From Words to Vectors: Text as Data for Economists

Lecture 4

Elodie Chervin Daniel Barbosa

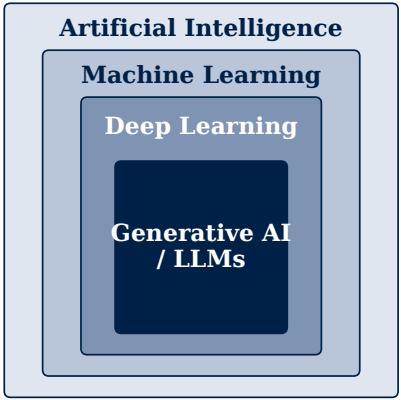
HT 2026

University of Oxford & Nuffield College

From AI to LLMs: what we are dealing with

Where this lecture sits: AI, ML, deep learning, LLMs

- **Artificial intelligence (AI)** — machines that perform tasks needing human-like intelligence: language, pattern recognition, decisions. Includes rule-based and symbolic methods, not only learning.
- **Machine learning (ML)** — algorithms that *learn* from data instead of being hand-coded (a spam filter trained on labelled emails, not hand-written rules).
- **Deep learning** — ML with multi-layer neural networks; the model learns its own features.
- **Generative AI** — systems that generate new content: text, code, images, audio, video.
- **LLMs** — one important GenAI family: deep networks trained at scale to predict the next token in text.



An LLM is a deep neural network aimed at next-token prediction, applied at very large scale.

What is a large language model?

Large language model (LLM): a deep neural network trained on massive text corpora to predict the next token in a sequence; understanding, generation, and conversation emerge from doing that prediction task at scale.

- "**Large**" is two things at once: model *size* (tens to hundreds of billions of parameters) and training *data* (large portions of public text).
- The training objective — predict the next token — is almost embarrassingly simple, yet at scale it yields models that translate, summarise, and answer questions.
- Modern LLMs are built on the **transformer architecture** (Vaswani et al. 2017): attention lets the model understand the context in which a token appears and how that token relates to the wider document. We defer the mechanics to Week 5.
- Because they *generate* text, LLMs are a form of **generative AI** (GenAI).

Same supervised template, $Y = f(X)$ — the label Y is just the next word.

What LLMs are used for (broadly)

Anything that involves parsing or producing text:

- machine translation between languages;
- summarisation of long documents;
- sentiment analysis and text classification;
- question answering and knowledge retrieval (medicine, law, finance);
- content and computer-code generation;
- chatbots and virtual assistants that augment search.

Example. Prompt: *"Summarise why inflation rose in two sentences."*
Reply: *"Energy prices increased production and transport costs. Strong demand allowed firms to pass those costs into consumer prices."*

Emergent behaviour: a model trained *only* to predict the next word turns out to translate, classify, and reason — abilities never explicitly taught. This breadth is what separates LLMs from the older, one-task-per-model NLP systems.

LLMs in economics — research tools & simulated agents

A fast-growing literature puts LLMs to work *inside* economics:

- **Korinek (2023)** — a hands-on taxonomy of how LLMs automate parts of the research workflow (ideation, coding, writing, data work).
- **Horton (2023)** — "homo silicus": treat an LLM as a simulated economic agent, endow it with preferences, and replay classic behavioural experiments.
- **Argyle et al. (2023)** — LLMs conditioned on demographics reproduce the response patterns of human subgroups ("silicon sampling").
- **Aher et al. (2023)** — LLMs replicate well-known social-science experiments across many simulated participants.

Two distinct uses: LLMs as research *assistants*, and LLMs as simulated *subjects*.

LLMs in economics — economic impacts

And a parallel literature studies their economic *impact*:

- **Eloundou et al. (2024)**, "GPTs are GPTs" — estimates that LLMs expose a large share of US jobs to task-level automation; AI as a general-purpose technology.
- **Brynjolfsson, Li & Raymond (2023)** — a generative-AI assistant raises call-centre productivity, most for less-experienced workers.
- **Noy & Zhang (2023)** — randomised writing tasks: ChatGPT raises output and compresses the skill gap.
- **Dell'Acqua et al. (2023)** — the "jagged frontier": big gains inside the model's competence, costly errors just outside it.

Three flavours of modern LLM systems

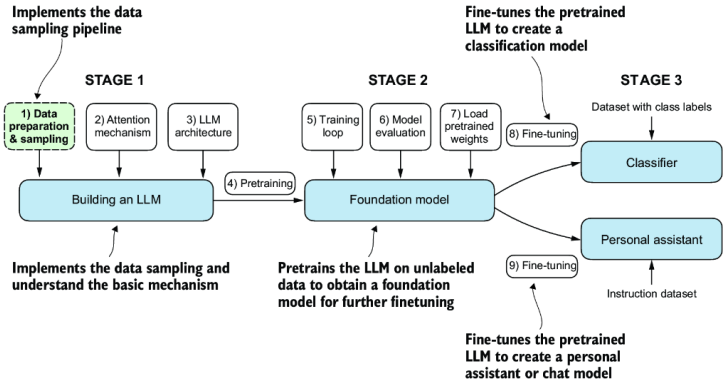
1. Basic LLMs. Prompt in → answer out, usually in one pass.
Example: "Draft a two-paragraph summary of this FOMC statement."

2. Reasoning models. Spend extra computation on intermediate reasoning before answering.
Example: "Compare two identification strategies; list assumptions, threats, and a final recommendation."

3. Agentic models. Reason *and* act: call tools, run code, retrieve documents, iterate after observing results.
Example: "Download 10-Ks, build a dictionary score, run the regression, and return the table."

All three sit on top of the same next-token engine; reasoning and agency are scaffolding around it.

How they are built: pretraining → foundation → finetuning



- **Stage 1 — pretraining.** Train on a huge corpus of *raw, unlabelled* text by next-word prediction. This is *self-supervised*: the label is the next word, so labels come "for free".
- Result: a **foundation** (base) model, e.g. GPT-3.
- **Stage 2 — finetuning.** Continue training on a smaller *labelled* set: instruction-following or classification.

What they train on: a large mix of text

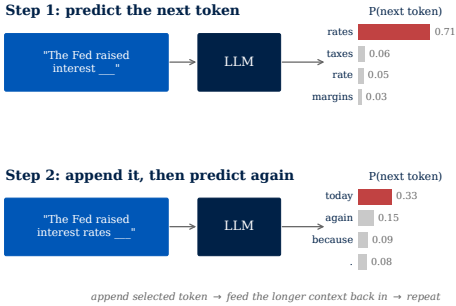
Pretraining data is a *combination of many textual datasets.* A representative recipe (GPT-3-scale, illustrative shares):

Source	What it is	Share
Web crawl (filtered)	general internet text	≈ 60%
Curated web text	high-quality links	≈ 22%
Books	internet book corpora	≈ 16%
Wikipedia	encyclopaedic text	≈ 3%

- A **token** is the unit a model reads — roughly a word or a piece of punctuation. Corpora run to *hundreds of billions* of tokens.
- Scale is not free: GPT-3 pretraining cost millions of dollars, and *scaling laws* (Kaplan et al. 2020; Hoffmann et al. 2022) relate performance to data, parameters, and compute.

Next-token prediction, concretely

- Feed the model a context: `"The Fed raised interest ___"`.
- It outputs a *probability distribution* over the whole vocabulary for the next token.
- Pick one (arg-max or sample), append it, and feed the longer string back in — this is what **autoregressive** means.
- Repeat to generate text one token at a time.



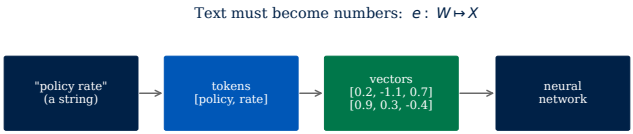
That single objective — guess the next token — is the entire training signal behind modern LLMs.

LLMs in one sentence

Key Concept

At a basic level, an LLM is a **deep neural network** trained on a **large corpus** — a combination of many textual datasets — to perform **next-token prediction**. Chat, reasoning, and agents are all built on top of that one objective.

But neural networks eat numbers, not words



- A neural network consumes real-valued *vectors* and is trained by SGD with back-propagation.
- Text is *categorical* — a string of symbols. You cannot multiply "policy rate" by a weight matrix.
- So every NLP system, LLMs included, must first map text → numbers: an encoder $e : W \mapsto X$.
- The quality of that map determines everything downstream.

The encoder: $e : W \mapsto X$ — the map from a string to a vector the network can process. *This* is what the rest of the lecture builds.

More on large-scale neural-network training, optimisation, and transformer training in Week 5. Up next: what that text looks like, and three increasingly powerful ways to build the encoder.

Working with Text Data

Why text?

Most empirically interesting data in social science is not in a spreadsheet:

- central-bank speeches, FOMC transcripts, BoE Inflation Reports;
- 10-K and 10-Q filings, earnings-call transcripts;
- news articles, opinion columns, social-media posts;
- court rulings, parliamentary debates, party manifestos;
- product descriptions, customer reviews, search queries.

Gaillac & L'Hour (2025): "A multitude of relevant economic data is only available in the form of strings of characters: newspaper articles, central bank speeches, stock analyst reports, political statements, social network comments, marketing messages…"

Three eras of NLP for economists

- **NLP 1.0 — counts.**
  - Pre-process the corpus, build a document-term matrix, weight terms by their frequency using TF-IDF.
  - Interpretable and useful for regressions/topic models; ignores word order.
  - *Gaillac & L'Hour Ch. 12.*
- **NLP 2.0 — static word embeddings.**
  - word2vec, GloVe, fastText: one dense vector per word, learned from local co-occurrence.
  - Captures semantic similarity; the token *bank* has the same vector in every sentence.
  - *Gaillac & L'Hour Ch. 13.*
- **NLP 3.0 — contextual embeddings.**
  - ELMo, BERT, GPT: the vector for a token depends on the surrounding sentence/document.
  - The token *bank* receives different vectors in "river bank" and "central bank".
  - *Gaillac & L'Hour Ch. 14.*

NLP 1.0 — Representing text as numbers

What is text, really?

At the lowest level, a document is just a string of characters:

- Latin letters, possibly decorated: `cafe`, `café`, `naïve`.
- Non-Latin scripts: Greek letters such as $\alpha\beta\gamma$; Arabic; Chinese; emoji such as `:)`.
- Punctuation, whitespace, line breaks, tabs: `don't`, `New-York`, `New York`.
- Digits, currency symbols, URLs, hashtags: `$4.25`, `10-K`, `https://sec.gov`, `#inflation`.

The empirical question: How do we obtain an *informative, quantitative* representation of strings of characters so we can use them in regressions, classifiers, or causal-inference pipelines?

Step one of any NLP 1.0 pipeline is pre-processing: turn the raw string into a list of comparable units we call *tokens*.

