Week 1: Introduction and Foundations

LSE MY459: Quantitative Text Analysis
https://lse-my459.github.io/

Ryan Hübert

Slides last updated: 20 January 2025

So much of human interaction involves text

- ➜ Social media
- ➜ Political speech
- ➜ Literature and artistic expression
- ➜ Laws and legal opinions
- ➜ Government records
- ➜ News articles

Texts reveal important things about the world

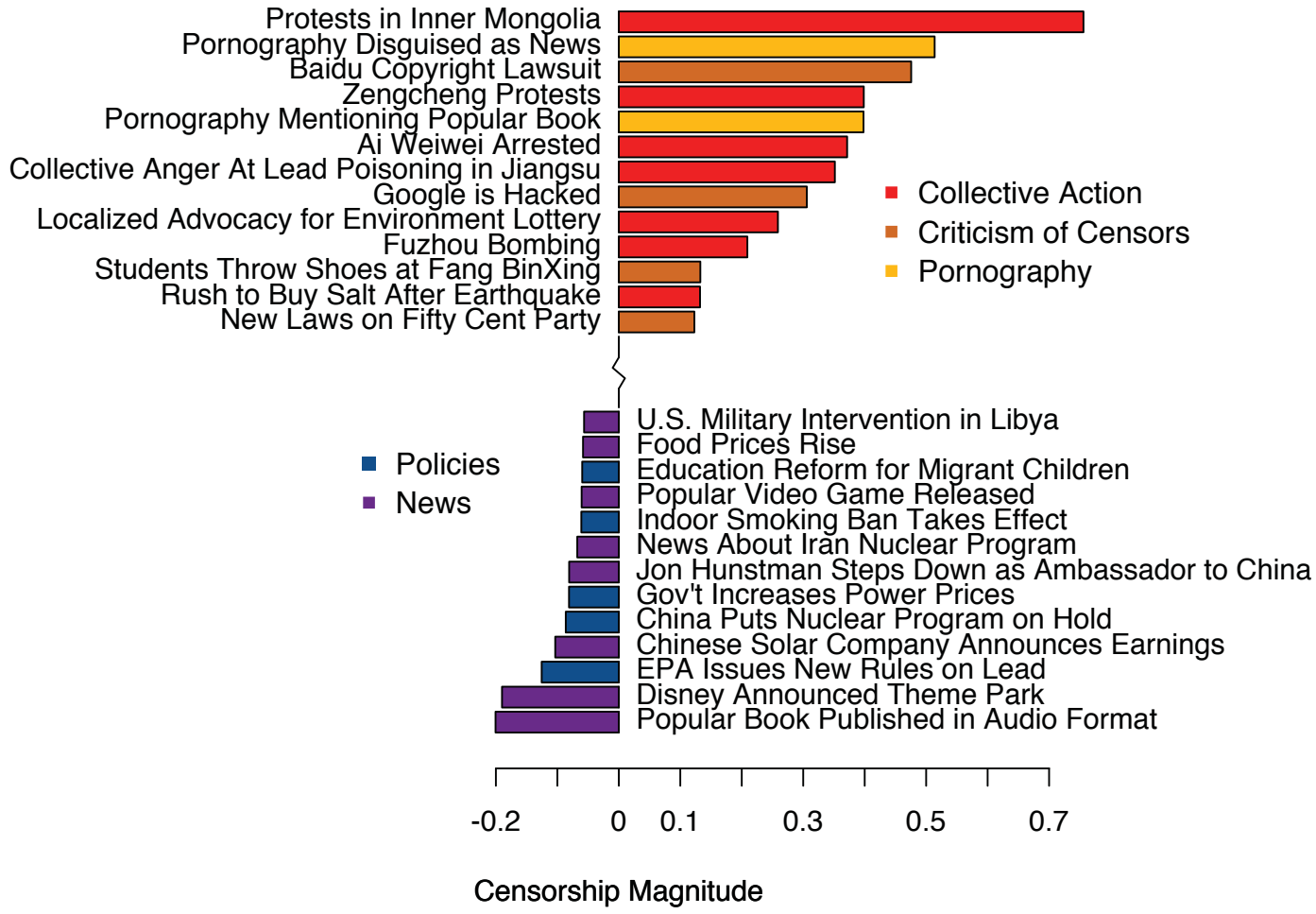
But what exactly??

