NLP

A Complete 10-Chapter Data Science & ML Course

Topics covered:

Text preprocessing, bag-of-words, TF-IDF, word embeddings/word2vec

LSTM sequence models, NER/POS tagging, attention & the road to Transformers

Pretrained embeddings and transfer learning, and a full worked pipeline

Exercises: 30 hands-on scenarios with worked solutions

Format: A4 · Dark-theme code examples

Course 4 of 6 in the Data Science & ML subject

Philip Osztromok · Generated with Claude

Table of Contents

CHAPTER 1 OF 10

What NLP Actually Is & The Text Preprocessing Pipeline

NLP

Chapter 1 · What NLP Actually Is & The Text Preprocessing Pipeline

`ds1` taught data handling, `ml1` taught classical models, `nn1` taught neural architectures — all three, so far, applied to numbers and categories. This course applies all of it to **text**, a genuinely messier, more ambiguous kind of data than anything the first three courses ever handed you.

Why Text Is ds1-4's Own Data-Cleaning Material, Taken to Its Extreme

`ds1-4` covered missing values, duplicates, inconsistent formatting, and string cleaning — real problems, but confined to a handful of structured columns with a known, fixed shape. A single paragraph of raw text has every one of those same problems *and* none of the fixed structure: no defined length, no fixed vocabulary, genuine ambiguity (the same word meaning different things in different contexts), and grammar and word order that carry real information a plain column of numbers never has to represent at all.

The Text Preprocessing Pipeline

1. **Tokenization** — splitting raw text into individual units.
2. **Lowercasing** — normalizing case.
3. **Stopword removal** — dropping very common, low-information words.
4. **Stemming or lemmatization** — reducing words to a base form.

Tokenization

```
"I don't think it's working." → ["I", "do", "n't", "think", "it", "'s", "working", "."]
```

Splitting text into **tokens** — usually words, sometimes subwords or characters — is the first, foundational step everything else builds on.

NOT A SOLVED, TRIVIAL PROBLEM EVERYWHERE

Contractions like `don't` already force a real decision (one token or two?). Languages without whitespace between words at all — Chinese, Japanese — need a genuinely different tokenization approach entirely. Tokenization looks trivial in English and stops looking trivial the moment the assumptions behind it are questioned even slightly.

Lowercasing

Treating `"The"` and `"the"` as the same token reduces vocabulary size and noise — genuinely useful for most tasks, and a real, honest trade-off worth naming: `"US"` (the country) and `"us"` (the pronoun) collapse into the same token once lowercased, quietly discarding a real distinction some tasks would actually need.

Stopword Removal

Words like `"the"`, `"a"`, `"is"` appear constantly and usually carry little distinguishing information — removing them reduces noise for many tasks.

A REAL, IMPORTANT GOTCHA — "NOT" IS A STOPWORD TOO

A standard stopword list typically includes negation words like `"not"`. Removing it from `"I do not like this movie"` leaves something dangerously close to the opposite sentiment. For any task where negation genuinely matters — sentiment analysis very much included — indiscriminate stopword removal can silently flip a sentence's own meaning. This isn't a hypothetical edge case; it's a real, well-documented trap.

Stemming vs. Lemmatization

	HOW IT WORKS	EXAMPLE
Stemming	Crude, rule-based chopping of common suffixes — fast, no dictionary needed	"running" → "run" — but an aggressive stemmer can produce non-words, e.g. "running" → "runn"
Lemmatization	Dictionary/grammar-aware reduction to a genuine root word — slower, more accurate	"better" → "good" — something pure rule-based stemming could never do at all

What This Pipeline Doesn't Do Yet

After all four steps, you have a cleaner set of tokens — still just words, not numbers. Every model this course eventually uses, from `ml1`'s own classifiers to `nn1`'s own networks, needs numeric input. `nlp1-2` covers the first real way to get there.

This Course's Own Roadmap

CHAPTER	DELIVERS
nlp1-2 / nlp1-3	Turning cleaned text into numbers a model can use

CHAPTER	DELIVERS
nlp1-4	What that numeric representation still loses
nlp1-5 / nlp1-6	Fixing meaning (embeddings) and order (sequence models) separately
nlp1-7	A real, practical NLP task — entity/tag recognition
nlp1-8 / nlp1-9	Why attention mattered for NLP specifically, and pretrained models in practice
nlp1-10	A real pipeline, and why LLMs are a genuinely different approach

Hands-On Exercises

EXERCISE 1

Explain, using this chapter's own comparison to ds1-4, why text is described as data cleaning's "most extreme case yet" rather than simply a different kind of data.

