Lecture 1 — Text as Data

Foundations & Preprocessing

Tamara Grechanaya
Multimodal Computational Methods in Political Science

Summer Semester 2026

Contents

Learning objectives **2**
Part I — Why analyze political text computationally? **2**
The motivation 2
The four principles (Grimmer & Stewart 2013) 3
Part II — The preprocessing pipeline **4**
The big picture 4
Step 1: Tokenization 4
Step 2: Lowercasing and normalization 5
Step 3: Stop word removal 6
Step 4: Stemming and lemmatization 6
A note on n-grams 7
Part III — From text to numbers **7**
Bag of words 7
The Document-Term Matrix 8
TF-IDF: weighting by distinctiveness 8
Preprocessing as a modeling decision 9
Part IV — Tools and the practical session **10**
The R toolkit 10
The practical session: walkthrough 10
Key takeaways **11**
Required reading **11**
Recommended reading **11**
Looking ahead **11**

Lecture 1 — Text as Data

Foundations & Preprocessing

Methods Seminar: Multimodal Computational Methods in Political Science, LMU Munich

Summer Semester 2026

1 Learning objectives

By the end of this lecture, you should be able to:

1. Articulate why computational text analysis matters for political science research, and recognize its limitations.
2. Identify the major sources of political text and the metadata that comes with each.
3. Apply the standard preprocessing pipeline (tokenization, normalization, stop word removal, stemming/lemmatization) to a corpus of political text, and justify each choice you make.
4. Convert a corpus into a Document-Term Matrix and weight it with TF-IDF.
5. Use `quanteda` in R to perform all of the above on a real political corpus.
6. Recognize that preprocessing is a *modeling decision*, not a neutral preparation step.

2 Part I — Why analyze political text computationally?

2.1 The motivation

Politics is conducted through language. Almost everything political scientists care about is expressed in text: speeches in parliament, party manifestos, court rulings, media coverage, social media debate, diplomatic communiqués, legislative bills, interest group lobbying documents, citizens' open-ended survey responses. If you want to study political behavior at the elite level, much of what you have access to is text. If you want to study public opinion in real time rather than waiting for the next survey wave, text from social media (or other sources) is often your only option.

For most of the discipline's history, this text was analyzed by hand. The Manifesto Project (formerly the Comparative Manifestos Project) is a famous example: starting in 1979, trained human coders read every sentence of every party's electoral manifesto in dozens of countries and assigned it to one of 56 policy categories. The result is a dataset that has supported hundreds of papers on party positions and electoral competition. But the work involved is staggering: tens of thousands of coder-hours to build and maintain. Inter-coder reliability — the extent to which two different coders assign the same sentence to the same category — is often only moderate (Cohen's κ around 0.6–0.7 in many topic-coding tasks). And manual coding cannot keep up with the modern data deluge:

there are millions of tweets per day, hundreds of newspaper articles per week per country, thousands of legislative speeches per year per chamber. Hand-coding simply doesn't scale.

Computational text analysis offers a different bargain. You sacrifice some of the nuance and contextual judgment that a human coder brings, but you gain four things in return:

Scale. A trained logistic regression classifier can label a million documents in minutes. A topic model can surface latent themes in a corpus of 100,000 speeches overnight. Whatever you can do for one document, you can usually do for a million at marginal cost.

Consistency. A computer applies the same rule to every document. It does not get tired in the afternoon, change its mind about borderline cases, or interpret the codebook differently than its colleague. Whatever bias the algorithm has, it applies that bias uniformly — which is actually a feature for measurement, not a bug.

Discovery. Some patterns are invisible to close reading because they involve aggregating signals across thousands of documents. Topic models, for instance, can reveal that two seemingly unrelated themes systematically co-occur — something no individual reader would notice from reading one document at a time.

Replicability. A preprocessing script and a model fit are fully reproducible. Two researchers running the same code on the same data will get exactly the same results. This is in stark contrast to manual content analysis, where replication requires retraining a new team of coders.

It is important to be clear about what this approach is *not*. Computational text analysis does not replace careful reading. It does not eliminate the need for substantive knowledge — quite the opposite, you need substantive knowledge to design the analysis sensibly and to interpret the results. And it does not produce findings that are automatically more "objective" than human coding. The choices you make at every step (what to preprocess, what model to use, what to validate against) embed your assumptions about the data. The advantage is that those assumptions are now explicit and reproducible.

