MY472 - Week 7
Textual Data

Friedrich Geiecke
8 November 2021

Introduction

- This week will be an introduction to processing textual data
- Most file formats we work with in this course (.csv, .xml, .json, etc.) use text to store data
- The quantitative analysis of textual data is highly relevant in social science research and beyond
- We will discuss some basic topics, for a full course see MY459 in Lent term

Plan for today

- Character encoding
- Text search: Globs and regular expressions
- Elementary textual analysis
- Coding session

Character encoding

Revisited: Basic units of data

- Bits
  - Smallest unit of storage; a 0 or 1
  - With *n* bits, can store 2^n patterns
- Bytes
  - 8 bits = 1 byte (why 1 byte can store 256 patterns)
  - ``eight bit encoding" represents characters through 8 bit, e.g. `A` represented as `65` $= 01000001$

ASCII

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Encoding

- A “character set” is a list of character with associated numerical representations
- The unique numbers associated with characters are called “code points”
- ASCII: The original character set, uses just 7 bits (2^7) – see http://ergoemacs.org/emacs/unicode_basics.html
- ASCII was later extended, e.g. ISO-8859, using 8 bits (2^8)
- Yet, this became a jungle with no standards, see http://en.wikipedia.org/wiki/Character_encoding

Potential encoding issues

(Wrongly) detected encoding:

- Encoding type/character set is not stored as metadata in plain text files
- Software therefore has to guess which encoding is used which might go wrong
- Assuming the wrong encoding when reading in/parsing a text file leads to import errors and corrupted characters (Mojibake): Underlying bit sequences are translated into the wrong characters

Space:

- 8 bits are much too little to store all known characters
- Encoding all character with say 32 bit, however, would imply a lot of rarely used bits as many common characters would only need the first 7
- Yet, with each character being stored with 32 zeros and ones, this would imply unnecessarily large file sizes

Widely used character encoding today: Unicode

- Created by the Unicode Consortium
- Common Unicode encoding formats: **UTF-8** and **UTF-16** (Unicode transformation format)
- UTF-8 is a variable-width character encoding and by far the most frequent character encoding on the world wide web today
- Variable amounts of bits are used for each character with the first byte/8 bits corresponding to ASCII
- Common characters therefore need less space, but system capable of storing vast amounts of character code points

UTF-8 details

Number of bytes	Byte 1	Byte 2	Byte 3	Byte 4
1	0xxxxxxx			
2	110xxxxx	10xxxxxx		
3	1110xxxx	10xxxxxx	10xxxxxx	
4	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

https://en.wikipedia.org/wiki/UTF-8

Try it out: Create two .txt files, one containing a single line with the character *a*, the other one a single line with the character *ü*. Then check the sizes of both files in bytes which should be different if files are encoded in UTF-8.

Things to watch out for

- Many text production softwares (e.g. MS Office-based products) might still use proprietary character encoding formats, such as Windows-1252
- Windows tends to use UTF-16, while Mac and other Unix-based platforms use UTF-8
- Judging a text file only through looking at it with e.g. a text editor can be misleading: The client may display gibberish but the encoding might still be as intended
- Generally no easy method of detecting encodings (except in HTML meta-data)

Some things to try with encoding issues

To determine the estimated character encoding of a file (note that this estimate might be incorrect)

- Linux, Unix, Mac: For example, `file -I filename.txt`, `file -I filename.json`, etc. in terminal
- Windows: For example, open with Notepad and check field in the lower right hand corner of the window

To change a file's encoding (e.g. to UTF-8)

- Linux, Unix, Mac: For example, `iconv -f ISO-8859-15 -t UTF-8 in.txt > out.txt` in terminal
- Windows: For example, open the text with Notepad, click "Save As", and choose a name and UTF-8 encoding. Alternatively, use PowerShell

For more information see e.g. this Stack Overflow post

Globs and regular expressions

Globs

- Searching and counting specific words in texts is key for quantitative textual anaylsis
- Globs offer a simple and intuitive approach to search through text with wildcard characters
- Glob patterns originally used to search file and folder names