View solution

EXERCISE 2

Using this chapter's own warn-box, explain concretely why removing "not" as a stopword is a real, serious risk for a sentiment-analysis task specifically, with a worked example.

View solution

EXERCISE 3

Using this chapter's own "better" → "good" example, explain why lemmatization can do something stemming structurally cannot, and identify what stemming would most plausibly produce for the same word instead.

View solution

CHAPTER 1 QUICK REFERENCE

- Text is ds1-4's own data-cleaning problem, with no fixed structure and genuine ambiguity added on top
- **Tokenization** — splitting text into units; genuinely hard in some languages, not just an English afterthought
- **Lowercasing** — reduces noise, but can collapse real distinctions ("US" vs. "us")
- **Stopword removal** — reduces noise, but can remove negation words like "not," silently flipping meaning
- **Stemming** (crude, rule-based) vs. **lemmatization** (dictionary-aware, can map "better" → "good")

- Cleaned tokens still aren't numbers — **Next chapter:** Bag-of-Words & Text Vectorization

CHAPTER 2 OF 10

Bag-of-Words & Text Vectorization

NLP

Chapter 2 · Bag-of-Words & Text Vectorization

`nlp1-1` left off with clean tokens — still just words. This chapter turns them into the one thing every model since `ml1-3` has actually required: numbers.

The Bag-of-Words Idea

Build a **vocabulary** — every unique word across the whole corpus. Represent each document as a vector with one position per vocabulary word, where the value is how many times that word appears in *that* document. "Bag" because order is discarded entirely — as if every word were tipped out of the document into an unordered bag and only the counts were kept.

```
Doc 1: "free money now"
Doc 2: "call me now"

Vocabulary: [call, free, me, money, now]

Doc 1 vector: [0, 1, 0, 1, 1]
Doc 2 vector: [1, 0, 1, 0, 1]
```

Feeding This Directly Into ml1-5's Own Logistic Regression

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.linear_model import LogisticRegression

vectorizer = CountVectorizer()
X = vectorizer.fit_transform(documents)   # exactly the "X" ml1-3/ml1-5 already used
y = labels                                # spam / not spam

model = LogisticRegression()
model.fit(X, y)                            # ml1-5's own workflow, unchanged
```

TEXT IS NOW JUST TABULAR DATA

Each vocabulary word is now literally a column, exactly like `ml1-3`'s own `mileage` and `year` columns, or `ml1-5`'s own `salary` and `age`. Nothing about `LogisticRegression.fit()` changes at all — it has no idea the columns started life as words rather than numeric measurements.

Two Real, Practical Details

Sparse Vectors

A real corpus can easily have a vocabulary of tens of thousands of unique words — but any single document only ever uses a tiny fraction of them. The resulting vectors are overwhelmingly zeros, a property called **sparsity**. This is a genuinely different practical concern from `ds1`'s own typically dense numeric tables, though most libraries (including `CountVectorizer`) store and compute with sparse vectors efficiently rather than wastefully storing every zero.

Words Never Seen During Training

The vocabulary is built from the training data. A brand-new document at prediction time containing a word the vocabulary has never seen simply has nowhere to put it — that word is typically dropped entirely, silently. A real, honest limitation worth naming plainly.

What This Chapter Deliberately Doesn't Fix

ORDER IS GONE — ON PURPOSE, FOR NOW

Nothing about a bag-of-words vector distinguishes `"dog bites man"` from `"man bites dog"` — both produce the identical set of word counts. This isn't an oversight; `nlp1-4` demonstrates exactly why this matters, concretely, before `nlp1-5` and `nlp1-6` separately fix it.

Hands-On Exercises

EXERCISE 1

Using this chapter's own two-document example, build the bag-of-words vector for a third document, "free call now," and explain how you determined each position's value.

View solution

EXERCISE 2

Explain, using this chapter's own finding-box, precisely why LogisticRegression.fit() requires no code changes at all to work with bag-of-words vectors instead of ml1-3's own numeric features.

View solution

EXERCISE 3

Explain what sparsity means for a bag-of-words vector, and explain why a real corpus's own vocabulary size makes sparsity essentially unavoidable rather than a rare edge case.

View solution

CHAPTER 2 QUICK REFERENCE

- **Bag-of-words** — one vector position per vocabulary word, value = count in that document, order discarded entirely
- Feeds directly into ml1-5's own `LogisticRegression.fit()` — text becomes ordinary tabular data, one column per word
- **Sparsity** — real vocabularies are huge; any one document's own vector is overwhelmingly zeros
- Words unseen during training simply can't be represented at prediction time — dropped silently
- Word order is completely lost — deliberately left unresolved, demonstrated concretely in **nlp1-4**
- **Next chapter:** TF-IDF — Weighting Words by Distinctiveness

CHAPTER 3 OF 10

TF-IDF — Weighting Words by Distinctiveness

NLP

Chapter 3 · TF-IDF — Weighting Words by Distinctiveness

`nlp1-2`'s own raw counts treat every word's occurrence as equally meaningful. It isn't — a word appearing constantly across every document tells you almost nothing about what makes any *one* of them distinctive.

Why Raw Counts Alone Are Misleading

A word like `"now"` might appear often in a spam message — and just as often in an ordinary one. Its raw count is high, but it does little to actually distinguish spam from not-spam. A genuinely rare word like `"viagra"`, by contrast, might appear only once — but that single occurrence is far more informative about what kind of message this is.

TF — Term Frequency

How often a word appears *within this document*, typically normalized by the document's own total word count — `ds1-6`'s own proportion vocabulary, applied directly: not a raw count, but a rate, so a short and a long document can be compared fairly.