2.2 The four principles (Grimmer & Stewart 2013)

The most influential statement of how to think about computational text analysis in political science is Grimmer and Stewart's 2013 article *Text as Data: The Promise and Pitfalls of Automatic Content Analysis Methods for Political Texts*. It is short, well-written, and required reading for this course. The article lays out four principles that have become foundational:

Principle 1: All quantitative models of language are wrong — but some are useful. This is George Box's famous aphorism applied to text. No model captures the full nuance of human language. A bag-of-words model literally throws away word order and treats "the dog bit the man" as identical to "the man bit the dog." This is obviously absurd — yet bag-of-words methods can still distinguish between the manifestos of left-wing and right-wing parties with high accuracy, predict election outcomes from political speeches, and surface meaningful policy topics in legislative debates. The goal is not to model language perfectly. The goal is to find a useful approximation that helps you answer your research question.

Principle 2: Quantitative methods for text augment humans, they do not replace them. Automated tools help you read more, not stop reading. You still need to read documents to understand the data, validate the results, and interpret findings substantively. A topic model that produces 50 topics will not tell you what those topics mean — you have to read the top

words and the most representative documents and apply your substantive knowledge to label them. Throughout this course, you will see that computational methods are tools that extend your reach as a researcher; they do not substitute for being a researcher.

Principle 3: There is no universally best method for text analysis. Different methods are good for different problems. Sentiment analysis with logistic regression is great for some tasks and terrible for others. Topic models are powerful for discovery but unreliable for precise measurement. BERT fine-tuning produces state-of-the-art classification performance but requires more data than simpler methods. The right tool depends on your research question, your corpus, and your theoretical commitments. This course gives you a toolkit and trains your judgment about when to use each tool.

Principle 4: Validate, validate, validate. Always check that the computational output corresponds to something meaningful in the world. Compare your classifier's predictions against a hand-coded sample. Read the documents that your topic model assigns to each topic. Check that your sentiment scores correlate with known events. Computational methods produce numbers; numbers feel objective; but those numbers are only as good as the validation you put behind them. We will return to this principle in every lecture of the course.

3 Part II — The preprocessing pipeline

3.1 The big picture

Before you can apply any quantitative method to text, you need to convert raw text strings into a structured numerical representation. This conversion is called *preprocessing*, and it always involves a sequence of transformations. The standard pipeline looks like this:

> Raw text → Tokenization → Normalization → Stop word removal → Stemming/Lemmatization → Numerical representation

Each step makes specific design choices, and each of those choices affects your downstream results. There is no neutral, default pipeline. The combination of choices that works best for sentiment analysis on tweets is different from what works best for topic modeling of legislative speeches. The single most important habit to develop is: **document every preprocessing decision and, ideally, test whether your conclusions are robust to alternative choices**. This is called *sensitivity analysis*, and it should be a routine part of any text analysis project.

3.2 Step 1: Tokenization

Tokenization splits a continuous string of text into individual units called tokens. In the simplest case, tokens are words, and tokenization means splitting on whitespace and punctuation. But in practice, several decisions need to be made.

Consider the sentence: *"The Prime Minister doesn't believe in tackling the climate emergency."*

A naive tokenizer would produce: `["The", "Prime", "Minister", "does", "n't", "believe", "in", "tackling", "the", "climate", "emergency", "."]`

Notice three things. First, the contraction "doesn't" has been split into "does" and "n't" — this is the spaCy default. You could also keep "doesn't" whole, or expand it to "does not." This choice

matters more than it seems: if you split "doesn't" into "does" and "n't" and then later remove stop words, the negation "n't" will be silently deleted, and your model will see only "does believe" — exactly the opposite of the intended meaning. This is one of the classic sources of bugs in sentiment analysis pipelines.

Second, the period at the end is treated as its own token. Whether you keep it depends on what you plan to do later. For bag-of-words models, punctuation is just noise. For sentence-level analysis or for structured tasks like named entity recognition, you need to preserve sentence boundaries.

Third, multi-word expressions like "Prime Minister" are split into two separate tokens. This loses information: "Prime Minister" is a specific concept, not just the conjunction of "prime" and "minister." We will partially address this when we discuss n-grams later in the lecture.