That's what you're here to learn about

A neat example: King, Pan, & Roberts (2013)

A neat example: King, Pan, & Roberts (2013)

Figure 4. Events with Highest and Lowest Censorship Magnitude



Outline for today

- Course logistics
- Why quantitative text analysis?
- Character encoding
- Digitally storing texts
- Working with text in R

Course logistics

Course schedule

Lectures: Mondays from 10:00 to 12:00 in CLM.3.02

Classes/seminars: only in weeks 2, 4, 7, 9, 11:

➜ *See LSE timetable for times and locations!*

No lectures or classes during Reading Week (week 6)

If you want to audit, must submit request to department

➜ You cannot attend class/seminar if you audit

Teaching team

Dr. Ryan Hübert
r.hubert@lse.ac.uk

- ➜ Course convenor for MY459 and MY559
- ➜ Leads all lectures and classes/seminars during weeks 1-7
- ➜ Book office hours through StudentHub

Dr. Friedrich Geiecke
f.c.geiecke@lse.ac.uk

- ➜ Course convenor for MY360
- ➜ Leads all lectures and classes/seminars during weeks 8-11
- ➜ Book office hours through StudentHub

Prerequisites

We will assume:

- You have access to a laptop that can run the required software for the course and that you will bring to class
- You are comfortable with R and RStudio, and know how to work with Rmarkdown/Quarto documents
- You are capable of using git and GitHub
- You have exposure to elementary linear algebra and have taken quantitative methods courses up to and including the equivalent of MY452 (regression)
- You are willing to engage with some amount of mathematical notation and concepts
- You have computing habits that minimise risk of problems

Course resources

Course website: https://lse-my459.github.io/

- Course schedule
- Lecture and seminar materials
- Reading lists: "primary" and "further"
  - We will draw *heavily* from textbook by Grimmer, Roberts and Stewart (abbreviated GSR)
- Links to sample code, exercises and datasets

Moodle pages: MY459, MY360

- Announcements
- Some supporting materials

Assessments

Formative: there is one formative problem set during WT

- ➜ This will be provided later in the term
- ➜ We will provide you with solutions for you to review

Summative: there is one two-hour exam during ST, which is worth 100% of your mark

- ➜ Note: different exams for MY360 and MY459/MY559

Please review department and school policies and information on assessments, but keep in mind:

- ➜ Distinctions/First Class Honours (70-100) typically only go to students who perform *exceptionally* well

Generative AI

We **allow** GitHub Copilot/ChatGPT/other generative AI assistants to aid in your learning during the course

But beware:

- ➜ You need proficiency to recognise good code and fix broken code; do not rely too heavily on AI
- ➜ These tools routinely produce broken code and incorrect information
- ➜ You are sharing your thoughts, ideas, and work with models that are proprietary
- ➜ You will *not* be able to use these tools on the final exam

Support

For administrative support:

- For auditing, registration, extensions, late assignments and department/school policy questions: methodology.admin@lse.ac.uk
- For questions about course platforms, including Moodle, website or GitHub Classrooms: r.hubert@lse.ac.uk
- For other administrative issues related to MY459 or MY559: r.hubert@lse.ac.uk
- For other administrative issues related to MY360: f.c.geiecke@lse.ac.uk
- For questions about course material: schedule office hours for the relevant instructors via StudentHub

Course outline

See the website.

There will be *some* overlap with other courses, e.g. MY472 and MY474

- ➜ This is a feature not a bug
- ➜ But here, our primary motivation is distinctive: do analysis with *text*
- ➜ We'll often discuss the peculiarities of applying "general" methods to texts
- ➜ Plus: many students didn't take MY472, MY474!

Preparing for lectures and seminars

- Make sure that you have both R and Rstudio installed and working on your computer
- Make sure that you have a GitHub account and that you know how to use git/GitHub
  - We will not teach this in the course, but we will provide support/guidance
  - Please review: https://github.com/lse-my459/lectures/tree/master/github-user-guides
- During the course, we will generally expect that you are constantly reviewing course materials and other sources (as needed)
- QTA is a massive topic; this class will move fast!