```
TF(word, doc) = (count of word in doc) / (total words in doc)
```

IDF — Inverse Document Frequency

How *rare* a word is across the whole corpus. A word appearing in nearly every document gets a low IDF (not distinctive); a word appearing in only a handful gets a high IDF (distinctive).

```
IDF(word) = log( total documents / documents containing word )
```

The logarithm is a deliberate dampener — without it, an extremely rare word's own score could grow disproportionately large and dominate everything else; the log keeps the scale reasonable while still rewarding rarity.

TF-IDF = TF × IDF

A word scores **high** when it's frequent in *this* document *and* rare across the corpus overall — genuinely distinctive for this specific document. It scores **low** when it's either rare here or common everywhere.

WORD	FREQUENT IN THIS DOC?	COMMON ACROSS CORPUS?	TF-IDF SCORE
"now" (appears in most messages)	Yes	Yes	Low
"viagra" (appears rarely, distinctively)	Yes, here	No	High

BUILT ENTIRELY FROM DS1-6'S OWN VOCABULARY

TF is a rate — the exact "proportion" concept `ds1-6` already defined. IDF's own document-count ratio is the same "how common is this across the whole population" reasoning `ds1-6` used for its own statistics, just applied to documents containing a word rather than data points sharing a value.

In Practice — A Drop-In Replacement

```
from sklearn.feature_extraction.text import TfidfVectorizer

vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(documents)   # same shape, same downstream ml1-5 workflow
```

`TfidfVectorizer` slots directly into `nlp1-2`'s own pipeline in place of `CountVectorizer` — everything downstream, including `ml1-5`'s own `LogisticRegression.fit()`, is unchanged.

What TF-IDF Does Not Fix

STILL BAG-OF-WORDS UNDERNEATH

TF-IDF only changes how strongly each word's own count is *weighted* — it's still fundamentally a bag-of-words representation, and `nlp1-2`'s own word-order loss applies completely unchanged. `"dog bites man"` and `"man bites dog"` still produce identical TF-IDF vectors — the exact same words, the exact same counts, just each one now weighted by distinctiveness rather than left as a raw count. TF-IDF solves one narrow problem (unequal word importance); it does nothing at all for the order problem `nlp1-4` covers next.

Hands-On Exercises

EXERCISE 1

Using this chapter's own IDF formula, explain why a word appearing in every single document in a corpus receives an IDF score of exactly zero, and explain what that means for its overall TF-IDF score regardless of how frequently it appears in any one document.

View solution

EXERCISE 2

Explain, using this chapter's own comparison table, why "viagra" and "now" can both be frequent within a specific document yet receive dramatically different TF-IDF scores.

View solution

EXERCISE 3

Explain, using this chapter's own warn-box, why "dog bites man" and "man bites dog" still produce identical TF-IDF vectors, and explain precisely what problem TF-IDF solves versus what problem it leaves completely untouched.

View solution

CHAPTER 3 QUICK REFERENCE

- **TF** — a word's own rate within one document (ds1-6's own proportion vocabulary)
- **IDF** — log(total docs / docs containing the word) — rarer across the corpus means a higher score
- **TF-IDF = TF × IDF** — high only when frequent here *and* rare elsewhere
- `TfidfVectorizer` — a drop-in replacement for nlp1-2's own `CountVectorizer`, same downstream ml1-5 workflow
- Still bag-of-words underneath — word order is completely unaffected; "dog bites man" = "man bites dog," unchanged from nlp1-2
- **Next chapter:** The Limits of Bag-of-Words

CHAPTER 4 OF 10

The Limits of Bag-of-Words

NLP

Chapter 4 · The Limits of Bag-of-Words

`nlp1-2` and `nlp1-3` both flagged the same loss and moved on. This chapter stops and actually proves it, then reveals a second, entirely separate blind spot neither chapter mentioned at all.

Problem 1: Word Order — Proven, Not Just Asserted

Vocabulary: `[bites, dog, man]`.