A few other decisions that come up frequently in political text:

- **Numbers and references.** "Section 230" is the name of a specific U.S. law about platform liability — keeping the number is meaningful. "1973" might refer to a specific event. On the other hand, parliamentary speeches are full of incidental numbers (line counts, page references) that are pure noise. There is no general answer; it depends on your corpus and your task.
- **Hyphenated words.** "Right-wing" — one token or two? "COVID-19" — usually one. Different tokenizers handle these differently.
- **URLs, hashtags, @-mentions.** Essential to handle for social media data, irrelevant for parliamentary text. We will see how to strip them shortly.

3.3 Step 2: Lowercasing and normalization

Normalization is the umbrella term for everything that reduces superficial variation in the text so that tokens with the same intended meaning map to the same string.

The most common normalization step is **lowercasing**: converting all text to lowercase. This merges "Brexit" with "brexit" (the former when capitalized at the start of a sentence, the latter when used in the middle), and "The" with "the". For most tasks this is helpful — it cuts the vocabulary roughly in half and removes irrelevant variation.

But lowercasing has a real cost. It destroys information that capitalization carries. In English, capitalized words at the start of a sentence are not informative, but capitalization in the middle of a sentence often is: it usually marks a proper noun (a person, place, or organization). Lowercasing collapses "Bush" the U.S. president and "bush" the plant. It collapses "May" the prime minister and "may" the modal verb. It also destroys the visual distinctiveness of acronyms — NHS, EU, UN, MP, GOP — which are often very important in political texts. If you are doing named entity recognition, you should usually *not* lowercase, because case is one of the strongest signals that a word is a name. For most other tasks, lowercasing is fine.

Other normalization steps include **removing punctuation** (commas, periods, quotation marks become invisible — useful for bag-of-words but loses sentence structure), **removing numbers** (good when numbers are noise, bad when years and legal references are meaningful), and **removing URLs, hashtags, and @-mentions** (essential for social media, rarely needed for parliamentary text). There is also **Unicode normalization**, which handles cases where the same visible character can be represented multiple ways in Unicode — relevant for multilingual corpora and for texts that have been copied from PDFs.

The unifying principle is: each normalization step is a trade-off between reducing vocabulary size

(good — makes everything faster and reduces sparsity) and preserving information (also good — but in tension with the first goal). You have to think about which information you actually need for your specific task.

3.4 Step 3: Stop word removal

Stop words are high-frequency, low-information words like articles, prepositions, and auxiliary verbs. In English, these include "the", "of", "and", "a", "to", "in", "is", "it", "that", and so on. The intuition is that these words appear in every document and therefore carry no information about what any specific document is about — so we might as well remove them, reducing vocabulary size and noise.

Standard stop word lists exist for most languages. In `quanteda`, calling `stopwords("en")` gives you a list of about 175 English stop words drawn from the Snowball project. Other lists exist (the NLTK list, the spaCy list, the scikit-learn list), and they differ from each other in small but sometimes consequential ways. There is no canonical English stop word list.

Stop word removal is the preprocessing step where you most often need to think carefully about the costs. The most common pitfall is **negation**: "not" is on virtually every English stop word list. But "not acceptable" and "acceptable" mean opposite things, and removing "not" silently flips the sentiment. The same is true for "no", "nor", and arguably "against". For sentiment analysis or stance detection, you should remove "not" from your stop word list — that is, use a *custom* stop word list that excludes negations.

A second pitfall is that stop words can encode style and identity even when they don't carry topical content. Authorship attribution, for example, often relies heavily on the frequencies of function words because individual authors have characteristic patterns of using "of" vs. "from", or "while" vs. "whilst". If you remove stop words, you destroy this signal. For most political science classification tasks this doesn't matter, but it is worth knowing about.

The general rule is: stop word removal is a default that is appropriate for most tasks, but you should always think about whether the words you are removing matter for your specific question. Inspect what gets removed. If anything looks suspicious, exclude it from your stop word list.

3.5 Step 4: Stemming and lemmatization

Both stemming and lemmatization aim to reduce inflected word forms to a common base, so that "govern", "governs", "governed", and "governing" all collapse into a single token. The reason to do this is to reduce vocabulary size and to ensure that semantically related words are not treated as completely separate features. Without this step, a logistic regression classifier learning that "govern" predicts a particular class would not automatically learn the same thing about "governing."

The two approaches differ in *how* they perform this reduction.