Why quantitative text analysis?

Why quantitative text analysis?

Hopefully obvious why we want to study texts — but why *quantitatively*?

Justin Grimmer's texts-as-haystack metaphor:

- ➜ Analysing a straw of hay: understanding *meaning* of a text
  - ➜ Humans are great! But computer struggle
- ➜ Organising haystack: describing, classifying, scaling texts
  - ➜ Humans struggle. But computers are great!
  - ➜ (What this course is about)

Quantitative text analysis (QTA) *improves* reading

(Some parallels to how we are coming to understand generative AI)

The three "cultures" of QTA

QTA is built on a foundation largely from computer science

- ➜ Usually known as **natural language processing (NLP)**
- ➜ Often focused on more "technical" issues, often with various industry applications
- ➜ Jurafsky and Martin (20XX) is the canonical textbook

QTA has come to the "liberal arts" relatively recently

- ➜ Scholars in **digital humanities** use QTA to learn about texts
- ➜ Social scientists use QTA to learn about bigger things in the social world, thus treating **text as data**

Three distinct "cultures" of QTA, but they are intertwined

The three "cultures" of QTA

Unapologetic starting point for this course: we are here to do social science using QTA methods

GSR was written for the social science culture—it's very useful

But... we cannot ignore the major elephant in the room: the recent advances in large language models

- ➜ This is still largely being developed in the CS culture
- ➜ Expanding rapidly within SS culture
- ➜ End of this course will cover some of the methods behind LLMs at an introductory level and provide social science applications

Core assumptions of QTA in the social sciences

1. Texts represent an observable implication of some underlying thing of interest, such as:

   - ➜ An attribute of the author
   - ➜ A sentiment or emotion
   - ➜ Salience of a political issue

2. Texts can be represented by extracting their "features"

   - ➜ most common approach: **bag of words**
   - ➜ many other possible definitions of "features" (e.g. word embeddings)

3. A tabular dataset (a **document-feature matrix**) can be analysed using quantitative methods to produce meaningful and valid estimates of the underlying thing of interest

Criticism of QTA

QTA has critics!

For example Da (2019) argues that in QTA:

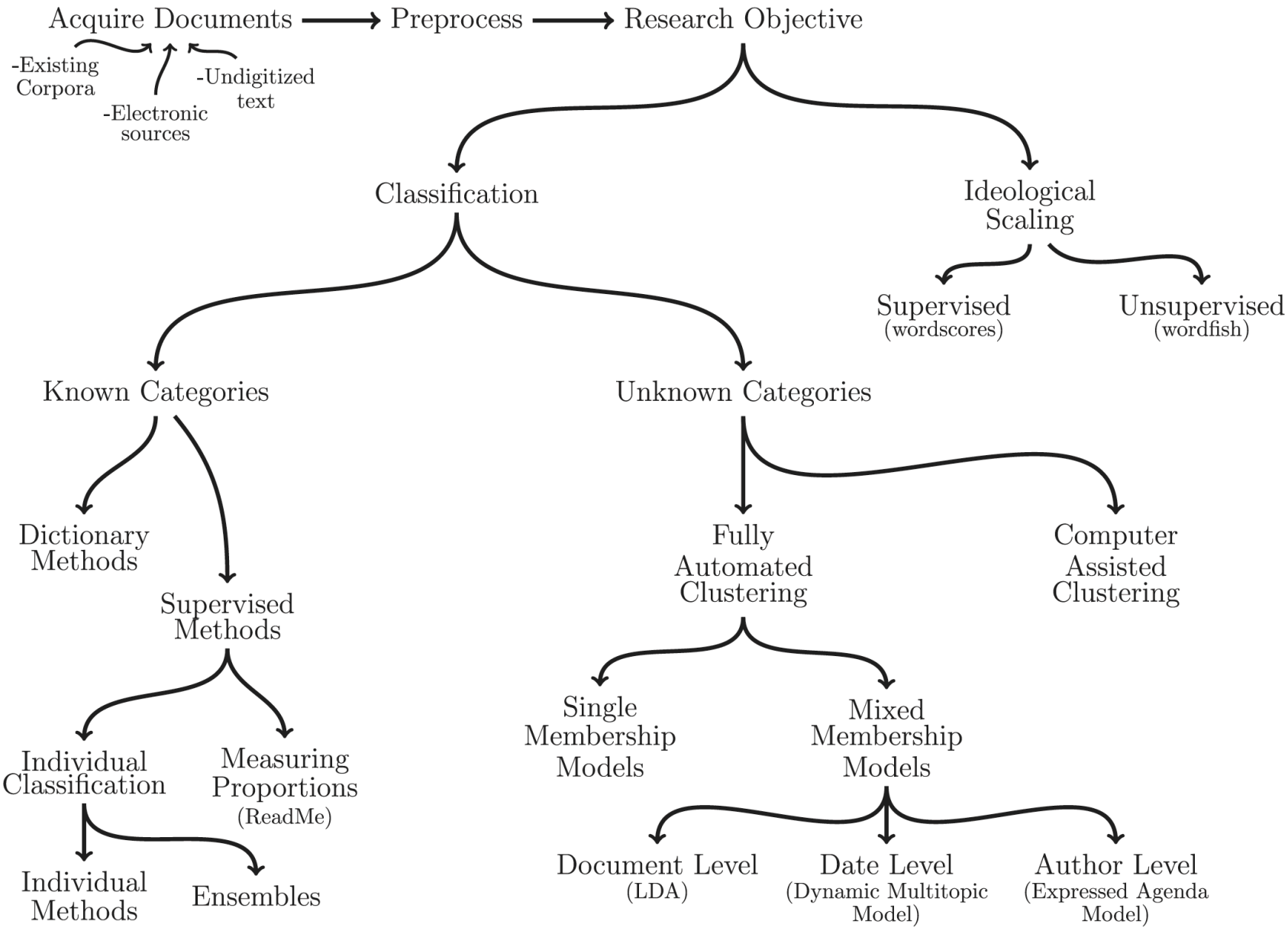
- "what is robust is obvious (in the empirical sense) and what is not obvious is not robust"

In other words: when we learn something interesting from QTA, it is not replicable and/or doesn't capture something general or "real"

A more cynical (somewhat euphemistic) way to put it: QTA involves too many "researcher degrees of freedom"

This is a problem for social scientists: our job is to produce new, general knowledge about the world

Criticism of QTA



Source: Grimmer and Stewart (2013)

Radical agnosticism

GRS's response: **radical agnosticism** with six principles

1. Social science theories and substantive knowledge are essential for [QTA] research design

   ➜ A QTA project needs to make sense as a work of substantive social science; do not simply do QTA because you can

2. Text analysis does not replace humans—it augments them

   ➜ Even with QTA, you cannot avoid the old fashioned work of *reading* and *thinking*; this should take most of your time

3. Building, refining and testing social science theories requires iteration and cumulation

   ➜ It's fine to take an inductive approach

Radical agnosticism

4. Text analysis methods distill generalizations from language

   ➜ All learning necessarily involves simplifying the world to extract more general lessons—QTA is no different