NLP 1.0 — Representing text as numbers

Notation for Textual Data

Notation

Vocabulary: Set Ω of V distinct tokens, indexed $v \in \{1, \dots, V\}$.

Document: An ordered list of N_d tokens, $\mathbf{w}_d = (w_{d,1}, \dots, w_{d,N_d})$, with $w_{d,n} \in \{1, \dots, V\}$.

Corpus: A collection of D documents $\mathbf{w} = (\mathbf{w}_1, \dots, \mathbf{w}_D)$, total length $N \equiv \sum_d N_d$.

The basic statistic for each document is the per-token count

$$x_{d,v} = \sum_{n=1}^{N_d} \mathbf{1}\{w_{d,n} = v\}, \qquad v = 1, \dots, V.$$

Stacking the rows gives the *document-term matrix* $\mathbf{X} \in \mathbb{N}^{D \times V}$.

Following Hansen, "Unstructured Data" Lecture 1, slide 31; Gaillac & L'Hour 12.1.

A toy corpus

Three documents, $D = 3$:

d_1: "stephen is nice"
d_2: "john is also nice"
d_3: "george is mean"

Unique vocabulary $\Omega = \{\text{stephen, is, nice, john, also, george, mean}\}$, $V = 7$, indexed:

	stephen	is	nice	john	also	george	mean
index	1	2	3	4	5	6	7

Indexed documents: $\mathbf{w}_1 = (1,2,3)$, $\mathbf{w}_2 = (4,2,5,3)$, $\mathbf{w}_3 = (6,2,7)$.

We will build a DTM and then TF-IDF on this corpus by hand later in the lecture.

The document-term matrix (DTM)

Document-term matrix: For corpus of D documents and vocabulary of size V, the matrix $\mathbf{X} \in \mathbb{N}^{D \times V}$ with $X_{d,v} = \#$ occurrences of token v in document d.

For the toy corpus $\{d_1, d_2, d_3\}$ (stephen / john / george):

$$\mathbf{X} = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \quad \text{(rows: docs; columns: vocabulary).}$$

- **Sparsity.** A typical FOMC statement contains ~ 30 tokens; the FOMC vocabulary has $V \approx 8{,}615$. So $\geq 99.6\%$ of every row is zero.
- **Bag-of-words.** The DTM throws away order: *dog bites man* and *man bites dog* have the same row.
- Storage: use a sparse-matrix format (`scipy.sparse`); never instantiate as dense.

NLP 1.0 — Representing text as numbers

Pre-processing

Pre-processing: turning a string into countable units

Four standard steps in sequence, each a researcher decision:

Step	What it does	Sample output
(raw input)	—	`The Committee's decision raised rates today.`
Normalisation	UTF-8, lowercase, strip punctuation/accents.	`the committee s decision raised rates today`
Tokenisation	Split on whitespace / punctuation; choose unit.	`[the, committee, s, decision, raised, rates, today]`
Stop-word removal	Drop function words (`the, of, and, ...`).	`[committee, decision, raised, rates, today]`
Stemming / lemmatisation	Map inflected forms to a single root.	`[committee, decision, raise, rate, today]`

Two corpora pre-processed differently are not comparable. Always report your pipeline (Denny & Spirling 2018: preprocessing can flip empirical conclusions).

Tokenisation choices

Example sentence: `The Committee's decision raised rates today.`

Choice	Token sequence
Word-level	`[The, Committee, 's, decision, raised, rates, today]`
Character-level	`[T, h, e, _, C, o, m, m, i, t, t, e, e, ...]`
Sub-word / BPE	`[The, Committee, 's, decision, raised, rates, today]`

- Word-level is readable but fragile to rare or misspelled words.
- Character-level handles anything, but sequences become long.
- Sub-word tokenisation is the modern transformer default: it keeps common words intact and splits rare words into reusable pieces.

Tokenisation: The first irreversible step. A tokeniser is defined by its vocabulary; two corpora pre-processed differently are not directly comparable.

How similar are documents?

Documents as vectors

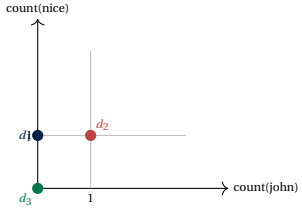
Each row $\mathbf{x}_d \in \mathbb{R}^V$ of the DTM is a vector. Vocabulary terms are the *coordinate axes*: $e_v \in \mathbb{R}^V$ has a 1 in position v and 0 elsewhere, and

$$\mathbf{x}_d \;=\; x_{d,1}\, e_1 + x_{d,2}\, e_2 + \cdots + x_{d,V}\, e_V .$$

For the toy document "john is also nice",

$$\mathbf{x}_2 = (0,1,1,1,1,0,0) = e_{\text{is}} + e_{\text{nice}} + e_{\text{john}} + e_{\text{also}}.$$

- Every pair of distinct terms is orthogonal: $e_v^\top e_{v'} = 0$ for $v \neq v'$.
- All terms are *equidistant*: $\|e_v - e_{v'}\|_2 = \sqrt{2}$ for any pair.
- Geometry of the DTM tells us about *documents*, not about *words* — terms have no notion of similarity yet.



Why not Euclidean distance?

Standard distance on $\mathbb{R}^V$:

$$d_{\text{Euc}}(\mathbf{x}_a, \mathbf{x}_b) = \sqrt{\sum_{v=1}^{V} (x_{a,v} - x_{b,v})^2}.$$

Counter-example. Suppose $\mathbf{x}_b = \alpha \mathbf{x}_a$ for some $\alpha > 0$ (same word *distribution*, just longer). Then

$$d_{\text{Euc}}(\mathbf{x}_a, \mathbf{x}_b) = |1 - \alpha| \, \|\mathbf{x}_a\|_2,$$

which can be arbitrarily large even though the two documents are stylistically identical.

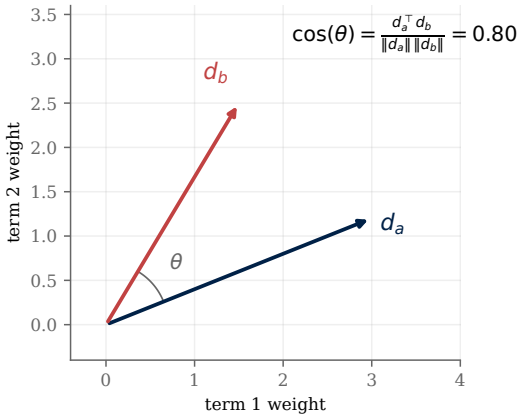
> Example: document A says *inflation risk* once, so $\mathbf{x}_a = (1, 1)$. Document B repeats the same phrase ten times, so $\mathbf{x}_b = (10, 10)$. They point in the same direction, but $d_{\text{Euc}}(\mathbf{x}_a, \mathbf{x}_b) = \sqrt{162}$.

Document length dominates Euclidean distance. For text we want a measure that is invariant to magnitude.

Cosine similarity

Cosine similarity: $\cos(\mathbf{x}_a, \mathbf{x}_b) = \dfrac{\mathbf{x}_a^\top \mathbf{x}_b}{\|\mathbf{x}_a\|_2 \, \|\mathbf{x}_b\|_2}$.

- Range: $[-1, 1]$ in general; $[0, 1]$ for non-negative vectors like DTM rows.
- Geometric: cosine of the angle between $\mathbf{x}_a$ and $\mathbf{x}_b$ in $\mathbb{R}^V$.
- Invariant to length: $\cos(\mathbf{x}_a, \alpha\,\mathbf{x}_a) = 1$ for every $\alpha > 0$.
- Equivalent to Euclidean distance on the unit sphere, since $\|\mathbf{x}_a/\|\mathbf{x}_a\| - \mathbf{x}_b/\|\mathbf{x}_b\|\|_2^2 = 2(1 - \cos)$.



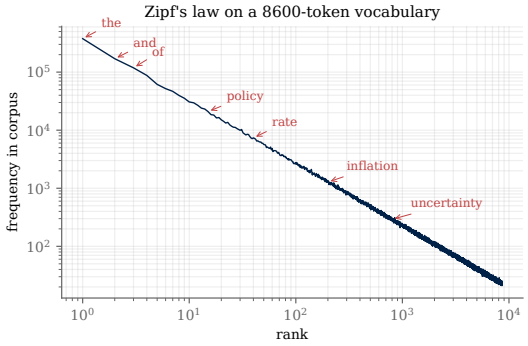
Standard distance in NLP 1.0. We will keep using it through embeddings (Section 4) and even through transformers.

Why should we care about word frequency?

Zipf's law

Zipf's law: In most natural-language corpora, the frequency of a token is approximately inversely proportional to its rank: $\text{freq}(r) \propto r^{-s}$ for some $s \approx 1$.

- Very few tokens dominate (function words: *the, of, and*).
- A long tail of low-frequency tokens carries most of the corpus-specific content.
- Raw counts are an awful feature: *the* appears 10x more than *inflation*, but the latter is what we care about.



Two responses: (i) rescale within document, (ii) reweight across documents. That is exactly TF-IDF.

Term frequency: damping the power law

Raw counts in a single document follow Zipf within that document too: a few tokens dominate. Dampen them by passing the count through a concave function:

$$tf_{d,v} = \begin{cases} 0, & x_{d,v} = 0; \\ 1 + \log(x_{d,v}), & x_{d,v} > 0. \end{cases}$$

- $x_{d,v} = 1 \Rightarrow tf_{d,v} = 1$;
- $x_{d,v} = 10 \Rightarrow tf_{d,v} \approx 3.3$;
- $x_{d,v} = 100 \Rightarrow tf_{d,v} \approx 5.6$.

Variant in common use. Normalise counts by document length: $tf_{d,v} = x_{d,v} / N_d$ (this is what `sklearn`'s `TfidfVectorizer` does by default). Both choices serve the same purpose: a token appearing twice as often should not carry twice as much weight.

Thought experiment: why we need IDF

Two-term dictionary $\mathcal{D} = \{v', v''\}$, two documents d' and d'' with

$$x_{d',v'} > x_{d'',v'} \quad \text{and} \quad x_{d',v''} < x_{d'',v''}.$$

Now suppose *no other document uses v' but every other document uses v''*.

Which document is "more about" the theme this dictionary captures?

Intuition: a token that appears *only* in d' is much more diagnostic of d''s content than a token that appears everywhere. Term frequency alone cannot tell us this — we need a corpus-level statistic.

Inverse document frequency

Let $df_v = \#\{d : x_{d,v} > 0\}$ be the number of documents containing token v.

Inverse document frequency: $$idf_v = \log \frac{D}{df_v}.$$

- Common tokens (e.g. *the*): $df_v \approx D$, so $idf_v \approx 0$.
- Rare tokens (e.g. *stagflation*): $df_v \ll D$, so idf_v is large.
- Logarithm dampens the effect: idf ranges over $[0, \log D]$, not $[1, D]$.

