Week 4: Textual Data

LSE MY472: Data for Data Scientists
https://lse-my472.github.io/

Autumn Term 2024

Ryan Hübert

Slides last updated: 22 October 2024

Introduction

- This week we will focus on 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 analyses, but for a full course see MY459 in Winter Term

Plan for today

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

Character encoding

Useful background: the basic units of data storage

- Bits
  - Smallest unit of storage on a computer: a 0 or 1
  - With n bits, can store 2^n patterns
  - E.g., 2 bits of storage gives four possibilities: 00, 01, 10, 11
- Bytes
  - 8 bits $=$ 1 byte
  - Hence, 1 byte can store 256 patterns
  - kilobytes, megabytes, gigabytes, etc. are metric aggregations of bytes (roughly... see https://en.wikipedia.org/wiki/Byte#Multiple-byte_units)

Character encoding

- **Character**: “smallest component of written language that has semantic value’ ’
  - See https://unicode.org/glossary/#character
- **Character set**: list of characters with associated numerical representations
- **Code points**: the unique “numbers” associated with characters in a character set
  - These can be expressed in multiple formats (hex, dec, etc.)
- The mapping between character and code points is called an **encoding**
- Encodings use differing number of bits to represent characters: 7-bit, 8-bit, 16-bit, etc.

The origins of encoding: ASCII

- ASCII: the original character set/encoding, uses just 7 bits
  - Could only encode up to $2^7 = 128$ characters... not enough!
- ASCII was later extended to 8 bits (2^8), e.g. ISO-8859-1
  - Now could encode $2^8 = 256$ characters... still not enough!
- Full tables here: original ASCII, ISO-8859-1
- As you can imagine: different languages, different characters $\rightarrow$ different character sets and encodings
- This is a mess... see http://en.wikipedia.org/wiki/Character_encoding

ASCII and “extended ASCII” examples

Character	Code Point (dec)	ASCII (7-bit)	ISO-8859-1 (8-bit)
$+$	43	0101011	00101011
6	54	0110110	00110110
A	65	1000001	01000001
h	104	1101000	01101000
ñ	164		10100100
¼	172		10101100
α	224		11100000

Potential encoding issues

1. **Wrongly detected encoding**

- ➜ Encoding type/character set is not stored as metadata in plain text files (e.g., .csv, .tab, .txt, .md, .Rmd, etc.)
- ➜ Software used to access plain text files guesses which encoding is used, sometimes incorrectly
- ➜ Assuming the wrong encoding when reading in/parsing a text file leads to import errors and corrupted characters
- ➜ This is known as Mojibake: underlying bit sequences are translated into the wrong characters
  - ➜ We'll see some examples.

Potential encoding issues

2. Space constraints

- Each bit used to represent a character uses storage
- 8 bit encoding uses less storage, but is not enough for a character set that has all known characters
- Encoding with 32 bits ($2^32 \approx 4.3$ billion code points), however, ensures all known characters can be stored
- But, in most situations, it implies storing a lot of "unused" bits and 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 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 examples

UTF-8 is a variable width encoding standard: 8, 16, 24, 32 bits

	Code	Byte 1	Byte 2	Byte 3	Byte 4
&	U+0026	00100110			
u	U+0075	01110101			
ü	U+00FC	11000011	10111100		
Д	U+0414	11010000	10010100		
ሊ	U+120A	11100001	10001000	10001010	
🫠	U+1FAE0	11110000	10011111	10101011	10100000

See: https://dencode.com/en/string/bin,
https://www.rapidtables.com/convert/number/ascii-to-binary.html

UTF-16 examples

UTF-16 is a variable width encoding standard: 16 or 32 bits

	Code	Byte 1	Byte 2	Byte 3	Byte 4
&	U+0026	00000000	00100110		
u	U+0075	00000000	01110101		
ü	U+00FC	00000000	11111100		
Д	U+0414	00000100	00010100		
ሊ	U+120A	00010010	00001010		
🫠	U+1FAE0	11011000	00111110	11011110	11100000

See: https://dencode.com/en/string/bin,
https://www.rapidtables.com/convert/number/ascii-to-binary.html

UTF-32 examples

UTF-32 is a fixed width encoding standard: 32 bits

	Code	Byte 1	Byte 2	Byte 3	Byte 4
&	U+0026	00000000	00000000	00000000	00100110
u	U+0075	00000000	00000000	00000000	01110101
ü	U+00FC	00000000	00000000	00000000	11111100
Д	U+0414	00000000	00000000	00000100	00010100
ሊ	U+120A	00000000	00000000	00010010	00001010
🫠	U+1FAE0	00000000	00000001	11111010	11100000

See: https://dencode.com/en/string/bin,
https://www.rapidtables.com/convert/number/ascii-to-binary.html

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 Unix-based platforms use UTF-8
- Text editors can be misleading: the client may display mojibake but the encoding might still be as intended
- Generally, no easy method of detecting encodings in basic text files

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 (see e.g. this Stack Overflow post)