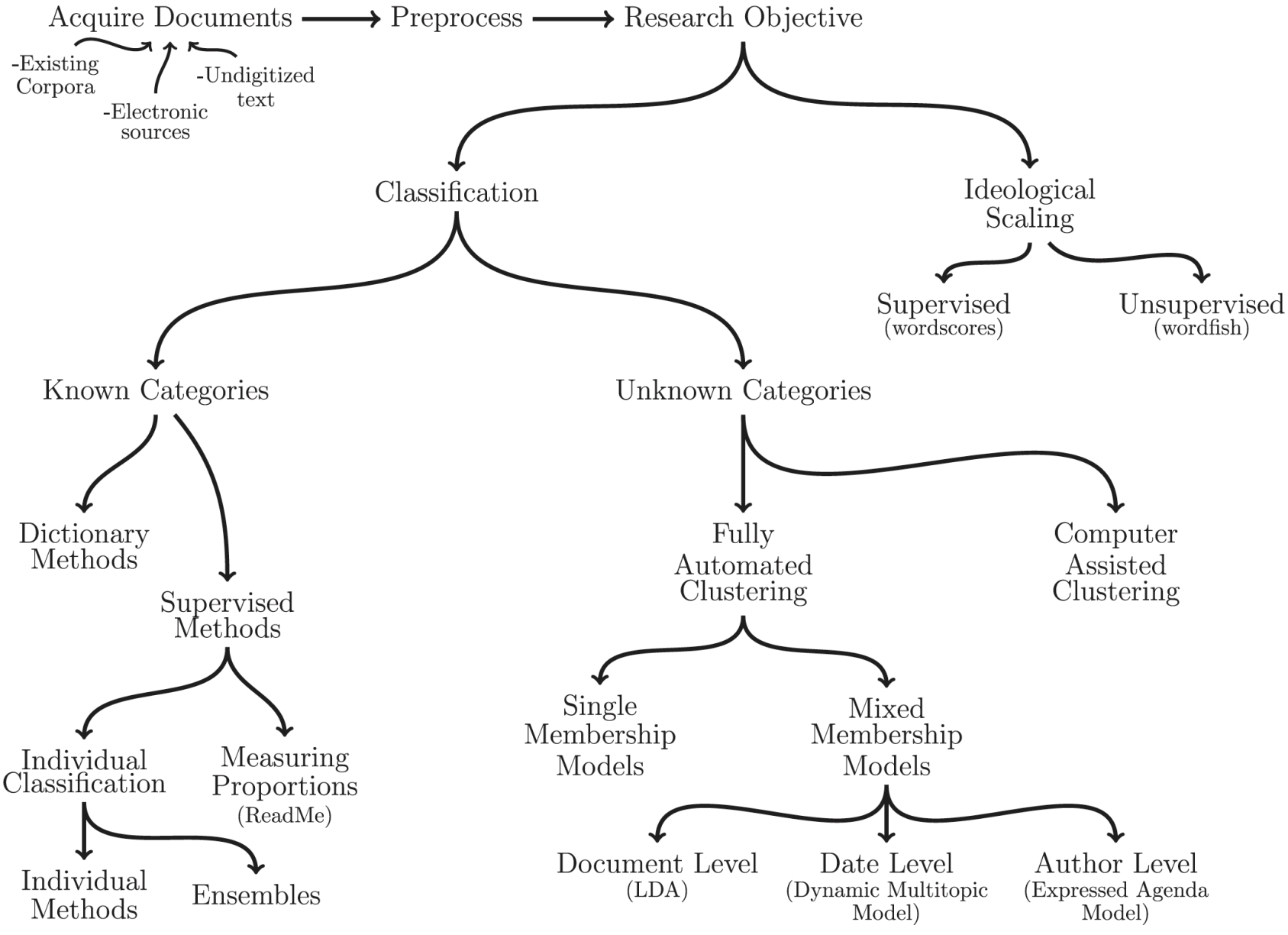
5. The best method depends on the task

   ➜ Before using a method from QTA, it should be clear what you are trying to do *in social scientific terms*

6. Validations are essential and depend on theory and task

   ➜ Perhaps the most important of all — check your outputs and spend time verifying and explaining that they make sense in context of your substantive question
   ➜ There are a wide range of techniques for doing this

Radical agnosticism



The big overview of QTA in the social sciences

Selection and representation: deciding what texts will be analysed; digitising and/or converting into common format; quantifying – roughly weeks 1-2

Discovery: "process of creating new conceptualizations or ways to organize the world" (GRS, p. 4) – roughly weeks 7-8

Measurement: "process where concepts are connected to data, allowing us to describe the prevalence of those concepts in the real world" (GRS, p. 4) – roughly weeks 3-5

Inference:

- ➜ prediction: using texts to *predict* things in the real world – roughly weeks 9-11
- ➜ causal inference: using texts to learn about what *causes* what in the real world – sadly, not enough time

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.)
- 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
  - E.g., decimal code point for "A" is 65, comprised of these bits: `1000001` (original ASCII) `01000001` (extended ASCII)
- 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

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		
𝓛	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

Digitally storing text

Digitally storing text

Textual data can be stored in a variety of digital file formats, but there are some ones you may encounter a lot:

- **Binary files**: text is computer-readable, but not human-readable; stores text as binary, e.g. `.bson`
- **Plain text files**: text is computer and human-readable; stores encoded characters but no other information
  - "Simple" text files, e.g. `.txt`
  - **Mark up** files, that store formatting as plain text "tags", e.g. `.md`, `.qmd`, `.Rmd`, `.html` (see Wikipedia explainer)
  - Programming scripts, e.g. `.R`, `.py`,
  - Text structured for data storage (including delimiters, containers, etc.), e.g. `.csv`, `.tab`, `.json`, `.xml`
  - Many word processor documents, e.g. `.docx` (technically XML)

Digitally storing text

- **Rich text files**: text is computer and human-readable; stores encoded characters and some formatting metadata, e.g. `.rtf`
  - Seems to be less common these days(?)
- **Cloud based documents**: text that is available on a cloud-based editor, e.g., Google Docs
  - These are usually plain text formats "under the hood"
  - Often can be exported in some kind of plain text format
- **"Image-like" files**: text is human-readable, might be computer-readable; stores text as images rendered on a page, e.g., `.pdf`, `.jpg`, `.png`, `.tif`, etc.

Optimal character recognition (OCR)

Image-like files: often how analog texts are digitised

➜ For example, scans of old paper documents

PDFs are most common image-like file type for storing text

➜ There are different types, see https://pdf.abbyy.com/learning-center/pdf-types/

➜ Some complications around document security

To extract text:

➜ PDFs that are digitally generated; can export text directly in e.g., Adobe Acrobat

➜ Otherwise, need to use **optimal character recognition (OCR)** — lots of tools for this

Potential encoding issues

1. Wrongly detected encoding

- When a plain text file is initially saved, it has an encoding
- But encoding is *not* stored as metadata in plain text files
- 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

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

Things to watch out for

- Many text production applications (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)

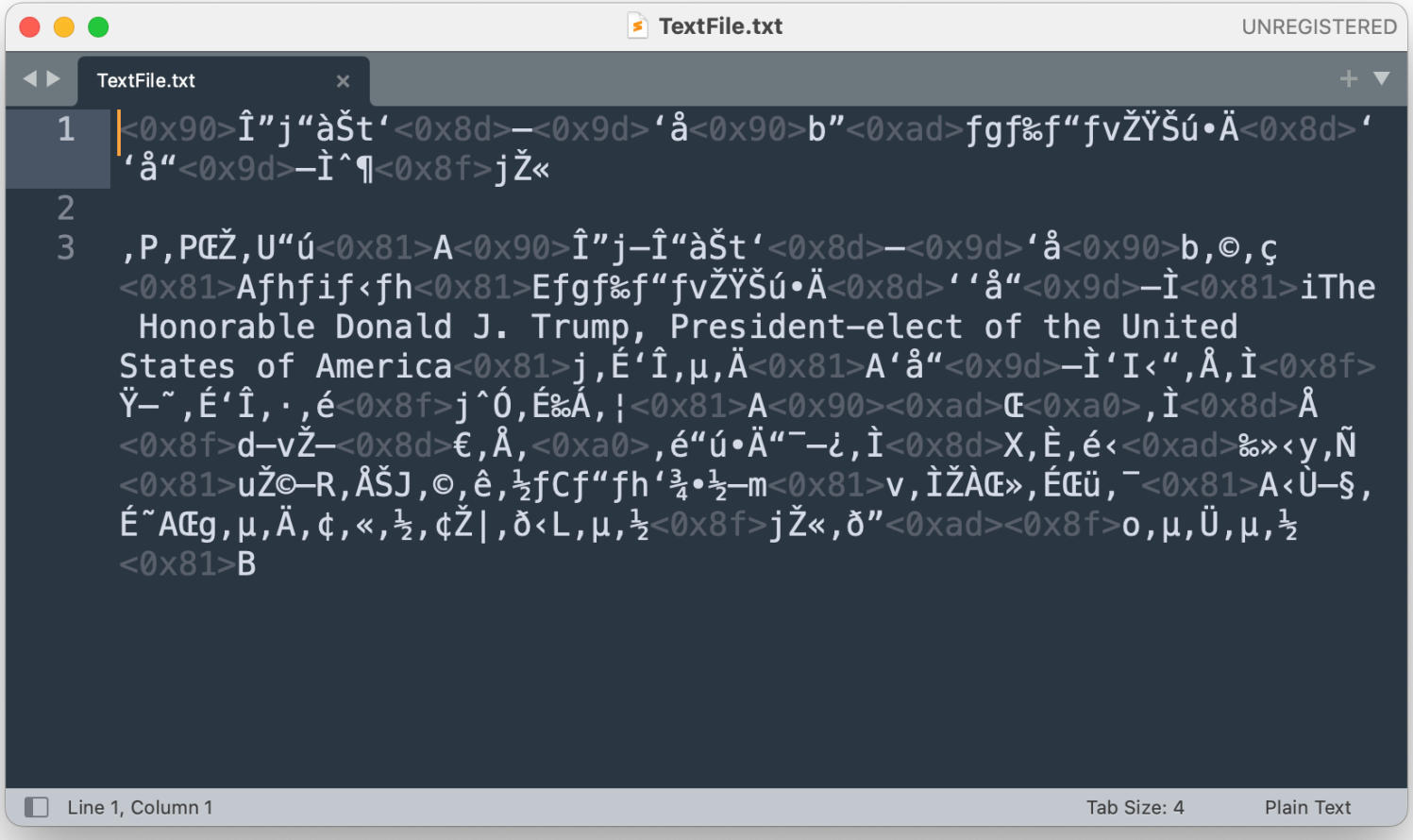
- Linux, Unix, Mac: For example, `iconv -f ISO-8859-15 -t UTF-8 in.txt > out.txt` in terminal
- Windows: For example, open the text with Notepad, click "Save As", and choose a name and UTF-8 encoding. Alternatively, use PowerShell