Smoothed variant in implementation: $idf_v = \log \frac{1+D}{1+df_v} + 1$ (`sklearn`'s default), to avoid $\log 0$ when a token is in every test-set document but not the training one.

TF-IDF

Putting it together:

$$\text{TF-IDF}_{d,v} = tf_{d,v} \cdot idf_v.$$

- **Within-document signal.** $tf_{d,v}$ is high when token v appears often in document d.
- **Across-corpus signal.** idf_v is high when token v is rare across the corpus.
- **Combined effect.** TF-IDF up-weights words that are common *here* but uncommon *elsewhere*; it down-weights boilerplate.
- **Dictionary score.** For a concept dictionary $\mathcal{D}$, use

$$s_d = \sum_{v \in \mathcal{D}} \text{TF-IDF}_{d,v}$$

to get a document-level measure.

Worked example: toy corpus end-to-end (1/2)

Toy corpus d_1, d_2, d_3 (stephen / john / george). After Lambda-step pre-processing (no stop-words removed for transparency), the DTM is:

$$\mathbf{X} = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \quad \text{vocab} = (\text{stephen, is, nice, john, also, george, mean}).$$

Document lengths: $N_1 = 3$, $N_2 = 4$, $N_3 = 3$. Document frequencies df_v:

$$df = (1, 3, 2, 1, 1, 1, 1).$$

Using $idf_v = \log(D/df_v)$ with $D = 3$, $\log_e$:

$$idf = (\log 3,\ 0,\ \log \tfrac{3}{2},\ \log 3,\ \log 3,\ \log 3,\ \log 3) \approx (1.10,\ 0,\ 0.41,\ 1.10,\ 1.10,\ 1.10,\ 1.10).$$

Note: “is” appears in every document, so its idf is zero — TF-IDF will not see it.

Worked example: toy corpus end-to-end (2/2)

With $tf_{d,v} = x_{d,v}/N_d$ and TF-IDF = $tf \cdot idf$, the TF-IDF matrix is approximately:

$$\text{TF-IDF} \approx \begin{bmatrix} 0.37 & 0 & 0.14 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.10 & 0.27 & 0.27 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.37 & 0.37 \end{bmatrix}.$$

Cosine similarities (after L2-normalising rows):

$$\cos(d_1, d_2) \approx 0.20, \quad \cos(d_1, d_3) \approx 0, \quad \cos(d_2, d_3) \approx 0.$$

Observe. "Is" was dropped by IDF, "nice" (in two docs) gets a small but positive weight, and the name tokens (*stephen, john, george*) carry almost all the document-level signal. This is exactly what we wanted.

What to do with the vector representation?

Textual regression

NLP 1.0 in the $Y = f(X)$ frame: X = TF-IDF (or count) vector built from the dictionary Ω; f = ridge / lasso / random forest. *Only the feature representation changed* relative to Lectures 1–3.

For outcome Y_i associated with document i:

$$Y_i = \beta_0 + \sum_{v=1}^{V} \beta_v X_{i,v} + \varepsilon_i.$$

- $V \sim 10^4$–10^5 regressors ⇒ high-dimensional regression.
- Always penalise: LASSO (ℓ_1) gives sparsity, ridge (ℓ_2) gives stability, elastic net interpolates.
- Alternatively, project text down to a small set of variables *first* (dictionary methods, LDA, embeddings).

Applications (appendix): Hansen, McMahon & Tong (2019) on Inflation Reports and gilt yields; Ke, Kelly & Xiu (2019) on news and returns; Bana (2022) on job-ad text and salaries.

Pause — tutorial exercise 1

Friday tutorial: NLP 1.0 on a real corpus

Pre-process a corpus of FOMC statements (or 10-Ks), build a DTM with `CountVectorizer`, compute TF-IDF, and compute the cosine-similarity matrix between meetings. Apply the Loughran–McDonald negative-word dictionary as a scoring function and regress an outcome (e.g. next-day S&P return) on the score. Compare with raw-count and TF-IDF weighting.

The exercise is illustrative of every step in this section. It also exposes the limits of NLP 1.0.

Where NLP 1.0 hits a wall

Where NLP 1.0 hits a wall

Two illustrative DTMs (Hansen, Lecture 2):

	school	university	college	teacher	professor
d_1	0	5	5	0	2
d_2	10	0	0	4	0

	tank	seal	frog	animal	navy	war
d'_1	10	10	3	2	0	0
d'_2	10	10	0	0	4	3

- Synonymy. $\cos(d_1, d_2) = 0$ even though both documents are about education — they share no surface tokens.
- Polysemy. $\cos(d'_1, d'_2) > 0$ even though d'_1 is about a zoo and d'_2 is about the navy — "tank" and "seal" have multiple meanings.
- No semantic geometry. All vocabulary terms are equidistant. "Interest" and "rate" are no closer than "interest" and "pizza".

Three structural fixes

We have three complementary ways to escape NLP 1.0's count world.

1. **Model the generative process over words.** Treat each document as a draw from a probability model — Bayesian smoothing, latent topics (LDA), supervised topic learning (MNIR). Still discrete and count-based; we keep it as an *optional track*.
2. **Replace one-hot with dense, static vectors.** Map each word to one learned vector in $\mathbb{R}^d$ where geometry encodes meaning. This is the path the rest of the lecture takes (*NLP 2.0*).
3. **Replace one-hot with dense, contextual vectors.** Let the vector depend on the surrounding sequence: the same word gets different representations in different contexts (*NLP 3.0*).

Appendix: NLP 1.0 — language models for discrete data

The dense-vector route is the one that scales. Static embeddings introduce geometry; contextual embeddings are what modern LLMs learn internally through next-token prediction.

Text as Dense Vectors

The limits of one-hot

Stepping back: throughout NLP 1.0 — counts, TF-IDF, and the discrete language models now in the appendix — each word was a coordinate axis. Three problems:

1. **Dimensionality.** A document vector lives in $\mathbb{R}^V$ with $V \sim 10^4$–10^6. Most components are zero. In textual regression, $V \gg n$.
2. **No semantic geometry.** For any pair w_i, w_j,

$$\|e_i - e_j\|_2 = \sqrt{2}, \quad \cos(e_i, e_j) = 0.$$

   "Interest" and "rate" are no closer than "interest" and "pizza".
3. **Curse of dimensionality in language modelling.** Joint distributions over sequences of V-valued tokens have V^T cells. Without dense representations, generalisation is hopeless (Bengio et al. 2000).

Goal. Replace $e_v \in \mathbb{R}^V$ with $x_v \in \mathbb{R}^d$, $d \ll V$, where geometry encodes meaning.

The distributional hypothesis

> *"You shall know a word by the company it keeps."*
>
> — *J. R. Firth (1957)*

Hansen's puzzle. Suppose I tell you the unknown word $%& appears in the following sentence:

> *"Every morning last summer in Greece, I visited the $%& where I would swim, play in the sand, and sunbathe."*

Even though you have never seen $%& before, you have strong beliefs about its meaning. Why?

Because of the context. "Swim, sand, sunbathe, summer, Greece" jointly imply $%& ≈ *beach*.

- Similar words appear in similar contexts, so context gives us a data-driven notion of meaning.

This is the foundation of word embeddings: learn a representation of w that predicts (or is predicted by) the words that surround it.

Three routes to embeddings

Three complementary recipes give us dense word vectors:

1. **Count-based.** Build a matrix of co-occurrence statistics (DTM, or word-context, or PPMI) and *compress* it via SVD / PCA. Linear algebra. Closed form.
   - Latent Semantic Analysis (LSA, Deerwester et al. 1990).
   - GloVe (Pennington et al. 2014) — factorises log-co-occurrence.
2. **Prediction-based.** Train a neural net on a self-supervised task (predict context from word, or word from context). The hidden layer *is* the embedding.
   - word2vec / skip-gram and CBOW (Mikolov et al. 2013).
   - fastText (Bojanowski et al. 2017) — subword variant.
   - Modern LLMs — next-token prediction at much larger scale.
3. **Contextual.** Replace one vector per word with one vector per token occurrence; the representation depends on surrounding words.

Text as Dense Vectors

Language Modelling for Text Representation and Generation

Recap from Lecture 2: Naive Bayes

Lecture 2 introduced Naive Bayes for spam / sentiment classification.

- Bayes' theorem: $P(c \mid X) = \dfrac{P(X \mid c)\,P(c)}{P(X)}$.
- Decision rule: $\hat{c} = \arg\max_c P(X \mid c)\,P(c)$.
- Naive assumption: features conditionally independent given c.
- Bag-of-words assumption: token *order* does not matter.
- Training: count tokens within each class; normalise; smooth.

Example: if $c \in \{\text{spam}, \text{not spam}\}$ and X is the token-count vector for an email, then $P(c)$ is the prior share of spam, $P(X \mid c)$ is the probability of seeing those token counts within that class, and $P(X)$ is the overall probability of that token vector.

Today we go one step further. Naive Bayes modelled $X \mid c$ generatively. Now flip the lens: model *language itself* — assign probabilities to token sequences and learn to predict the next token. That predictive view powers dense word embeddings and, scaled up, modern LLMs.

What is a language model?

Language model: A probability distribution $P(w_1, \ldots, w_T)$ over sequences of tokens.

By the chain rule, every language model factorises as

$$P(w_1, \ldots, w_T) = P(w_1)\, P(w_2 \mid w_1)\, P(w_3 \mid w_1, w_2) \cdots P(w_T \mid w_1, \ldots, w_{T-1})$$
$$= \prod_{t=1}^{T} P(w_t \mid w_1, \ldots, w_{t-1}).$$

Different choices of how to model $P(w_t \mid w_{<t})$ give different language models:

- **Unigram** (simplest; see *appendix*): assume $P(w_t \mid w_{<t}) = P(w_t)$. Drop history altogether.
- **Bigram, n-gram**: condition on the last $n-1$ tokens.
- **Neural next-token** (this section & Week 5): condition on the full left context via a neural network.

Next-token prediction — modelling $P(w_t \mid w_{<t})$ — is the unifying objective. word2vec approximates it locally; LLMs model the whole left context at scale.

What are language models for?

A language model gives probabilities to token sequences. What we use it for depends on how much structure and context we need.

- **Unigram / simple count models.** Useful when word frequencies are the object: smoothing sparse counts, building topic models, estimating document-level word distributions, or creating interpretable text features (developed in the *appendix*).
- ***n*-gram models.** Useful when local order matters: phrases such as *New York*, *interest rate*, or *not good*.
- **Neural next-token models.** Useful when meaning depends on long context: summarisation, translation, question answering, chat, coding, and modern LLM applications.

The trade-off is interpretability versus context. Unigrams are transparent but ignore order; neural LMs are less transparent but learn contextual representations directly from next-token prediction.