- 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

Some things to try with encoding issues (in R)

In R, e.g. via `readr` (for more discussion, see R4DS)

- For a character vector `x`, obtain texts assuming a different encoding with `parse_character(x, locale = locale(encoding = "Latin1"))`
- Make guess about encoding with `guess_encoding(charToRaw(x))`

(In my experience: python has more robust tools for dealing with encoding issues)

Resources

Character encoding is complicated, but VERY important

Highly recommend:

https://kunststube.net/encoding/

And also Wikipedia pages on:

- character encoding
- ASCII
- Unicode
- UTF-8

Globs and regular expressions

Globs

- Searching and counting specific words in texts is key for quantitative analysis of textual data
- 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: examples of syntax

Wildcard	Description	Examples	Examples of 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 ...

Regular expressions

- Powerful and much more flexible tool to search (and replace) text
- Different syntax than globs
- Many editors that work with plain text (e.g. Rstudio, VS Code) 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 text 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

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;
  - We could use `.*` to represent 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, which can be very helpful when replacing text
- https://en.wikipedia.org/wiki/Regular_expression

Regular expressions in R and beyond

- `stringr` is a great package for strings that uses regular expressions:
  - `str_view()` show results of searches with regular expressions
  - `str_extract()` allows you to extract keywords from strings through regular expressions
  - `str_replace()` finds and replaces regular expressions
- Detailed discussion of strings and regular expressions with `stringr` in R here
- R markdown with many examples here
- Regular expressions are used for flexible word searches in the `quanteda` package

More resources

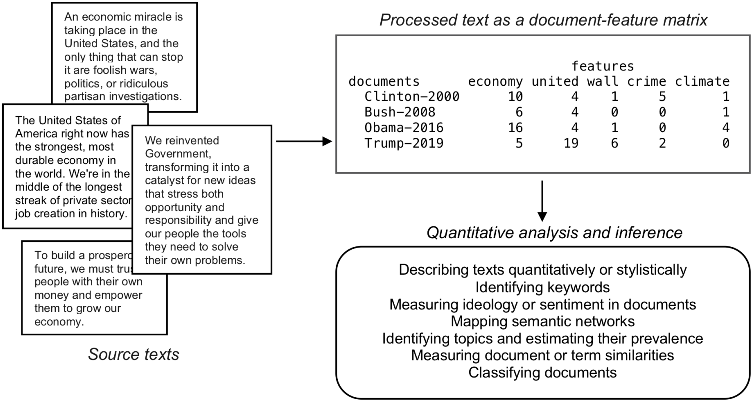
- Some good general discussions of the topic also on Youtube, e.g. here
- In depth treatment of regular expression (programming language independent): *Mastering Regular Expressions* by Jeffrey E. F. Fried
- There are several great websites to test regular expressions, which allow you to provide sample text, write a regex and show you matches
  - regxr.com
  - regex101.com

Elementary text analysis

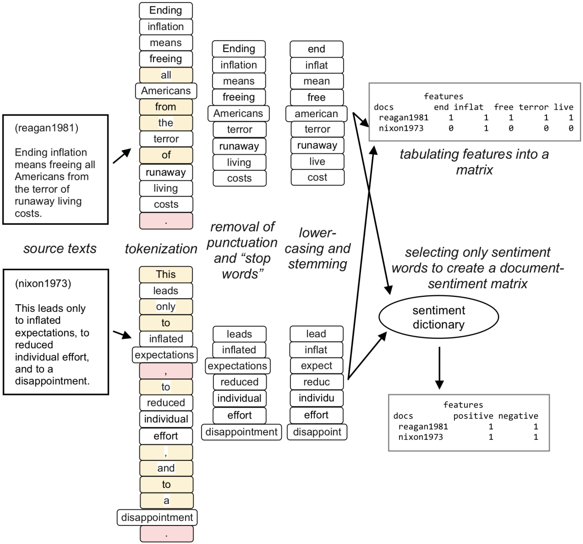
Moving from texts to numbers

- 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: Tokenisation + dictionary method



Some key concepts

- Document-feature matrix (dfm): As many rows as documents, as many columns was words/features after cleaning
  - This is also often called a "document-term matrix" or dtm
- Stopwords: Common words such as "the", "to", etc.
- Stemming: Heuristic process to obtain the stem of words which in essence 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, ...
interests, invigor*, joke*, joking, jolly, joy*, ...
kind, kindly, kindn*, kiss*, laidback, laugh*, ...
likeab*, liked, likes, liking, livel*, lmao*, ...
```

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: How do we use a document-feature matrix to predict document labels (e.g. spam/not spam)?
- Topic models: How do we find sets of words which tend to appear together?
- Word and document embeddings: How can we represent words or documents as vectors and analyse their distances/similarities?
- How do we take into account the sequential nature of text?
- Etc.

Coding

Markdown files

- 01-regular-expressions-in-r.Rmd
- 02-text-analysis.Rmd
- 03-parsing-pdfs.Rmd