Stemming is rule-based. It applies a fixed set of heuristics that chop off common suffixes. The most famous English stemmer is the Porter stemmer (and its successor, the Snowball stemmer). Stemming is fast and requires no dictionary or linguistic resources. But it has two problems. First, it produces non-words: "political" stems to "polit", which is not a real word. This makes the output harder to read and less useful for downstream analysis that involves human inspection. Second, and more seriously, it makes errors. "University" and "universal" both stem to "univers" — they are unrelated concepts but get merged. This is called **overstemming**. Conversely, "ran" and "run" do

not get merged because the Porter stemmer's rules don't handle irregular verbs — this is called **understemming**.

Lemmatization is dictionary-based. It looks up each word in a morphological dictionary to find its canonical form, called the *lemma*. So "govern", "governs", "governed", and "governing" all map to "govern" (the dictionary form of the verb), but "government" stays as "government" (because it's a noun, distinct from the verb). "University" stays as "university" and "universal" stays as "universal" — they don't get merged. The output is always a real word, and the linguistic distinctions are preserved.

The cost of lemmatization is computational and infrastructural. It is slower than stemming, and it requires a trained language model — typically `spacyr` (an R wrapper for spaCy) or `udpipe`. Both have excellent pre-trained English models and support other major languages.

For serious research, lemmatization is generally preferred. For exploratory work and for topic models (where the goal is to surface broad themes rather than precise distinctions), stemming is often acceptable and is much faster. The Porter stemmer is built into `quanteda` via `tokens_wordstem(language = "en")`.

3.6 A note on n-grams

So far, every step has assumed that each token is a single word. But many meaningful expressions span multiple words: "climate change", "minimum wage", "Prime Minister", "European Union", "border control". Bag-of-words treats these as separate tokens and loses the connection between them.

The standard fix is to include **n-grams** — sequences of n consecutive tokens. A *unigram* is a single word ("climate"). A *bigram* is two consecutive words ("climate change"). A *trigram* is three. In practice, including bigrams in addition to unigrams often improves classification accuracy noticeably, because politically meaningful concepts are often two-word phrases.

The cost of n-grams is vocabulary explosion. If you have 50,000 unigrams, you could have up to 2.5 billion possible bigrams. In practice most of those never occur, but the count still grows dramatically. The practical solution is to keep only frequent bigrams (e.g., bigrams that appear at least 10 times in the corpus) or to use linguistically motivated phrase detection to extract only the bigrams that are statistically meaningful. `quanteda` supports both approaches.

4 Part III — From text to numbers

4.1 Bag of words

The simplest way to convert preprocessed text into numbers is the **bag of words** representation: each document is represented as a vector of counts, where each entry tells you how many times a particular word appears in that document.

Take a preprocessed sentence like "climate action government tackle climate emergency." The bag-of-words vector for this document, restricted to the words that actually appear, looks like: `[climate: 2, action: 1, government: 1, tackle: 1, emergency: 1]`. Word order is lost completely — we only know which words appeared and how often.

This loss of word order is the fundamental limitation of bag-of-words. The famous example is that "the dog bit the man" and "the man bit the dog" produce identical vectors. Sentences with completely different meanings can be indistinguishable. And yet — and this is the surprise — bag-of-words representations work remarkably well for many tasks. They can distinguish manifestos by party with high accuracy, classify the topic of news articles, scale legislators on ideological dimensions, and detect sentiment in product reviews. The reason is that for many tasks, the *content words* a document uses are a strong enough signal that ignoring order doesn't hurt much. Word order matters for some tasks (especially anything involving negation or fine-grained reasoning), but for many practical purposes the bag-of-words shortcut is a good approximation.

4.2 The Document-Term Matrix

When you have many documents, you stack their bag-of-words vectors into a single matrix. Rows are documents, columns are words from the corpus vocabulary, and each cell contains the count of that word in that document. This is the **Document-Term Matrix** (DTM) — sometimes called the Document-Feature Matrix (DFM) in the `quanteda` ecosystem.

Formally: if you have m documents and a vocabulary of size |V|, the DTM is an m × |V| matrix of non-negative integers. The DTM is the fundamental data structure that almost every classical text analysis method takes as input.

The DTM has one important property: it is **sparse**. A typical political corpus has tens of thousands of distinct words, but any single document uses only a few hundred. This means the vast majority of cells in the DTM are zeros. For a corpus of 10,000 speeches with a vocabulary of 50,000 words, you have 500 million cells, of which probably 99.5% are zero. This sparsity creates both computational and statistical challenges. Computationally, you need sparse matrix data structures (which `quanteda` uses internally) — storing all those zeros explicitly would waste enormous amounts of memory. Statistically, most features are observed in only a handful of documents, which makes them noisy and unreliable.