Text as Dense Vectors

Next Token Prediction

From word2vec to next-token prediction

Local-context token prediction (appendix) already trains a neural network by *self-supervision*: predict a neighbouring word, with labels mined for free from raw text. A neural language model changes one thing.

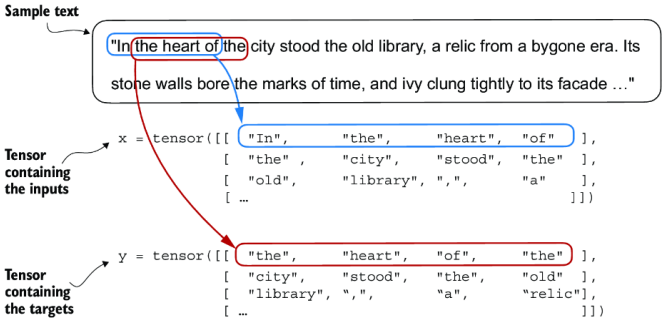
- **Local-context model (skip-gram).** Target $w_t \rightarrow$ predict a *context* word w_{t+j} inside a symmetric window; order within the window is ignored.
- **Neural language model.** Condition on the *entire left context* and predict the *next* token: $P(w_t \mid w_1, \ldots, w_{t-1})$ — the chain-rule factor from earlier, now realised by a neural network.

Same template, two changes: the target becomes the *next* token, and the context becomes *causal* (left-to-right). The lookup table of vectors, SGD, and the softmax over the vocabulary all carry over.

The next three slides follow Raschka, *Build a Large Language Model (From Scratch)*, Ch. 2: how raw text becomes the input a neural language model actually consumes.

Manufacturing the data: a sliding window (Raschka 2.6)

Tokenise the corpus into IDs, then slide a fixed context window of length L across it. Each window is one training example; the *target* is the input shifted right by one token.

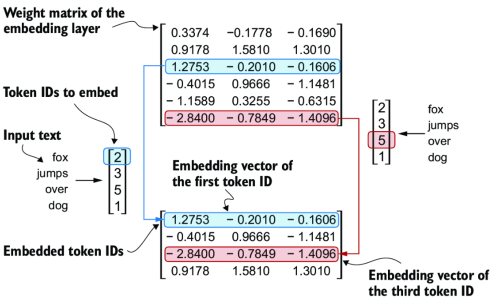


- Input $\mathbf{x} = (w_1, \ldots, w_L)$, target $\mathbf{y} = (w_2, \ldots, w_{L+1})$: predict-the-next-token at every position.
- Labels are free — just the text itself (*self-supervised*). A *stride* controls overlap; batches stack many windows.

Token embeddings: a learned lookup table (Raschka 2.7)

A neural language model cannot consume integer IDs. Its first layer is an *embedding layer*: a trainable matrix $E \in \mathbb{R}^{V \times d}$ whose row v is the dense vector for token v.

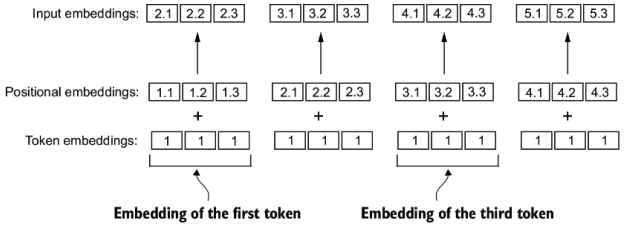
- A token ID is just a row index: "look up row v" (equivalently, $E^\top$ times a one-hot vector).
- This is exactly word2vec's matrix U — but instead of training it on a side task and freezing it, we *initialise it randomly* and learn it jointly with next-token prediction.
- $V \sim 10^4$–10^5 rows, $d \sim 256$–1024 columns; the embeddings are model *parameters*, updated by SGD.



NLP 2.0 learns these vectors as the end product; a neural LM treats the same table as just its input layer.

Encoding word positions (Raschka 2.8)

The embedding layer maps each token to the *same* vector wherever it appears — it is order-blind, exactly the bag-of-words limitation of NLP 1.0 and word2vec.

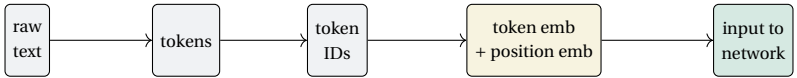


- Add a learned *positional embedding* $p_i \in \mathbb{R}^d$, one per position i; network input is $z_i = x_{w_i} + p_i$ (same dimension d, now position-aware).
- Without it, “rates raised interest” and “raised interest rates” look identical to the model.

Order is restored by addition, not by changing the architecture. GPT learns absolute positions; other schemes are sinusoidal or relative (Week 5).

The input pipeline, end to end

Putting Raschka 2.6–2.8 together, the journey from raw text to the network input is:



- A batch of windows becomes a tensor of token + positional embeddings, $\mathbb{R}^{B \times L \times d}$.
- This static input matrix is exactly what a transformer consumes (NLP 3.0 / Week 5). Self-attention then turns these *static* input vectors into *contextual* ones.

But the vectors in the lookup table already carry remarkable structure on their own — which is what we look at next.

Text as Dense Vectors

Geometry of Embeddings

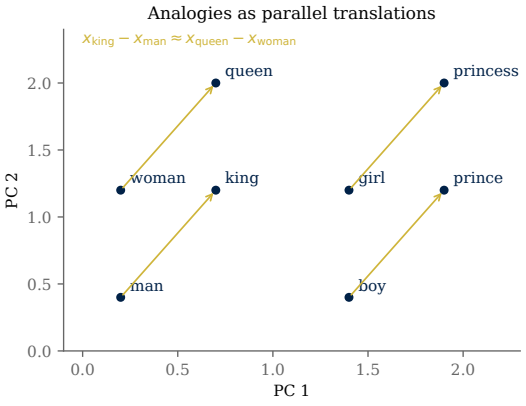
Geometry of embeddings: analogies

Trained embeddings have a striking property:

$$x_{\text{king}} - x_{\text{man}} + x_{\text{woman}} \approx x_{\text{queen}}.$$

$$x_{\text{Paris}} - x_{\text{France}} + x_{\text{Italy}} \approx x_{\text{Rome}}.$$

- Analogies encoded as parallel translations in the latent space.
- Discovered without supervision, purely from co-occurrence.
- Works for high-frequency, well-trained pairs; less reliable on rare words.



Capital–country analogies (GL Table 13.1)

Pre-trained fastText vectors. Nearest capital to country – baseline + closest neighbour:

Country	1st nearest capital	cos	2nd nearest	cos
France	Paris	.69	Brussels	.52
Spain	Madrid	.73	Lisbon	.51
Germany	Berlin	.70	Vienna	.56
Italy	Rome	.66	Vienna	.45
Sweden	Stockholm	.75	Copenhagen	.57
Russia	Moscow	.76	Beijing	.44
China	Beijing	.77	Moscow	.42

Note. The model was never told which capital belongs to which country. The relation was learned from co-occurrence in raw text.

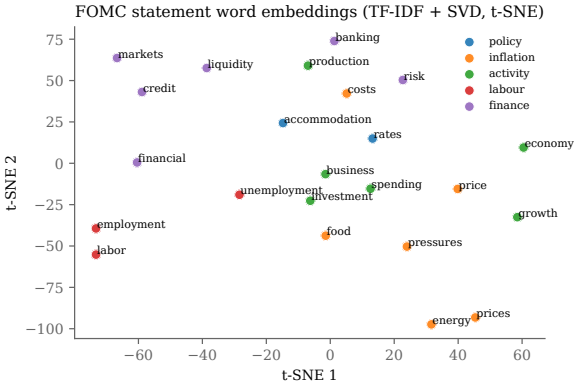
Embeddings on FOMC text

Train skip-gram on the FOMC corpus. Nearest neighbours (cosine):

near "uncertainty"		near "risk"	
uncertainties	.74	risks	.74
anxiety	.48	threat	.61
pessimism	.48	danger	.54
skepticism	.47	dangers	.46
optimism	.45	vulnerability	.46
caution	.44	chances	.45
gloom	.44	probability	.43
sensitivity	.43	possibility	.41
angst	.43	likelihood	.41

Observation. Embeddings recover synonyms (*uncertainty*/*uncertainties*) but also *affective neighbours* (*anxiety*, *gloom*, *angst*, *skepticism*). They learn lexical fields, not just words.

t-SNE: projecting embeddings to 2D



- **t-SNE** (*t-distributed stochastic neighbour embedding*) is a nonlinear visualisation method.
- It projects high-dimensional embeddings to 2D while trying to preserve local nearest neighbours.
- Policy, inflation, labour-market, and financial-stability terms cluster.
- No labels used. Structure is purely a property of how words co-occur.
- Use cases in economics: discover sectors, occupations, country groups; audit for bias.

Pretrained embeddings and transfer learning

Most empirical work does *not* train embeddings from scratch. Standard pretrained vectors:

- **GloVe** (Pennington, Socher & Manning, 2014). Trained on Wikipedia + Gigaword. Dimensions 50, 100, 200, 300.
- **fastText** (Bojanowski et al. 2017). Subword information ⇒ no out-of-vocabulary problem; works for morphologically rich languages.
- **Domain-specific.** FinBERT-style finance vectors; legal embeddings; biomedical embeddings.

Transfer learning: Use an embedding trained on Wikipedia / Common Crawl as the starting point. Optionally fine-tune on your corpus. Cuts training time and the data-volume requirement by orders of magnitude.

Hansen et al. (2021): match executive-search resumes to O*NET skill descriptors using pretrained embeddings; the search corpus is too small to train embeddings of its own.

Choice of training corpus matters

Embeddings encode relationships from *the corpus they saw.* Train on Harvard Business Review vs. Wikipedia / news; look at nearest neighbours:

HBR (“team”)	Generic (“team”)	HBR (“leader”)	Generic (“leader”)
teams	teams	leadership	leaders
project_team	squad	manager	opposition
management_team	players	person	rebel
executive_team	football	strong_leader	party
staff	coach	chief_executive	

Domain matters. The same word lives in different lexical fields. Always train (or pretrain on) a corpus that resembles your downstream task.

WEAT: implicit association in embeddings

Caliskan, Bryson & Narayanan (2017) show that word embeddings inherit *the same* biases that the Implicit Association Test (IAT) detects in humans.

WEAT: Word-Embedding Association Test. For two target word sets X, Y (e.g. male / female names) and two attribute sets A, B (e.g. career / family terms):

$$s(w, A, B) = \text{mean}_{a \in A} \cos(w, a) - \text{mean}_{b \in B} \cos(w, b).$$

$$\text{WEAT} = \frac{\sum_{x \in X} s(x, A, B) - \sum_{y \in Y} s(y, A, B)}{\text{std}_{c \in X \cup Y}\, s(c, A, B)}.$$

Standardised effect size measuring how differently the two target sets align with the two attribute sets.
Caliskan et al.: WEAT *replicates* the human IAT pattern across many bias dimensions (race-pleasant, gender-career, age-pleasant).