Globs: Exemplary syntax

Wildcard	Description	Examples	Exemplary matches
*****	Any number (also zero) of characters	tax*, *tax*	taxation, overtaxed
?	Single character	??flation	inflation or deflation
[ab], [AB], [17], etc.	List of characters	module-[17].Rmd	module-1.Rmd or module-7.Rmd
[a-z], [A-Z], [0-9]	Range of characters	module-[A-Z].Rmd	module-A.Rmd or module-B.Rmd or module-C.Rmd …

https://en.wikipedia.org/wiki/Glob_(programming)

Regular expressions

- Powerful and much more flexible tool to search (and replace) text
- Different syntax than globs
- Text editors (e.g. Atom) can usually find and replace terms with regular expressions
- Can also be used in many programming languages, e.g. when counting or collecting certain keywords in textual analysis
- In R, we can e.g. use `stringr` or `quanteda` to search for keywords with regular expressions
- Topic could fill lectures itself, we will cover some basics here

Sample text

```
Inflation in the Eurozone

2pm
2:30pm
2.15pm
2 15
11.30
22-30
5-15pm

Münster
Muenster
Munster

@
@JoeBiden
@KamalaHarris
```

Regular expressions: Syntax

- Regular expressions can consist of literal characters and metacharacters
- **Literal characters**: Usual text
- **Metacharacters**: ^ $ [] () {} * + . ? etc.
- When a meta character shall be treated as usual text in a search, escape it with (unless it is in a set []) \
- For example, searching **.** in regex notation will select any character, but searching **\.** will select the actual full stop character

Syntax: Specifying characters (1/2)

- .: Matches any character (also white spaces)
- \d: Matches any digit 0-9
- \w: Matches any character a-z, A-Z, 0-9, _
- \s: Matches white spaces
- Capitalised versions negate: \S matches everything that is not a white space etc.

Syntax: Specifying characters (2/2)

- ^: Matches characters at the beginning of the line or string, e.g. ^M will select all capital m at the beginning of strings or lines
- $: Matches characters at the end of the line or string, e.g. m$ will select all lowercase m at the end of strings or lines
- []: Character set, e.g. [a-zA-Z] selects single characters from the Latin alphabet in lower and upper case letters, [ai] selects characters that are “a” or “i”, [0-9] digits from 0 to 9
- [^]: In brackets, ^ has a different meaning namely “not”, e.g. [^a-z] selects all characters that are not from the lower case alphabet

Syntax: Selecting sequences of characters

In order to select whole words, we need to add quantifiers to individual characters:

- *: Zero or more times, e.g. in[a-z]* will select *in* and also *inflation* in a search; .* represents all characters and white spaces
- +: One or more times, e.g. in[a-z]+ will not select *in* but *inflation*
- ?: Denotes optional characters, e.g. re?ally will select really and rally
- {}: Specifies lengths of sequences, e.g. \d{3} selects sequences of 3 digits, \w{3,4} selects sequences between 3 and 4 general characters, and \d{3,} selects sequences of at least 3 digits

Syntax: Boolean or and capturing groups

- |: Boolean or
- (): Capturing groups, e.g. (ue?|ü) selects u, ue, and ü. This means that when searching text, the regular expression M(ue?|ü)nster will find *Münster*, *Muenster*, and *Munster*. The captured groups can also be referenced with integer counts while e.g. replacing which can be very helpful
- https://en.wikipedia.org/wiki/Regular_expression