Several practical strategies help manage the size of the DTM. The most common is **feature trimming**: removing very rare words (those appearing in fewer than, say, 5 documents — they are usually typos, names, or words too rare to generalize from) and very frequent words (those appearing in more than 95% of documents — they cannot distinguish documents from each other). You can also use **feature selection** to keep only the top k most informative features, ranked by some criterion like variance or chi-squared association with the outcome. For most political science tasks, trimming the DTM down to about 5,000–15,000 features is a good starting point.

4.3 TF-IDF: weighting by distinctiveness

Raw word counts have a weakness: they treat all words as equally informative. A word like "the" appears in every document, but it tells you nothing about what any specific document is about. A word like "Brexit" appears in only some documents, and when it does, it tells you a lot. Raw counts don't distinguish between these two situations.

TF-IDF (Term Frequency × Inverse Document Frequency) is a weighting scheme that fixes this. The idea is to weight each word by how *distinctive* it is — that is, how frequent it is in this particular document combined with how rare it is across the corpus as a whole.

The two components are:

Term Frequency (TF): how often a word appears in a specific document. Several variants exist. The simplest is the raw count. A common normalized version is `TF(t, d) = count(t in d) / |d|` — the count divided by the document length. Another is `log(1 + count)`, which dampens the effect of very high counts. The choice rarely matters much for downstream results.

Inverse Document Frequency (IDF): how rare a word is across the entire corpus. The standard formula is `IDF(t) = log(N / df(t))`, where N is the total number of documents and df(t) is the *document frequency* — the number of documents containing the word at least once. A word that appears in every document has IDF = log(1) = 0. A word that appears in only 1 of 1,000 documents has IDF = log(1000) ≈ 6.9. The logarithm dampens the differences so that very rare words don't dominate.

The TF-IDF weight is the product: `TF-IDF(t, d) = TF(t, d) $\times$ IDF(t)`.

A worked example: imagine a corpus of 1,000 House of Commons speeches. Consider the word "Brexit" in a particular speech by a Conservative MP. The speech has 500 words, and "Brexit" appears 8 times. So TF = 8/500 = 0.016. The word "Brexit" appears in 50 of the 1,000 speeches in the corpus. So IDF = log(1000/50) = log(20) ≈ 1.301. The TF-IDF weight is therefore 0.016 × 1.301 ≈ 0.0208.

Now compare with the word "the" in the same speech. "the" appears 25 times, so TF = 0.05 (much higher than Brexit). But "the" appears in all 1,000 speeches, so IDF = log(1000/1000) = log(1) = 0. The TF-IDF weight is therefore 0.05 × 0 = 0. Even though "the" is much more frequent than "Brexit" in absolute terms, TF-IDF assigns it zero weight because it carries no information about what distinguishes this speech from others.

This is the magic of TF-IDF: it automatically down-weights common words even if you didn't remove them as stop words, and it amplifies the words that actually distinguish documents from each other. It is a heuristic — there are no parameters to tune and no probabilistic foundation — but it works extraordinarily well in practice and is a standard component of most classical text analysis pipelines.

4.4 Preprocessing as a modeling decision

I want to end this section by emphasizing the most important conceptual point of the lecture. Every preprocessing choice is a modeling decision that embeds assumptions about what matters for your research question. Removing stop words assumes that function words don't matter. Stemming assumes that morphological variants should be treated the same. Lowercasing assumes that case is not informative. Setting a minimum document frequency assumes that rare words are noise. None of these assumptions is always right.

The discipline-wide best practice is twofold. First, **document every preprocessing choice** in your code and your paper, so that others can understand and replicate what you did. Second, **test sensitivity**: rerun your main analysis with one or two alternative preprocessing pipelines and report whether the substantive conclusions change. If they don't, your findings are robust. If they do, that is informative — it means your conclusions depend on assumptions that you should be transparent about.

5 Part IV — Tools and the practical session

5.1 The R toolkit

This course will use R for the first four lectures and switch to Python for the last three (where deep learning libraries are dominant). For text analysis in R, the main package is `quanteda`. It is a comprehensive framework for text processing — corpora, tokenization, document-feature matrices, weighting, n-grams, keyness statistics, and integration with downstream analysis. Think of it as the `tidyverse` of text analysis: opinionated, well-designed, fast, and well-documented.