Pause — tutorial exercise 2

Friday tutorial: word embeddings in action

Load pretrained GloVe (or fastText) embeddings. Inspect nearest neighbours of *inflation*, *growth*, *risk*. Run two analogies (king − man + woman; Paris − France + Italy). Compute WEAT for a gender–career bias pair. *Stretch*: train a 50-dim skip-gram on the FOMC corpus from Exercise 1 with `gensim` and compare nearest neighbours to the generic pretrained vectors.

Together with Exercise 1, you will have walked the full NLP 1.0 / 2.0 pipeline end to end on a real economic-text corpus.

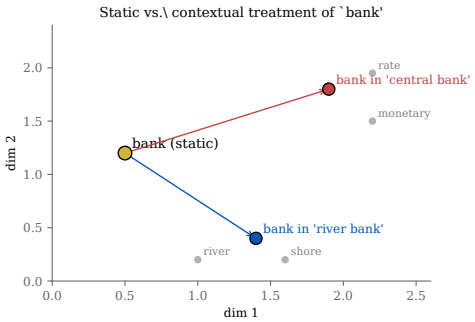
NLP 3.0 — Beyond static vectors

Polysemy — one word, many meanings

Static embeddings assign *one* vector to each surface form. But many words have multiple distinct meanings.

- *mouse*: small furry rodent / pointing device.
- *bank*: river edge / financial institution.
- *rate*: interest rate / pace / evaluate.
- *shock*: monetary-policy shock / electrical shock / surprise.

word2vec collapses all senses into a single vector, located somewhere between them — rarely a good match for any single sense.



Pronouns and reference

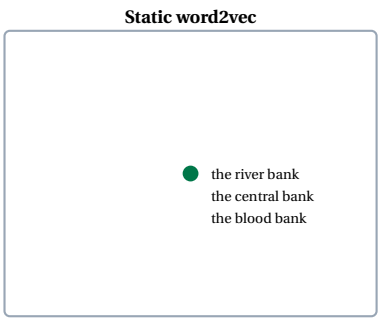
Consider two sentences:

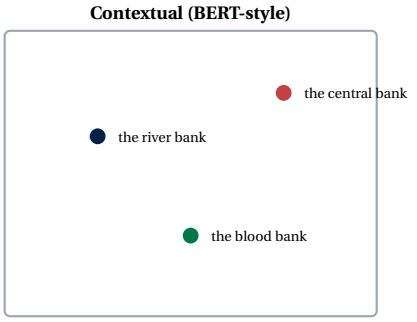
- *"The cat ran after the mouse as it escaped."* Here *it* refers to the mouse.
- *"The ECB is ready to do whatever it takes to preserve the euro. And believe me, it will be enough."* Here *it* refers to the ECB's commitment.

In both, the surface token *it* is identical. But to make sense of either sentence, our representation of *it* has to be different.

Gaillac & L'Hour, Ch. 14: "Unlike previous approaches, [modern language models] take the actual context of a sentence into account, making them very flexible and powerful… These models stem from a computer-science literature that directly models the language in a realistic fashion."

What we need: contextual embeddings





Same surface token, different sentences: word2vec returns one fixed vector; a contextual encoder returns a different vector each time, conditional on the surrounding sequence.

How is this achieved? Self-attention (preview)

The technology that produces contextual embeddings is the *transformer*, introduced by Vaswani et al. (2017) and used in BERT (Devlin et al. 2019), GPT (Radford et al. 2018+), etc.

- Replace the static lookup table with a deep neural network that processes the *entire sequence* at once.
- Each token's output vector is a weighted combination of every input token's vector — weights determined by a learnable *attention* mechanism.
- Pre-trained on masked-token or next-token prediction over hundreds of billions of words.
- Output: a context-conditioned vector for every position in the input.

Why now? Hardware (GPU/TPU), data scale (web-scale corpora), and the attention mechanism converged in 2017–2019 to make this practical.

Coming up in Weeks 5 and 6

Weeks 5 and 6 take the contextual story end-to-end:

- Attention, self-attention, multi-head attention from first principles.
- Transformer blocks; the BERT and GPT architectures.
- Pre-training tasks (masked language modelling vs. causal language modelling).
- Fine-tuning, prompt engineering, instruction tuning.
- Scaling laws and what changed since 2020.
- Building a tiny GPT and training it on FOMC text.

The point. We move from static token vectors to the transformer machinery that turns a token's left context into a contextual representation and then a next-token distribution.

Review questions

1. Why does cosine similarity beat Euclidean distance for documents of unequal length? Give a one-line example.
2. A neural language model is trained by next-token prediction on (input, target) pairs from a sliding window. Explain how the pairs are built and why no human labels are needed. What is the role of the token-embedding layer, and why do we also add positional embeddings?
3. Why does the softmax denominator in skip-gram make naive maximum-likelihood estimation intractable, and how does negative sampling get around it?
4. Give two concrete sentences in which a static word embedding for *rate* returns the same vector but should differ. Explain in one sentence why a transformer can do better.

A.1 NLP 1.0: Language Models for Discrete Data

The unigram model

Treat tokens in a document as i.i.d. draws from a single distribution:

$$P(w_1, \ldots, w_T \mid \beta) = \prod_{t=1}^{T} \beta_{w_t}, \qquad \beta \in \Delta^{V-1}.$$

All information collapses into the document's count vector $C = (C_1, \ldots, C_V)$ with $C_v = \sum_t \mathbf{1}\{w_t = v\}$. The likelihood becomes

$$L(\beta) = \prod_{v=1}^{V} \beta_v^{C_v}, \qquad \log L(\beta) = \sum_{v=1}^{V} C_v \log \beta_v.$$

Conditional on T, $C \sim \text{Multinomial}(\beta, T)$. The unigram model is exactly multinomial.

Maximum likelihood for the unigram

Maximise $\sum_v C_v \log \beta_v$ subject to $\sum_v \beta_v = 1$. Lagrangian:

$$\mathscr{L}(\beta, \lambda) = \sum_v C_v \log \beta_v - \lambda\Big(\sum_v \beta_v - 1\Big).$$

First-order conditions: $C_v / \beta_v = \lambda$ for all v, so $\beta_v = C_v / \lambda$. Summing over v and using the constraint, $\lambda = \sum_v C_v = T$. Therefore

$$\boxed{\widehat{\beta}_v^{\mathrm{MLE}} = \frac{C_v}{T} = f_v}$$

— the MLE is just the empirical frequency. Intuitive, but as we will see, dangerous on small samples.

Why this isn't enough — a Portuguese example

Imagine you encounter a small fragment of an unfamiliar language:

"eles bebem" *(they drink)*

Assume the true vocabulary has $V = 10{,}000$ words. The MLE on this two-token corpus assigns:

$$\widehat{\beta}_{\text{eles}} = 0.5, \quad \widehat{\beta}_{\text{bebem}} = 0.5, \quad \widehat{\beta}_{v} = 0 \text{ for every other } v.$$

The point: MLE treats unseen words as impossible. That is too strong when the corpus is tiny.

- Before seeing the fragment, you know Portuguese has thousands of regularly used words.
- You also know word frequencies are heavy-tailed: many words are rare, but not impossible.
- You might even borrow information from a related language, such as Spanish, to shape expectations.

We need an estimator that keeps probability mass for plausible unseen words. Bayesian smoothing does exactly that.

Bayes' rule for parameters

Treat β as a random variable with prior $p(\beta)$ encoding what we know before seeing C. Then

$$\underbrace{p(\beta \mid C)}_{\text{posterior}} = \frac{\overbrace{p(C \mid \beta)}^{\text{likelihood}}\overbrace{p(\beta)}^{\text{prior}}}{\underbrace{p(C)}_{\text{evidence}}} \propto p(C \mid \beta)\, p(\beta).$$

Everything we want to know about β given the data lives in $p(\beta \mid C)$:

- Point estimate? Use the posterior mode (MAP) or mean.
- Uncertainty quantification? Use posterior credible intervals.
- Predictive distribution? Integrate over the posterior: $p(\tilde{C} \mid C) = \int p(\tilde{C} \mid \beta) p(\beta \mid C) d\beta$.

What is a Bayesian estimate?

Three useful summaries of the posterior $p(\beta \mid C)$:

1. **Maximum a posteriori (MAP).** $\widehat{\beta}_{\text{MAP}} = \arg\max_{\beta} p(\beta \mid C)$. Coincides with the penalised-MLE perspective: $\log p(\beta \mid C) = \log p(C \mid \beta) + \log p(\beta) + \text{const}$, so a log-prior is a penalty on β.
2. **Posterior mean.** $\mathbb{E}[\beta \mid C]$. Optimal for squared-error loss; smooth.
3. **Credible interval.** A such that $P(\beta \in A \mid C) \geq 1 - \alpha$. Direct probabilistic interpretation.

In modern empirical work, MAP and posterior mean are the two summaries you actually see. We will derive both for the unigram model.

Conjugate priors and the Dirichlet

The multinomial likelihood is $\prod_v \beta_v^{C_v}$. A prior with the *same functional form* multiplies cleanly into the likelihood and yields a posterior of the same form. That is a *conjugate prior*.

Dirichlet(η): $p(\beta \mid \eta) = \dfrac{\Gamma(\sum_v \eta_v)}{\prod_v \Gamma(\eta_v)} \prod_{v=1}^{V} \beta_v^{\eta_v - 1}, \quad \beta \in \Delta^{V-1}, \quad \eta \in \mathbb{R}_+^V.$

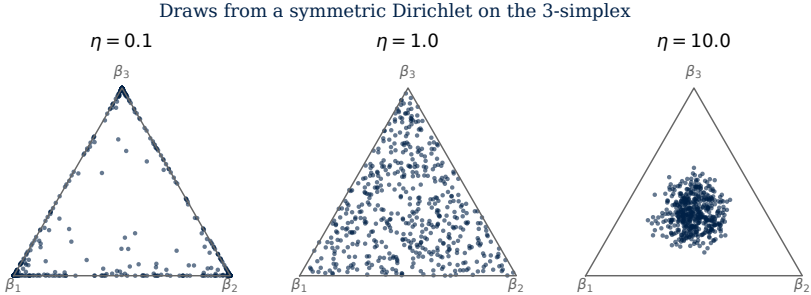
- Marginal $\beta_v \sim \text{Beta}(\eta_v, \sum_{u \neq v} \eta_u)$.
- Mean: $\mathbb{E}[\beta_v] = \eta_v / \eta_0$ where $\eta_0 = \sum_v \eta_v$.
- Variance: $\text{Var}(\beta_v) = \eta_v(\eta_0 - \eta_v) / [\eta_0^2(\eta_0 + 1)]$.