Regular expressions in R and beyond

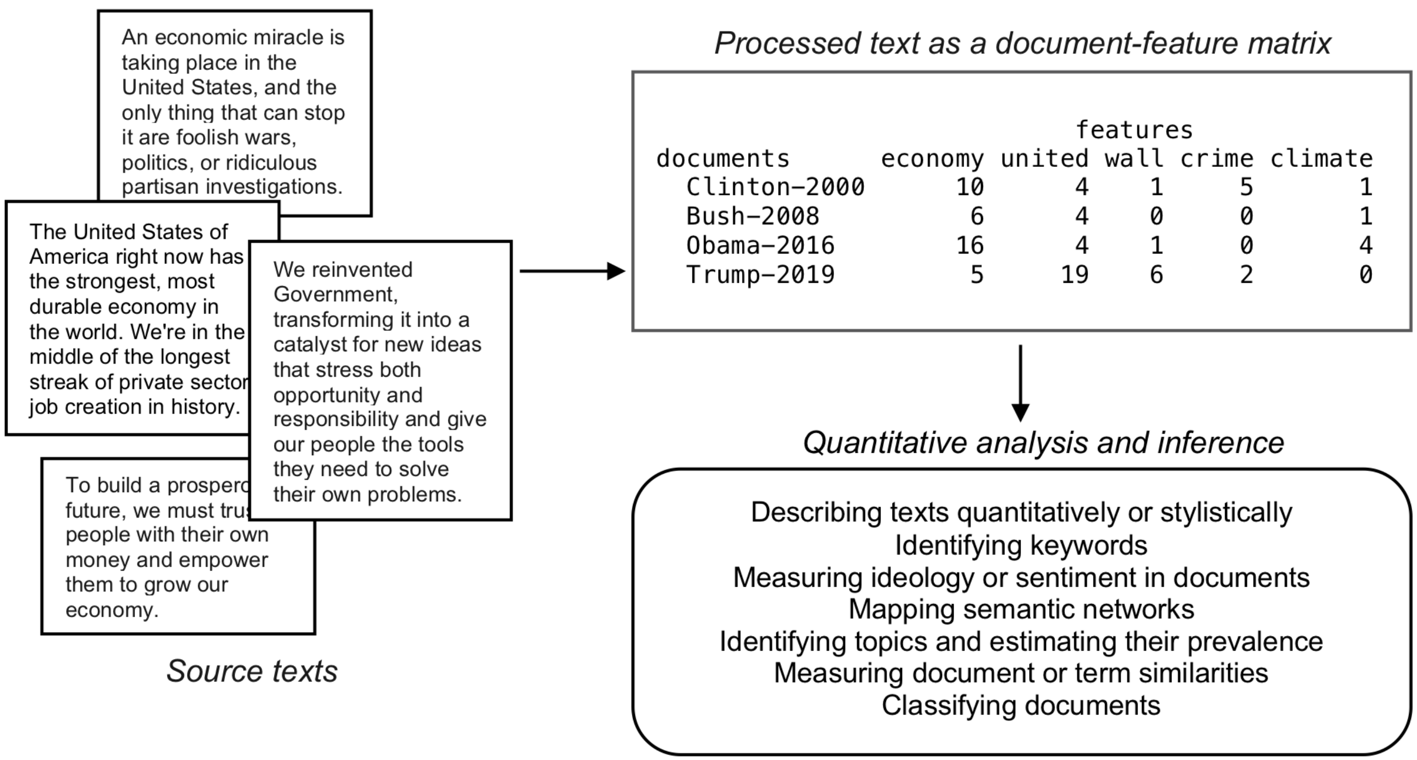
- Regular expression can e.g. used for very flexible word searches in the `quanteda` package
- A good package for strings in R that also allows searching characters with regular expression is `stringr`. Functions such as `str_view` allow to view results of searches with regular expressions and `str_extract` allows to extract keywords from strings through regular expressions
- Detailed discussion of strings and regular expressions with `stringr` in R here
- R markdown with many examples here
- Some good general discussions of the topic also on Youtube, e.g. here
- In depth treatment of regular expression (programming language independent): *Mastering Regular Expressions* by Jeffrey E. F. Fried