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))`
- Read a file with known encoding and write files using the default encoding of UTF-8

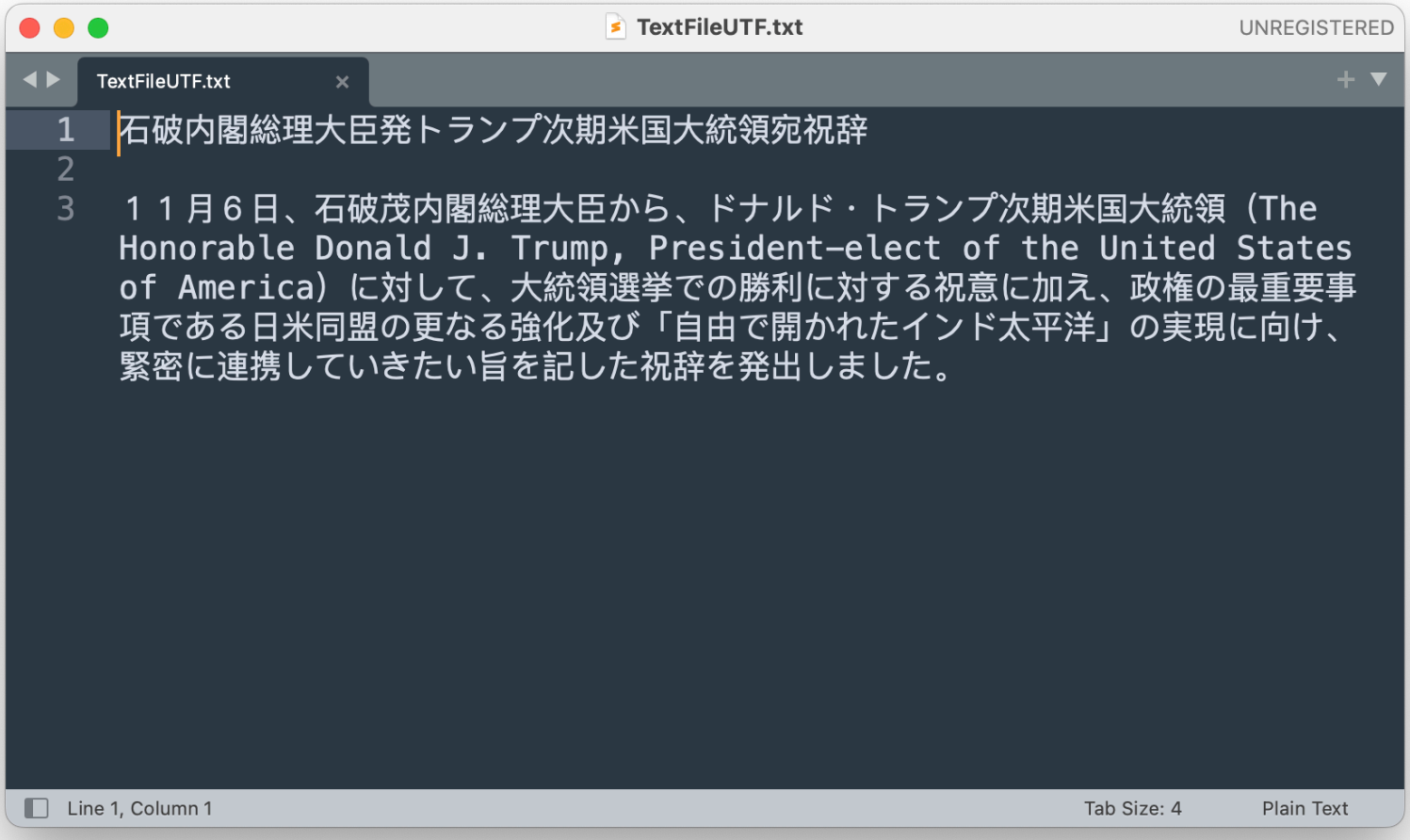
Illustrative example



Illustrative example

```
library("readr")
text <- read_file("TextFile.txt",
                  locale = locale(encoding="Shift_JIS"))
write_file(text, "TextFileUTF.txt")
```

Illustrative example



Some unsolicited advice

Get in habit of storing work and data in plain text formats

- ➜ Easier to work with
- ➜ "Safer" archiving
- ➜ More portable across contexts

If you like writing formatted text that's easy on your eyes (as I do!), consider moving over to:

- ➜ Markdown/Rmarkdown
- ➜ Quarto
- ➜ LaTeX

Resources

Character encoding is complicated, but VERY important

Highly recommend:

https://kunststube.net/encoding/

And also Wikipedia pages on:

- plain text
- formatted text
- character encoding
- ASCII
- Unicode
- UTF-8

Working with text in R

Strings/characters in R

An object consisting of plain text characters is often referred to as a **string** or a **character string**

In R, a vector of strings is a character vector; more generally, character and string are interchageable

```
my.string <- "Hello world!"
my.chr.vec <- c("Hello Earth!", "Hello Mars!")
```

With "base R" you can look at substrings and paste things:

```
paste(my.string, " Hello sun!")
```

```
[1] "Hello world!  Hello sun!"
```

```
substr(my.string, 1, 4)
```

```
[1] "Hell"
```

Strings/characters in R

In this class, we'll make use of powerful tools available in the `tidyverse` suite of R packages

The `stringr` package is one such package for working with strings

For example, with `stringr` you can

- find patterns, e.g., `str_detect()`
- "mutate" strings, e.g., `str_replace()`, `str_to_lower()`
- combine or split strings, e.g., `str_c()`, `str_split()`
- clean strings, e.g. `str_squish()`, `str_trim()`, `str_pad()`

Can do some of this in base R too, and in python, `re` library is similar to `stringr`

Review the docs: `tidyverse`, `stringr`, `stringr` cheatsheet

Regular expressions

- Searching and counting strings is key for QTA
- **Regular expressions** (usually called "regex") provide a powerful and flexible tool to search (and replace) text
  - Similar to globs, but more powerful
- 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, both `stringr` and `quanteda` allow for regex searches
- 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
- https://en.wikipedia.org/wiki/Regular_expression

Regular expressions: syntax

With regular expressions, you can

1. Specify characters, including with "wildcards"
2. Specify sequences of characters
3. Use booleans and "capturing groups"

Detailed discussion of strings and regular expressions with `stringr` in R here

R markdown with many examples here

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