Useful hyperparameter convention: $\eta = (\eta, \ldots, \eta)$ symmetric. Then $\eta_0 = V\eta$ and the magnitude η controls how peaked the distribution over the simplex is.

Reading the Dirichlet

Symmetric Dirichlet$(\eta, \ldots, \eta)$ on the V-simplex:

- $\eta = 1$: *uniform* on the simplex. All distributions equally likely.
- $\eta > 1$: mass concentrated at the centre. "All words roughly equally likely."
- $\eta < 1$: mass concentrated at the corners. "Most words have near-zero probability; a few dominate."



Posterior with conjugate Dirichlet

Combine the Dirichlet prior with the multinomial likelihood:

$$\begin{aligned} p(\beta \mid C, \eta) &\propto p(C \mid \beta)\, p(\beta \mid \eta) \\ &\propto \left(\prod_v \beta_v^{C_v}\right)\left(\prod_v \beta_v^{\eta_v - 1}\right) \\ &= \prod_v \beta_v^{C_v + \eta_v - 1}. \end{aligned}$$

This is the kernel of a Dirichlet with parameters $\eta_v + C_v$. So

$$\boxed{\beta \mid C, \eta \sim \text{Dirichlet}\big(\eta_1 + C_1, \ldots, \eta_V + C_V\big).}$$

Interpretation. Hyperparameters η_v act like *pseudo-counts* added to the data. The Bayesian estimator is the empirical frequency with extra observations baked in by the prior.

Posterior mean: a smoothed MLE

Using the Dirichlet mean formula on the posterior:

$$\mathbb{E}[\beta_v \mid C, \eta] = \frac{\eta_v + C_v}{\eta_0 + T}, \qquad \eta_0 = \sum_v \eta_v, \;\; T = \sum_v C_v.$$

Compare with the MLE $\widehat{\beta}_v^{\mathrm{MLE}} = C_v / T$:

- Bayesian estimator is a convex combination of the prior mean η_v/η_0 and the MLE C_v/T, with weights $\eta_0/(\eta_0+T)$ and $T/(\eta_0+T)$.
- Tokens never seen in the data still get positive probability. Catastrophe of the Portuguese example is fixed.
- This is exactly the Laplace "add-one smoothing" used in classical NLP — a special case with $\eta_v = 1$ for all v.

Data overwhelming the prior

As we accumulate data, the prior fades:

$$\mathbb{E}[\beta_v \mid C, \eta] = \underbrace{\frac{\eta_0}{\eta_0 + T}}_{\to 0} \cdot \frac{\eta_v}{\eta_0} + \underbrace{\frac{T}{\eta_0 + T}}_{\to 1} \cdot \frac{C_v}{T}.$$

Variance:

$$\mathrm{Var}(\beta_v \mid C) = \frac{(\eta_v + C_v)(\eta_0 + T - \eta_v - C_v)}{(\eta_0 + T)^2(\eta_0 + T + 1)} \xrightarrow{T \to \infty} 0.$$

Bernstein–von Mises: Under regularity conditions, the posterior of a finite-dimensional parameter concentrates around the MLE at rate $1/\sqrt{T}$ as $T \to \infty$. Bayesian and frequentist estimators agree asymptotically; they disagree where it matters — on small samples.

Ridge regression as Bayesian inference

Same logic outside the unigram. Linear model $y_i = x_i^\top \beta + \varepsilon_i$, $\varepsilon_i \sim \mathcal{N}(0, \sigma^2)$, with prior $\beta_j \sim \mathcal{N}(0, \tau^2)$ independently. The posterior is proportional to

$$\prod_i \exp\Big(-\tfrac{(y_i - x_i^\top \beta)^2}{2\sigma^2}\Big) \prod_j \exp\Big(-\tfrac{\beta_j^2}{2\tau^2}\Big).$$

Taking log and dropping constants, the MAP minimises

$$\sum_i (y_i - x_i^\top \beta)^2 \; + \; \frac{\sigma^2}{\tau^2} \sum_j \beta_j^2.$$

This is ridge regression with $\lambda = \sigma^2/\tau^2$. The Gaussian prior *is* the ℓ_2 penalty. Stronger prior $\Rightarrow$ smaller $\tau \Rightarrow$ bigger λ.

LASSO as Bayesian inference

Same model, replace the Gaussian prior with a Laplace prior:

$$p(\beta_j) = \frac{\lambda'}{2} \exp\big(-\lambda'|\beta_j|\big).$$

The MAP now minimises

$$\sum_i (y_i - x_i^\top \beta)^2 + 2\sigma^2 \lambda' \sum_j |\beta_j|.$$

This is LASSO with penalty $\lambda = 2\sigma^2\lambda'$. The Laplace prior has a kink at zero, which is what produces sparsity — the MAP is exactly zero for many components.

Unifying picture: Penalisation methods you have already seen — ridge, LASSO, elastic net — correspond to particular log-prior choices on the regression coefficients. Bayesian inference and penalised regression are two languages for the same shrinkage idea.

A.1 NLP 1.0: Language Models for Discrete Data

Topics

Allowing topics: the unigram mixture

Real corpora are not a single multinomial. The probability of seeing *inflation* in a document about employment is lower than in a document about monetary policy. Introduce a latent category $Z \in \{1, \dots, K\}$.

$$P(w_1, \dots, w_T \mid \beta, \rho) = \prod_{t=1}^{T} \sum_{k=1}^{K} \rho_k \beta_{w_t, k}.$$

- $\rho_k = P(Z = k)$, $\sum_k \rho_k = 1$: marginal probabilities over topics.
- $\beta_{v,k} = P(W = v \mid Z = k)$: topic-specific word distribution.
- Each *document* is drawn from a single topic; the topic is unobserved.

Cannot maximise directly. The latent variable makes the log-likelihood non-concave.

Estimation via Expectation–Maximisation

If we observed the topic Z_j for each document, the log-likelihood would be

$$\ell(\beta, \rho \mid Z) = \sum_j \sum_k \mathbf{1}\{Z_j = k\} \Big[\log \rho_k + \sum_v C_{j,v} \log \beta_{v,k} \Big].$$

We don't. EM alternates between guessing Z and re-fitting β, ρ:

E-step. Use current (β, ρ) to compute posterior topic responsibilities:

$$\widehat{Z}^{(t)}_{j,k} = \frac{\rho^{(t)}_k \prod_v (\beta^{(t)}_{v,k})^{C_{j,v}}}{\sum_{k'} \rho^{(t)}_{k'} \prod_v (\beta^{(t)}_{v,k'})^{C_{j,v}}}.$$

M-step. Re-fit by weighted counting:

$$\beta^{(t+1)}_{v,k} = \frac{\sum_j \widehat{Z}^{(t)}_{j,k} C_{j,v}}{\sum_j \widehat{Z}^{(t)}_{j,k} \sum_v C_{j,v}}, \qquad \rho^{(t+1)}_k = \frac{1}{D} \sum_j \widehat{Z}^{(t)}_{j,k}.$$

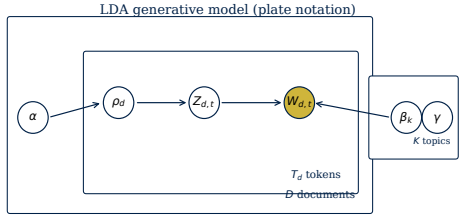
Each iteration increases $\log L$. Topics emerge from the data, are interpreted ex-post by inspecting top words.

Latent Dirichlet Allocation (Blei et al. 2003)

Relax the "one topic per document" assumption. Each document is a *mixture* of topics.

Generative model. For each document d:

1. Draw topic distribution $\rho_d \sim \text{Dirichlet}(\alpha) \in \Delta^{K-1}$.
2. For each of T_d word positions:
   2.1 Draw a topic $Z_{d,t} \sim \text{Multinomial}(\rho_d)$.
   2.2 Draw a word $W_{d,t} \sim \text{Multinomial}(\beta_{\cdot,Z_{d,t}})$, where $\beta_{\cdot,k} \sim \text{Dirichlet}(\gamma)$ a priori.



LDA as soft matrix factorisation

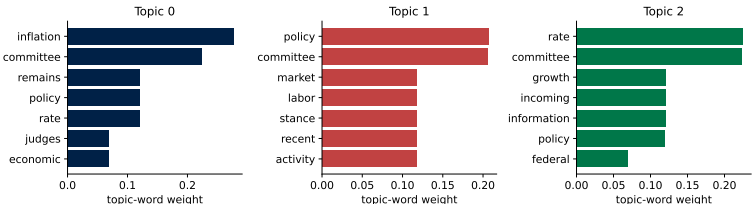
Marginalising the latent topic Z_t inside each word draw,

$$P(W_{d,t} = v \mid \rho_d, \beta) = \sum_{k=1}^{K} \rho_{d,k}\, \beta_{v,k}.$$

Stacking documents and topics into matrices $\rho \in \mathbb{R}_+^{D \times K}$ and $\beta \in \mathbb{R}_+^{V \times K}$ (with row / column sums equal to one),

$$\frac{1}{T_d} C_{d,\cdot} \approx \rho_d \beta^\top, \qquad \frac{C}{T} \approx \rho \beta^\top.$$

LDA is a probabilistic, non-negative matrix factorisation of the row-normalised DTM into doc-topic and topic-word distributions.



Estimation: Gibbs and variational EM

The integral $P(W_{1:T} \mid \alpha, \beta) = \int p(\rho \mid \alpha) \prod_t \sum_k \rho_k \beta_{w_t,k} \, d\rho$ has no closed form. Two standard remedies:

- **Collapsed Gibbs sampling.** Sample $Z_{d,t}$ for one word at a time conditional on the others, marginalising over ρ and β. Easy to implement; mixes slowly on large corpora.
- **Variational EM.** Approximate $p(\rho, \beta, Z \mid W)$ with a factorised distribution $q(\rho)\, q(\beta)\, q(Z)$; optimise the evidence lower bound (ELBO). Much faster; used in `gensim`, `scikit-learn`.

Choice of K: held-out perplexity, or a coherence score (NPMI-based). Both are imperfect; honest practice is to report several K.

LDA in practice — Hansen, McMahon & Prat (2018)

Question. Did the 1993 publication of FOMC transcripts change deliberation?

Setup.

- Natural experiment: 1993 Senate hearings revealed FOMC tapes had always been kept; transcripts started being published.
- Corpus: 46,169 statements, 8,615 vocabulary tokens.
- Run LDA with $K = 50$ and $K = 70$.

Outcomes. For each speaker × meeting, build:

- topic-concentration Herfindahl;
- similarity of one speaker's topic profile to the FOMC average;
- % time spent on factual vs. policy topics.

Finding. Post-1993, speakers' topic mixes *converge* toward the chair's. Evidence of conformist behaviour induced by transparency.