Elementary textual analysis

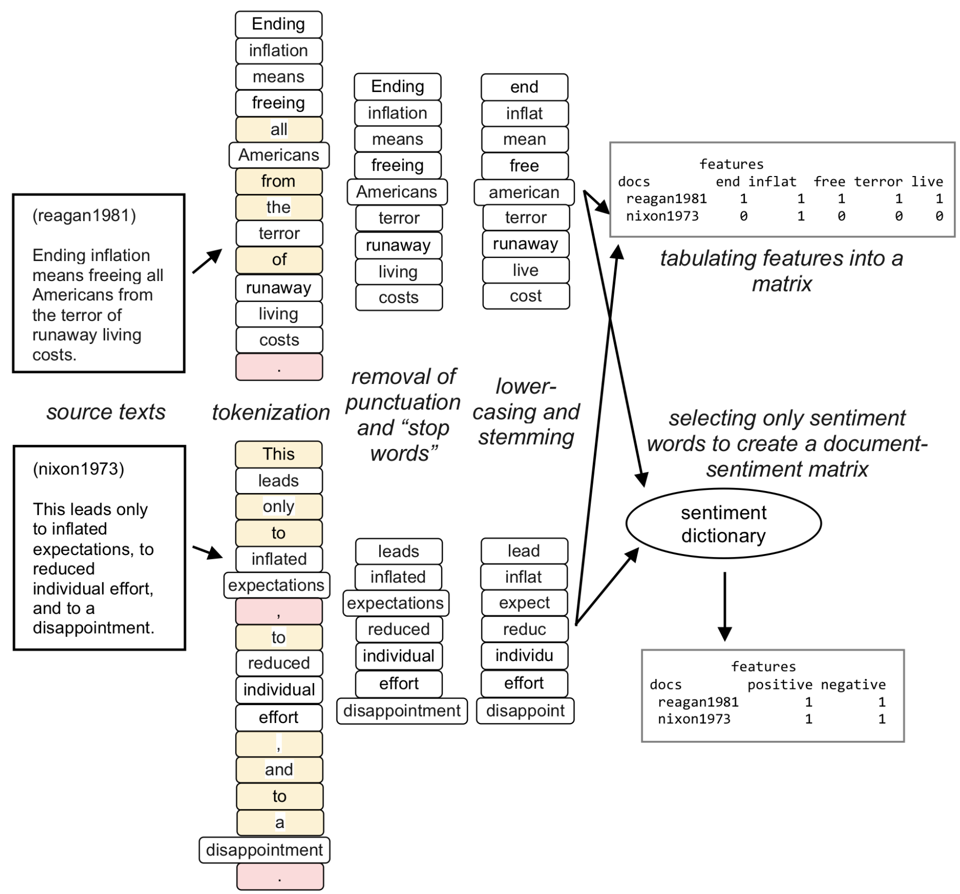
Moving from texts to numbers

- To analyse text quantitatively, the key question is how to move from text to numbers
- We will look at very common approaches that count words in documents
- This abstracts from the sequential dependency of words (beyond n-grams) and is sometimes referred to as a bag-of-words approach

Common workflow



Common workflow: Tokenization step in more detail + additional dictionary method



Some key concepts

- Document-feature matrix (dfm): As many rows as documents, as many columns was words/features after cleaning
- Stopwords: Common words such as “the”, “to”, etc.
- Stemming: Heuristic process to obtain the stem of words which in essense groups terms, see the following link for a detailed discussion
- n-grams: Sequences of words, e.g. bigrams (2) or trigrams (3). For example allows to record “not good” as a feature

Dictionary approaches

- Map each word or phrase to a “dictionary” of words, e.g. associated with a known “sentiment” or psychological state or with certain topics
- Treats matches within each dictionary as equivalent
- Examples: Linguistic Inquiry and Word Count, or the General Inquirer

Dictionary example (from LIWC 2015)

```
Dictionary object with 1 key entry.
- [posemo]:
- like, like*, :), (:, accept, accepta*, accepted, accepting, accepts, active, …
interests, invigor*, joke*, joking, jolly, joy*, keen*, kidding,
kind, kindly, kindn*, kiss*, laidback, laugh*, legit, libert*,
likeab*, liked, likes, liking, livel*, lmao*, lmfao*, lol, love, loved, lovelier, ...
```

Problems with dictionary approaches

- Polysemy – multiple meanings: The word “kind” has three!
- From State of the Union corpus: 318 matches
    - kind/NOUN – 95%
    - kind (of)/ADVERB – 1%
    - kind/ADJECTIVE – 4%
- These are known as false positives
- Other problem: False negatives (what we miss)
    - Missed: kindliness
    - Also missed: altruistic and magnanimous
- How to treat conflicting keywords in the same string? “Had a great day … not.”

Further topics

- Text classification: Store labels for individual documents (e.g. spam or no spam, positive or negative sentiment) in a vector y and use the dfm as feature matrix X with variables/features being the word counts in documents. Use e.g. logistic regression, random forest, etc. to predict labels $\hat{y}$
- Topic models: Find sets of words in large amounts of documents which tend to appear together
- Word and document embeddings: Represent words or documents as vectors and analyse their distances/similarities in an automated way
- Neural network based approaches that can take the sequential nature of text into account quite well
- etc.

Coding session

Markdown file this week

- 01-regular-expressions-in-r.Rmd
- 02-textual-analysis.Rmd