```
"dog bites man" → [1, 1, 1]
"man bites dog" → [1, 1, 1]
```

Identical vectors. Not approximately similar — *identical*, down to the last digit, whether built with `nlp1-2`'s own raw counts or `nlp1-3`'s own TF-IDF weights. A model trained on either representation has no way whatsoever to distinguish these two sentences, because as far as its own input is concerned, they aren't two sentences at all — they're the exact same input, twice.

WHY JOURNALISM HAS A NAME FOR THIS EXACT INVERSION

"Dog bites man" is famously described in journalism as *not* news — ordinary, unremarkable. "Man bites dog" is the same three words, reordered, describing something genuinely surprising and newsworthy. A representation that treats these as identical isn't failing at some minor technicality — it's failing at exactly the distinction a human reader would consider most important.

Problem 2: No Notion of Meaning — A Genuinely Separate Issue

Every vocabulary word is its own independent, arbitrary dimension. `"good"` and `"great"` are just two unrelated vector positions — as far as the representation itself is concerned, `"good"` is exactly as similar to `"great"` as it is to `"car"` or `"purple"`. Nothing about bag-of-words or TF-IDF encodes synonymy or semantic closeness at all.

A CONCRETE, PRACTICAL CONSEQUENCE

A sentiment classifier trained mostly on reviews using the word `"great"` has no built-in way to recognize `"excellent"` as equally positive if `"excellent"` rarely appeared during training — the model can only generalize from the literal, specific words it happened to see, never from what those words actually mean.

This also reframes `nlp1-2`'s own out-of-vocabulary limitation as a direct symptom of this deeper problem, not just an annoying edge case: a genuinely new word like `"excellent"` gets silently dropped rather than recognized as close in meaning to `"great,"` precisely because nothing in the representation has any concept of "close in meaning" to begin with.

Two Separate Problems, Two Separate Fixes

PROBLEM 1

Word Order

Fixed by **nlp1-6** — sequence models, revisiting `nn1-8`'s own RNN/LSTM material, applied to text for real.

PROBLEM 2

Word Meaning

Fixed by **nlp1-5** — word embeddings, representing words as dense vectors positioned by genuine semantic closeness.

NEITHER FIX ALONE SOLVES BOTH PROBLEMS

Fixing meaning doesn't fix order, and fixing order doesn't fix meaning — they're genuinely independent failures with genuinely independent solutions. Even after `nlp1-5`'s own embeddings arrive, order remains completely unaddressed until `nlp1-6` adds a sequence model on top.

Hands-On Exercises

EXERCISE 1

Using this chapter's own vocabulary and vectors, explain precisely why "dog bites man" and "man bites dog" produce mathematically identical vectors, not merely similar ones, under both nlp1-2's and nlp1-3's own representations.

View solution

EXERCISE 2

Explain why this chapter reframes nlp1-2's own out-of-vocabulary limitation as "a direct symptom" of the meaning problem rather than a separate, unrelated issue.

View solution

EXERCISE 3

Explain why this chapter insists the order problem and the meaning problem require two genuinely separate fixes, using this chapter's own warn-box, and explain what would still be broken if only nlp1-5's own embeddings were applied without nlp1-6's own sequence modeling.

View solution

CHAPTER 4 QUICK REFERENCE

- **Problem 1 — Word order:** "dog bites man" and "man bites dog" produce mathematically identical vectors under both nlp1-2 and nlp1-3
- **Problem 2 — Word meaning:** every word is its own unrelated dimension — "good" and "great" are as unrelated as "good" and "car"
- Out-of-vocabulary words (nlp1-2) are a direct symptom of the meaning problem, not a separate edge case
- Two genuinely independent problems, two genuinely independent fixes — neither alone solves both
- **Next chapter:** Word Embeddings & Word2Vec (fixes meaning) · **nlp1-6** fixes order

CHAPTER 5 OF 10

Word Embeddings & Word2Vec

NLP

Chapter 5 · Word Embeddings & Word2Vec

`nlp1-4` diagnosed Problem 2: every word is its own isolated, unrelated dimension. This chapter fixes it — and reveals that the tool doing the fixing is something you already know how to build.

Word Embeddings — The Core Idea

Instead of a huge, sparse vector with one arbitrary dimension per vocabulary word, represent each word as a small, **dense** vector — commonly 100–300 numbers — learned so that semantically similar words end up genuinely *close together* in that vector space. `nlp1-4` 's own example, resolved directly: in embedding space, `"good"` and `"great"` sit near each other; `"good"` and `"car"` sit far apart. A real, learned, geometrically meaningful relationship — not an arbitrary one.

Word2Vec — Learning This From Raw Text Alone

Word2vec learns embeddings **self-supervised** — no manual labeling at all, purely from patterns in how words co-occur across a huge corpus of ordinary, unlabeled text. Two real training setups:

APPROACH	PREDICTS
CBOW	The target word, from its surrounding context words
Skip-gram	The surrounding context words, from the target word

A DELIBERATE FORWARD-POINTER TO LLM1

The "labels" here are just other words already present in the raw text — the task supervises itself. This is conceptually the same underlying idea (though vastly different in scale and architecture) as the self-supervised "predict the next token" pretraining `llm1` covers for large language models. Word2vec is a genuine, small-scale ancestor of that approach, not an unrelated technique.