A.1 NLP 1.0: Language Models for Discrete Data

Supervised Topic Learning

Multinomial Inverse Regression — motivation

LDA is unsupervised: topics are discovered without reference to any outcome y_d. But sometimes we have y_d and want a text representation *tailored to predicting it.*

- Predict firm-level returns from 10-K text.
- Predict political affiliation from speech.
- Predict next-quarter GDP growth from news.

Discriminative approach. Penalised regression $y_d = X_d^\top \beta + \varepsilon_d$ with X_d = TF-IDF. Standard, but throws away the generative structure of language.

MNIR (Taddy 2013). Be *generative* for $X_d \mid y_d$, then use Bayes' rule to invert. Ng & Jordan (2001): generative classifiers can have lower asymptotic error and reach it faster, especially on small samples.

MNIR — the model

Conditional on covariates y_d (an outcome, or a label, or a vector of both), the document's word counts are

$$X_d \sim \text{Multinomial}\big(\beta_d, N_d\big), \qquad \beta_{d,v} = \frac{\exp\big(a_v + y_d^\top b_v\big)}{\sum_{v'} \exp\big(a_{v'} + y_d^\top b_{v'}\big)}.$$

- a_v: baseline log-frequency of token v.
- b_v: how token v's log-frequency shifts with the covariates.
- Multinomial logistic link — exactly a discrete-choice model with V alternatives and N_d trials.
- Many parameters. a is V-dim, b is $V \times \dim(y)$. Need regularisation.

Gamma-Lasso prior. Token-specific Laplace prior on b_v with a Gamma hyper-prior on the scale — adapts the penalty per token. Implemented in the R package `textir` (Taddy, "distrom").

MNIR — sufficient reduction

Key result (Taddy 2013). Given fitted $\hat{b}$, define the *sufficient reduction* projection of document d along covariate j:

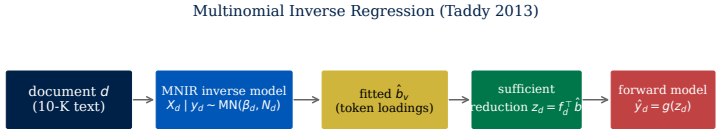
$$z_{d,j} = f_d^\top \hat{b}_{\cdot,j}, \qquad f_d = X_d / N_d = \text{token frequencies.}$$

Then $z_{d,j}$ is sufficient for $y_{d,j}$ in the sense

$$y_{d,j} \perp X_d, N_d \,\Big|\, z_{d,j}, y_{d,-j}.$$

All information in the high-dimensional document relevant for y_j is captured in the scalar $z_{d,j}$.

Workflow.



Text → generative model on words → low-dim sufficient projection → outcome regression

MNIR in practice — COVID-19 firm exposures

Application: which industries / firms were hit hardest by the pandemic shock?

- Outcome y_d: firm d's abnormal equity return on "pandemic-fallout" jump dates.
- Text X_d: Risk Factors section of firm d's most recent 10-K (2010–2016, ~ 2,100 firms).
- Fit MNIR; build targeted risk-factor indices by grouping high-loading bigrams (*ecommerce*, *web services*, *oil and gas*, *restaurants*, ...).

	Pandemic fallout days	**Super Tuesday**
Baseline (sector FE, leverage, size)	0.329	0.199
Baseline + targeted risk factors	0.410	0.242
Baseline + full sufficient reduction	0.501	0.349

Text-derived risk factors explain ~25% more cross-sectional return variance on pandemic-fallout days. Negative loadings: *retail, restaurants, hotels, airlines, oil, gas.* Positive: *ecommerce, web services, raw metals, vaccines.*

A.1 — n-grams and pointwise mutual information

Problem. Sentence “He lives in the city of New York” contains `he_lives`, `lives_in`, `in_the`, `the_city`, `city_of`, `of_New`, `New_York`. Only the last carries genuine information beyond its parts.

Pointwise mutual information: $\mathrm{PMI}(w_t, w_c) = \log \dfrac{\widehat{P}(w_t, w_c)}{\widehat{P}(w_t)\,\widehat{P}(w_c)}$

- Numerator: empirical probability of seeing w_c in the context of w_t.
- Denominator: probability if the two occurred independently.
- PMI $> 0 \Leftrightarrow$ words co-occur more than chance.

Many pairs never co-occur, so $\widehat{P}(w_t, w_c) = 0$ and PMI is undefined. Standard fix:

$$\mathrm{PPMI}(w_t, w_c) = \max\bigl(0, \mathrm{PMI}(w_t, w_c)\bigr).$$

Back

A.2 Dimensionality Reduction via SVD

Route 1: Latent Semantic Analysis

Start from the document-term matrix. Apply truncated SVD:

$$\underbrace{C}_{D\times V} \approx \underbrace{U_d}_{D\times d}\underbrace{\Sigma_d}_{d\times d}\underbrace{V_d^\top}_{d\times V}.$$

Row i of $U_d\Sigma_d^{1/2}$ is a d-dim *document* embedding; row j of $V_d\Sigma_d^{1/2}$ is a d-dim *word* embedding.

- This is just PCA on the (uncentred) DTM.
- Choose d from a scree plot, or by perplexity / held-out reconstruction.
- Pre-multiply C by a TF-IDF reweighting first to neutralise stop-word dominance.

Deerwester et al. (1990): the foundational paper. Latent Semantic *Indexing* when used for information retrieval; *Analysis* when used for representation.

PCA in 30 seconds

We have already met PCA in Lecture 1. The story:

- Eigen-decomposition of a square symmetric matrix: $M = U\Lambda U^\top$ with U orthonormal and Λ diagonal.
- Apply this to the variance-covariance matrix of a centred data matrix; columns of U are principal axes; eigenvalues give variance captured.
- Project data onto top-k principal axes $\Rightarrow$ optimal rank-k reconstruction (Eckart–Young).

SVD generalises eigendecomposition to non-square matrices:

$$X = D\Lambda V^\top, \qquad XX^\top = D\Lambda^2 D^\top.$$

Columns of D are document-side factors; columns of V are term-side factors.

Eckart–Young: best low-rank approximation

Eckart–Young theorem: Among all rank-k matrices $X^{(k)}$, the one minimising $\|X - X^{(k)}\|_F^2$ is $X^{(k)} = D\,\Lambda^{(k)}\,V^\top$, where $\Lambda^{(k)}$ keeps the top-k singular values and zeroes the rest.

Implication for LSA. The best k-dimensional embedding of documents and words — in the Frobenius sense — is the truncated SVD. No other linear factorisation does better.

Caveat. "Best in Frobenius norm" is not the same as "best for downstream prediction". LSA gives a strong baseline; word2vec / GloVe often beat it on analogy and similarity tasks, even though they target related (but non-identical) objectives.

Worked LSA: car / automobile / ship / boat (1/2)

Six toy documents, four-token vocabulary (Hansen, Lecture 2):

$$C = \begin{array}{c|cccc} & \text{car} & \text{automobile} & \text{ship} & \text{boat} \\ \hline d_1 & 10 & 0 & 1 & 0 \\ d_2 & 5 & 5 & 1 & 1 \\ d_3 & 0 & 14 & 0 & 0 \\ d_4 & 0 & 2 & 10 & 5 \\ d_5 & 1 & 0 & 20 & 21 \\ d_6 & 0 & 0 & 2 & 7 \end{array}$$

Cosine similarity matrix on the full matrix:

$$\cos = \begin{bmatrix} 1 & .70 & .00 & .08 & .10 & .02 \\ .70 & 1 & .69 & .30 & .21 & .17 \\ .00 & .69 & 1 & .17 & .00 & .00 \\ .08 & .30 & .17 & 1 & .92 & .66 \\ .10 & .21 & .00 & .92 & 1 & .88 \\ .02 & .17 & .00 & .66 & .88 & 1 \end{bmatrix}.$$

Why is $\cos(d_1, d_3) = 0$? Synonymy. “Car” and “automobile” are different tokens to NLP 1.0.

Worked LSA: car / automobile / ship / boat (2/2)

Truncated SVD with $k = 2$. Singular values are $(31.61, 15.14, 10.90, 5.03)$ — the first two dominate. Cosine similarities on the rank-2 reconstruction:

$$\cos^{(2)} = \begin{bmatrix} 1 & .97 & .91 & .60 & .45 & .47 \\ .97 & 1 & .97 & .43 & .26 & .29 \\ .91 & .97 & 1 & .23 & .05 & .07 \\ .60 & .43 & .23 & 1 & .98 & .98 \\ .45 & .26 & .05 & .98 & 1 & .99 \\ .47 & .29 & .07 & .98 & .99 & 1 \end{bmatrix}.$$

- Synonymy resolved: $\cos^{(2)}(d_1, d_3)$ jumped from 0 to 0.91.
- Two clusters emerge cleanly — cars and boats.

Take-away. A linear dimension reduction *discovered* that “car” and “automobile” are related, without anyone telling it so.

LSA in economics

Like dictionary methods, LSA found a foothold in finance and macro because it is fast, interpretable, and easy to apply at scale:

- **Boukus & Rosenberg (2006).** LSA on FOMC minutes; document embeddings predict subsequent yield-curve moves.
- **Hendry & Madeley (2010).** LSA on Bank of Canada communications; topics correlate with policy decisions.
- **Naijaria (2018).** LSA on scientific abstracts; embedding distances measure overlap in research agendas across countries.
- **Ter Ellen et al. (2021).** LSA on financial-newspaper articles to derive narrative monetary-policy shocks.

Why has LSA been displaced for newer empirical work? Word-context predictions capture more than co-occurrence counts, and pretrained word2vec / GloVe / fastText vectors are freely available.

A.3 Local Context Token Prediction

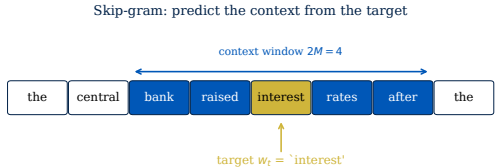
Predict from local context

Setup. For each target word w_t in the corpus, look at a window of size M:

$$(w_{t-M}, \ldots, w_{t-1}, w_t, w_{t+1}, \ldots, w_{t+M}).$$

Train a model whose only job is to predict the context from the target (or vice versa). The *hidden layer* of that model is the embedding.

Self-supervised. The labels come for free from the corpus itself — we never need a human-annotated dataset.



Once trained, throw away the prediction head. Keep the matrix of embeddings.

word2vec setup

Each vocabulary token v is associated with *two* vectors:

- $x_v \in \mathbb{R}^d$: the *target* (or "input") embedding.
- $y_v \in \mathbb{R}^d$: the *context* (or "output") embedding.

The model says that the probability of seeing context word c in the window of target word v depends on the inner product $y_c^\top x_v$ — larger inner product, more likely co-occurrence.