Other packages you will encounter:

- **`quanteda.textstats`** — statistical operations on text: frequency tables, keyness, readability, lexical diversity.
- **`quanteda.textplots`** — plotting functions: word clouds, comparison plots, network plots.
- **`quanteda.textmodels`** — classical text classification models including Naive Bayes and Wordfish (we'll use this in Lecture 2).
- **`tidytext`** — an alternative tidy-data approach to text mining. Integrates seamlessly with `dplyr` and `ggplot2`. Good for exploration.
- **`readtext`** — for importing text from PDF, Word, JSON, CSV, and URLs.
- **`spacyr`** — R interface to the Python spaCy library, used for lemmatization, named entity recognition, and part-of-speech tagging. Best-in-class for English and other major languages.
- **`udpipe`** — pure R alternative to spaCy with similar functionality, supporting 60+ languages.

For the practical session of this lecture, we will only need `quanteda`, `quanteda.textstats`, `quanteda.textplots`, and the `tidyverse`.

5.2 The practical session: walkthrough

For the practical session, students should download the **ParlSpeech V2** dataset (Rauh & Schwalbach, available on Harvard Dataverse) and use the U.K. House of Commons subset, pre-filtered to a single parliament — say, 2017–2019 — to keep the corpus to about 5,000–10,000 speeches for in-class work. The relevant columns are `text`, `speaker`, `party`, `date`, and `agenda`.

The walkthrough proceeds through five steps. First, load the CSV and inspect it to understand the structure. Second, build a `quanteda` corpus and tokenize: remove punctuation, numbers, and URLs; remove English stop words; lowercase. This is also a good moment to reflect what gets lost in each of these steps. Third, build the Document-Feature Matrix from the tokenized corpus and trim rare words to keep the matrix manageable. Fourth, group the DFM by party and inspect the top features per group — we can see the most frequent words for Labour vs. Conservative vs. Green and recognize their substantive priorities. Fifth, compute TF-IDF and use the `textstat_keyness` function to compute statistically significant party-distinctive vocabulary, then visualize with `textplot_keyness` and `ggplot2`.

The discussion questions for the practical session should focus on the three things:

1. **What did stop word removal do to your data?** Inspect the output and find a case where removing a stop word probably changed the meaning.
2. **Are the top words by party what you expected?** When they aren't — and they often aren't — this is informative. It might mean your preprocessing is too aggressive, your corpus is unbalanced, or your priors are wrong.

3. **What happens if you don't remove stop words at all? Or don't lowercase? Or don't trim rare words?** This is the sensitivity analysis discussion and it's worth doing for at least one alternative pipeline.

6 Key takeaways

1. **Text is the most abundant source of political data**, and the only way to use it at scale is to convert it into numerical representations through a preprocessing pipeline.
2. **Preprocessing is not neutral.** Every step (stop words, stemming, lowercasing, n-grams, frequency thresholds) encodes assumptions about what matters. Document and justify your choices, and test sensitivity to alternative pipelines.
3. **The Document-Term Matrix is the bridge from text to quantitative analysis.** It is sparse, high-dimensional, and the foundation for almost everything we will do in Lectures 2–4.
4. **TF-IDF improves on raw counts** by automatically weighting words by distinctiveness. Common words get pushed toward zero; rare and document-specific words get amplified.
5. **Validation matters.** Always check that your computational outputs correspond to something meaningful. Read documents, inspect the words being removed, look at the top features. Do not trust numbers without checking what they mean.

7 Required reading

- Grimmer, J., & Stewart, B. M. (2013). "Text as Data: The Promise and Pitfalls of Automatic Content Analysis Methods for Political Texts." *Political Analysis*, 21(3), 267–297.

8 Recommended reading

- Grimmer J., Roberts M. E. & Stewart B. M. (2022). Text as Data, Chapters 1 to 5.
- Welbers, K., Van Atteveldt, W., & Benoit, K. (2017). "Text Analysis in R." *Communication Methods and Measures*, 11(4), 245–265.
- The `quanteda` tutorials at https://tutorials.quanteda.io

9 Looking ahead

In **Lecture 2**, we will move from representing text to *classifying* it. Given the Document-Term Matrix you built today, we will train logistic regression and Naive Bayes classifiers to predict labels — for example, the party of a speech, the topic of a manifesto sentence, or the sentiment of a tweet. This is the first time the course moves from descriptive representation to *prediction*, and it is the most common use of text analysis in political science research.