Word2Vec Is Literally nn1-1's Own Neuron, Generalized

Word2vec's own architecture is a small feedforward network, built entirely from `nn1-1`'s own vocabulary: an input layer (a one-hot encoded word), a single hidden layer, and an output layer predicting the context or target word.

THE DETAIL MOST EXPLANATIONS SKIP

The embedding isn't some separate output the network produces — it *is* the hidden layer's own learned weight matrix, once training finishes. Look up a word's embedding, and you're literally reading off one row of weights this small network learned via `nn1-5`'s own backpropagation. Nothing new was needed to build word2vec — it's `nn1-1`'s own generalized layer stack, applied to exactly this self-supervised prediction task.

The Geometric Payoff — Honestly Framed

Trained on a large enough corpus, the resulting embedding space captures genuine semantic and syntactic relationships well enough that vector arithmetic becomes meaningful: the famous `king - man + woman ≈ queen`.

A REAL PROPERTY, PRESENTED HONESTLY

This is a genuine, documented, widely-cited illustration — not a fabricated demo. It's also a deliberately clean, popular example chosen because it works well for illustration. Real embeddings capture many subtler relationships beyond this one case, but the arithmetic doesn't hold with the same clean precision for every possible word relationship in practice. Treat it as a genuine, illustrative property of the learned space, not a guarantee that this exact kind of arithmetic works perfectly and universally.

In Practice — Pretrained, Not Trained From Scratch

Training word2vec from scratch needs a genuinely large corpus. In practice, pretrained embeddings — trained once, on massive text collections, by someone else — are typically used directly rather than retrained per project. `nlp1-9` covers this properly; for now, know that it's the normal way embeddings actually get used.

What's Still Broken

`nlp1-4`'s meaning problem is solved. Its *other* diagnosed problem — word order — is completely untouched by anything in this chapter. Averaging or otherwise combining word embeddings without regard to their sequence still loses order entirely. `nlp1-6` is where that gets fixed.

Hands-On Exercises

EXERCISE 1

Explain the difference between CBOW and skip-gram, and explain why both count as genuinely self-supervised despite requiring no human-provided labels at all.

View solution

EXERCISE 2

Using this chapter's own finding-box, explain precisely what a word's embedding actually is inside word2vec's own network, and explain why this chapter calls it "nn1-1's own generalized layer stack" rather than a new kind of model.

View solution

EXERCISE 3

Explain why this chapter presents "king − man + woman ≈ queen" with an explicit caveat rather than as unqualified proof that embeddings perfectly capture all semantic relationships.

View solution

CHAPTER 5 QUICK REFERENCE

- **Word embeddings** — dense, small vectors where semantic closeness becomes geometric closeness, fixing nlp1-4's meaning problem
- **Word2vec** — self-supervised: CBOW predicts a word from context, skip-gram predicts context from a word; no manual labeling — a genuine small-scale ancestor of llm1's own next-token pretraining
- The embedding *is* the hidden layer's own weight matrix — word2vec is literally nn1-1's own neuron/layer stack, generalized
- "king − man + woman ≈ queen" is real but illustrative — not a guarantee of perfectly clean arithmetic everywhere
- Pretrained embeddings are the normal path in practice — full coverage in nlp1-9
- Meaning: solved. Order: still completely unaddressed — **Next chapter:** Sequence Models for Text

CHAPTER 6 OF 10

Sequence Models for Text

NLP

Chapter 6 · Sequence Models for Text

`nlp1-5` fixed meaning. This chapter fixes the other half of `nlp1-4`'s own diagnosis — word order — by finally putting `nn1-8`'s own RNN/LSTM material to real, concrete use on real text.

The Real Pipeline, End to End

1. Raw text → `nlp1-1`'s own preprocessing (tokenize, clean).
2. Each token → `nlp1-5`'s own word embedding.
3. The sequence of embeddings, *in order*, fed one at a time into an RNN/LSTM (`nn1-8`).
4. The LSTM's own final hidden state — a compressed summary of the whole sequence — fed into a small classifier head.

Resolving nlp1-4's Own Cliffhanger, Concretely

`nn1-8`'s own hidden-state update at every step depends on *both* the current input *and* the previous hidden state — order-dependent by construction. Feed the embeddings for `"dog"`, `"bites"`, `"man"` into the LSTM in that exact order, and the resulting final hidden state is genuinely, mathematically different from feeding `"man"`, `"bites"`, `"dog"` — because at every intermediate step, the hidden state carries forward a different history depending on what came before it.

SOMETHING BAG-OF-WORDS AND PLAIN EMBEDDINGS ALONE COULD NEVER DO

`nlp1-2` / `nlp1-3`'s own bag-of-words vectors were provably identical for both sentences (`nlp1-4`). Averaging `nlp1-5`'s own embeddings together, with no sequence model, would *also* produce identical results for both — averaging, like counting, has no notion of position either. Only combining meaning-aware embeddings *with* an order-preserving sequence model — exactly this chapter's own pipeline — genuinely distinguishes the two sentences.

A Real Worked Task — Sentiment Classification