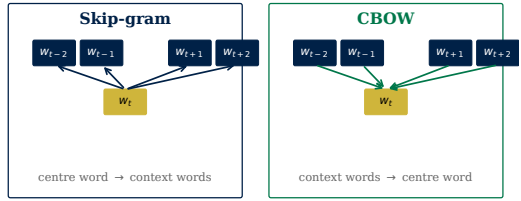
After training, the *target* embeddings $\{x_v\}_{v=1}^V$ are usually what you keep. Stack them into the matrix $U \in \mathbb{R}^{V \times d}$ and use as a lookup table.

Why two embeddings? The asymmetry between "target" and "context" simplifies the training objective. A simpler variant ties $y_v = x_v$ and halves the parameters — often works fine.

Skip-gram: predict context from target

Probability of seeing word w as a context of target v:

$$P(W_c = w \mid W_t = v) = \frac{\exp(y_w^\top x_v)}{\sum_{k=1}^{V} \exp(y_k^\top x_v)}.$$



Joint probability of the window (assuming independence inside the window):

$$P(\text{context} \mid v) = \prod_{j \neq 0} P(W_{t+j} \mid W_t = v).$$

Log-likelihood of the corpus:

$$\ell = \sum_{t=1}^{T} \sum_{j=-M, j\neq 0}^{M} \log P(W_{t+j} = w_{t+j} \mid W_t = w_t).$$

Maximise over $\{x_v, y_v\}_{v=1}^{V}$ via SGD.

CBOW: predict target from context

Mirror of skip-gram. Average the context-word embeddings:

$$\bar{u}_t = \frac{1}{2M} \sum_{j=-M, j\neq 0}^{M} y_{t+j}.$$

Then the target word distribution is

$$P(W_t = v \mid \text{context}) = \frac{\exp(x_v^\top \bar{u}_t)}{\sum_k \exp(x_k^\top \bar{u}_t)}.$$

- Faster to train than skip-gram (one forward pass per window instead of $2M$).
- Tends to work better on frequent words; skip-gram works better on rare words.
- In both, the embeddings $\{x_v\}$ are what you keep.

The softmax bottleneck

Both skip-gram and CBOW objectives have a softmax denominator:

$$\sum_{k=1}^{V} \exp(y_k^\top x_t).$$

For $V \sim 10^5$ or 10^6 this is computationally prohibitive — every gradient update touches every embedding.

Two standard remedies:

- **Subsample frequent words** (next slide).
- **Negative sampling** (slide after that): replace the softmax with a logistic-regression-style binary classifier.

Trick 1: subsample frequent words

Frequent words (*the, of, in*) carry little information and dominate the gradient. Mikolov et al. propose: drop each occurrence of w with probability

$$P_{\text{drop}}(w) = \max\left(0,\, 1 - \sqrt{\frac{\delta}{\text{freq}(w)}}\right), \qquad \delta \in [10^{-5}, 10^{-3}].$$

- Very frequent w: $P_{\text{drop}}(w)$ close to 1, dropped often.
- Rare w: $P_{\text{drop}}(w) = 0$, always kept.
- Effective vocabulary becomes smaller and more informative.

Empirically, subsampling alone speeds up training by a large factor and *improves* the quality of the learned embeddings.

Trick 2: negative sampling

Reframe word2vec as a *binary* classification problem: distinguish observed target–context pairs from random pairs. Let $D = 1$ if (x, y) comes from the corpus, 0 otherwise.

$$P(D = 1 \mid x, y) \;=\; \sigma(y^\top x) \;=\; \frac{1}{1 + e^{-y^\top x}}.$$

For each observed pair, sample B "negative" pairs (x, y_b) with y_b drawn at random from a noise distribution.

Objective (per observation, summed across the corpus):

$$\ell \;=\; \sum_{t=1}^{T} \left[\log\sigma(y_0^\top x_t) \;+\; \sum_{b=1}^{B} \log\sigma(-y_b^\top x_t) \right].$$

Sampling distribution: probability $\propto \text{freq}(w)^{3/4}$. Recommended: $B \in [5, 20]$ for small corpora; $B \in [2, 5]$ for large ones.

Negative sampling — intuition

Negative sampling replaces the V-way softmax with $B+1$ logistic regressions:

- Make $y_0^\top x_t$ *large*: the true context word should score highly with the target.
- Make $y_b^\top x_t$ *small* for B random words: noise pairs should score poorly.
- No denominator sum over the full vocabulary — gradient is cheap.

Result. Training scales linearly in the corpus size, independently of V. A vocabulary of 10^6 is now feasible on a laptop.

Levy & Goldberg (2014): the minimiser of the negative-sampling objective implicitly factorises a shifted PMI matrix. Numerical equivalence to a (more carefully weighted) SVD of PPMI.

Hyperparameters and training

Three knobs matter most:

- **Window size M.** 5–10 typical. Smaller M ⇒ syntactic similarities (*walked* / *walking*); larger M ⇒ topical similarities (*walked* / *hike*).
- **Embedding dim d.** 100–300 typical. Beyond $d = 300$, marginal gains.
- **Pretrained vs from-scratch.** On a ~10M-token corpus you can train your own; below that, use pretrained vectors and optionally fine-tune.

Rodriguez & Spirling (2022): for political-science corpora, $d > 100$ and $M \geq 5$ are sufficient; further increases bring only marginal improvements.

Antoniak & Mimno (2018): on moderate corpora, cosine similarities between embeddings can be unstable across re-runs — bootstrap and average if reporting numbers.

A.4 NLP 1.0 in Action: What Economists Do with Counts

The dictionary-method recipe

Most empirical-economics papers using text follow the same recipe:

1. Pick (or build) a *dictionary* of terms that capture the concept of interest.
2. For each document, count (or score) occurrences of dictionary terms.
3. Normalise — divide by document length, by total terms, or by TF-IDF.
4. Plug the score into a downstream regression / classifier.

Why this works in economics: Most outcomes we care about (returns, votes, prices) can be predicted by a handful of high-signal lexical features. Dictionary methods are interpretable: every regression coefficient maps to a list of words. That makes them defensible in seminar talks.

Tetlock (2007) — WSJ sentiment and returns

Data. Wall Street Journal "Abreast of the Market" column, 1984–1999.

Dictionary. Harvard IV-4 psycho-social dictionary, 77 categories.

Method. Count category words in each article, then extract principal components.

Constructed measure. The first PC loads heavily on *negative, weak, fail, fall*, so Tetlock interprets it as a *pessimism* index.

- Pessimism predicts low short-term returns on the Dow.
- The effect *reverses* over a few weeks: news-driven mispricing, not fundamentals.
- This is the template for dictionary-based finance text analysis.

Tetlock, P. (2007), "Giving Content to Investor Sentiment", *Journal of Finance* 62(3).

Loughran & McDonald (2011) — domain matters

Problem. Following Tetlock, people grab the Harvard negative list and apply it to 10-Ks. But that list contains *tax, cost, capital, liability, vice* — words that are neutral or even positive in a financial-reporting context.

Loughran–McDonald fix.

- Build a finance-specific lexicon by inspecting 10-Ks directly.
- Negative list: *restated, litigation, termination, unpaid, investigation, …*
- Six lists: negative, positive, uncertainty, litigious, modal-strong, modal-weak.
- Public release: `https://sraf.nd.edu/loughranmcdonald-master-dictionary/`.

Result. The context-specific dictionary has substantially greater predictive power for return regressions than the generic Harvard one. Today the LM dictionary is the default in finance.

What does a downloaded 10-K look like?

Downloaded example. Apple Inc. 2023 Form 10-K from SEC EDGAR: `aapl-20230930.htm`.

Why it matters. The raw data is not a clean spreadsheet. It is HTML plus XBRL tags, tables, section headings, page markers, and boilerplate.

```
<span>Item 1A. Risk Factors</span>
<span>The Company is subject to various risks...</span>
<ix:nonNumeric name="dei:DocumentFiscalYearFocus">2023</ix:nonNumeric>
<ix:nonNumeric name="dei:EntityCentralIndexKey">0000320193</ix:nonNumeric>
```

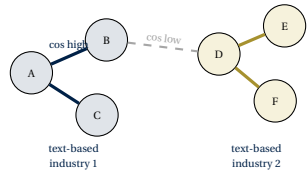
Source: SEC EDGAR, Apple Inc. 10-K, accession `0000320193-23-000106`.

Hoberg & Phillips (2016) — endogenous industries

Question. Are SIC/NAICS classifications too coarse for measuring product-market competition?

Data. 10-K business descriptions for ~ 50,000 firm-years.

Method. Build a TF-IDF vector for each firm's business description, then compute pairwise cosine similarity.



- Output: *text-based industry classifications* (TNIC), available at `hobergphillips.tuck.dartmouth.edu`.
- Predicts profitability, M&A, and pricing power far better than SIC.

Hoberg & Phillips, "Text-Based Network Industries", *JPE* 124(5), 2016. Schematic of the paper's cosine-similarity/clustering idea.

A.5 Beyond Token Representation

Document representation and classification

Given word embeddings $\{x_v\}$, build a document embedding by aggregating:

- **Simple average.** $\bar{x}_d = \frac{1}{T_d} \sum_t x_{w_t}$. Surprisingly hard to beat for short documents.
- **TF-IDF-weighted average.** Give rare informative words more weight.
- **Arora et al. (2017).** Weighted average $\frac{a}{a+\text{freq}(w_t)} x_{w_t}$, then subtract the projection onto the first principal component across all documents (removes a "syntax" direction shared by all sentences).

Classifier. Stack a softmax head: $P(\text{class} = k \mid \bar{x}_d) = \exp(\beta_k^\top \bar{x}_d) / \sum_{k'} \exp(\beta_{k'}^\top \bar{x}_d)$. Train end-to-end (let embeddings be parameters) or freeze the embeddings and train only $\{\beta_k\}$.

fastText supervised classifier (Bojanowski et al. 2017) does exactly this with subword embeddings.

Embeddings beyond text — shopper baskets

Ruiz, Athey & Blei (2020): apply the embedding idea to consumer goods.

- "Documents" = shopping baskets; "words" = products.
- Learn x_v, y_v for each product such that the probability of v being added to a basket depends on $\theta_d^\top x_v + y_v^\top \bar{x}_{\text{basket so far}}$, where θ_d is consumer preferences.
- Embeddings encode substitution and complementarity:
  - Complementarity if $y_v^\top x_w > 0$ and $y_w^\top x_v > 0$ (baguette ↔ butter).
  - Substitutability if products are exchangeable without complementarity (different ham brands).

Take-away. The distributional hypothesis is not unique to language. Anywhere you have "objects" that co-occur in "contexts", you can learn an embedding. Same recipe works on: papers and authors, papers and citations, songs and playlists, drugs and patients.