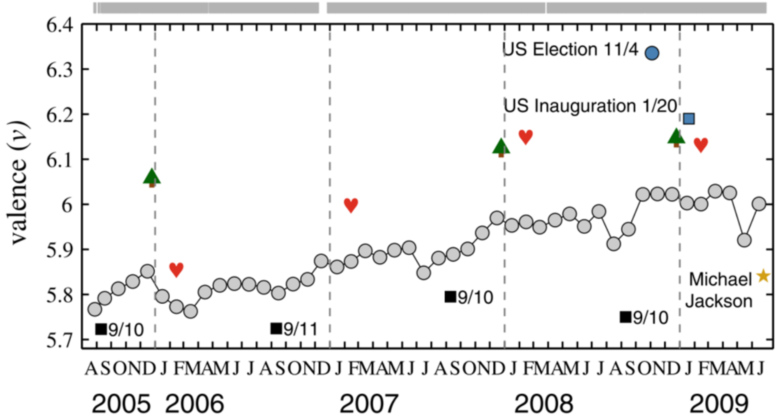
Examples 3: Happiness in song lyrics

Valence time series for song titles broken down by representative genres (Dodds & Danforth 2009)

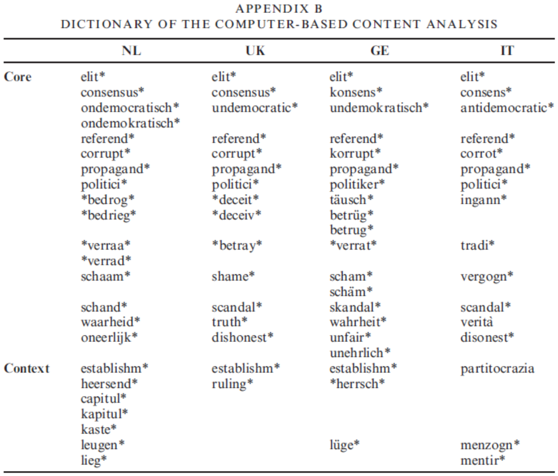


Examples 4: Happiness in blogs

Time series of average monthly valence for blog sentences starting with "I feel. . . " (Dodds & Danforth 2009)



Advantage: Multi-lingual



APPENDIX B
DICTIONARY OF THE COMPUTER-BASED CONTENT ANALYSIS

	NL	UK	GE	IT
Core	elit*	elit*	elit*	elit*
	consensus*	consensus*	konsens*	consens*
	ondemocratisch*	undemocratic*	undemokratisch*	antidemocratic*
	ondemokratisch*			
	referend*	referend*	referend*	referend*
	corrupt*	corrupt*	korrupt*	corrot*
	propagand*	propagand*	propagand*	propagand*
	politici*	politici*	politiker*	politici*
	bedrog	*deceit*	täusch*	ingann*
	bedrieg	*deceiv*	betrüg*	
			betrug*	
	verraa	*betray*	*verrat*	tradi*
	verrad			
	schaam*	shame*	scham*	vergogn*
			schäm*	
	schand*	scandal*	skandal*	scandal*
	waarheid*	truth*	wahrheit*	verità
	oneerlijk*	dishonest*	unfair*	disonest*
			unehrlich*	
Context	establishm*	establishm*	establishm*	partitocrazia
	heersend*	ruling*	*herrsch*	
	capitul*			
	kapitul*			
	kaste*			
	leugen*		lüge*	menzogn*
	lieg*			mentir*

(from Rooduijn and Pauwels 2011)

Disdvantage: Highly specific to context

- Example: Loughran and McDonald used the Harvard-IV-4 TagNeg (H4N) dictionary to classify sentiment for a corpus of 50,115 firm-year 10-K filings from 1994-2008
- they found that almost three-fourths of the “negative” words of H4N were typically not negative in a financial context e.g. mine or cancer, or tax, cost, capital, board, liability, foreign, and vice
- Problem: **polysemes** words that have multiple meanings
- Another problem: dictionary lacked important negative financial words, such as felony, litigation, restated, misstatement, and unanticipated

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

Creating a simple dictionary

To create a simple dictionary of parts of speech, for instance we could define a dictionary consisting of articles and conjunctions, using:

```r
posDict <- dictionary(list(articles = c("the", "a", "and"),
           conjunctions = c("and", "but", "or", "nor", "for", "yet", "so"))
```

We can use this dictionary when we create a dfm to let this define a set of features:

```r
posDfm <- dfm(data_corpus_inaugural, dictionary=posDict)
posDfm[1:5,]
## Document-feature matrix of: 5 documents, 2 features (0% sparse).
## 5 x 2 sparse Matrix of class "dfmSparse"
##                  features
## docs               articles conjunctions
##   1789-Washington      178           73
##   1793-Washington       15            4
##   1797-Adams           344          192
##   1801-Jefferson       232          109
##   1805-Jefferson       256          126
```

Task 6

Create your own dictionary or download a dictionary of the web and use it to define a set of features for the UK manifesto texts on immigration.

Task 6 solution

```
ownDict <- dictionary(file = paste0("http://www.kenbenoit.net/courses/",
                                    "essex2014qta/LaverGarry.cat"),
           format = "wordstat")
ukDfm <- dfm(mycorpus, dictionary=ownDict)
topfeatures(ukDfm)
##              VALUES.CONSERVATIVE                ECONOMY.=STATE=
##                              107                             92
## LAW_AND_ORDER.LAW-CONSERVATIVE           INSTITUTIONS.NEUTRAL
##                               59                             52
##                  ECONOMY.+STATE+                ECONOMY.-STATE-
##                               45                             41
##                        CULTURE._                 VALUES.LIBERAL
##                               36                             33
##     ENVIRONMENT.PRO ENVIRONMENT                  GROUPS.ETHNIC
##                               25                             22
```

Next Session

- Estimating Policy Positions from Political Texts