```
import torch.nn as nn

class TextClassifier(nn.Module):
    def __init__(self, vocab_size, embed_dim, hidden_dim):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embed_dim)   # nlp1-5's own embeddings
        self.lstm = nn.LSTM(embed_dim, hidden_dim, batch_first=True)   # nn1-8
        self.output = nn.Linear(hidden_dim, 1)                  # nn1-1's own closing callback,
again

    def forward(self, x):
        embedded = self.embedding(x)              # one embedding per token, in order
        _, (hidden, _) = self.lstm(embedded)        # the final hidden state — nn1-8's own "memory"
of the whole sequence
        return torch.sigmoid(self.output(hidden[-1]))   # ml1-5's own logistic regression, one more
time
```

PRETRAINED EMBEDDINGS SLOT RIGHT IN

`nn.Embedding`'s own weight matrix can be initialized directly from `nlp1-5`'s own pretrained word2vec vectors rather than learned from scratch — a direct preview of `nlp1-9`'s own transfer-learning material.

Both Problems, Genuinely Solved Together

`nlp1-4` diagnosed two independent problems — meaning and order — and insisted neither fix alone solves both. This chapter's own pipeline is the actual combination: `nlp1-5`'s embeddings supply meaning, this chapter's own sequence model supplies order. Only together do they produce a representation that genuinely distinguishes `"dog bites man"` from `"man bites dog"` *and* recognizes `"excellent"` as similar to `"great."`

What's Next

This chapter's own model produces exactly *one* output for an entire sequence — a single sentiment label per document. `nlp1-7` extends the identical underlying idea to produce *one output per token* instead — real, practical tasks like recognizing names and grammatical roles word by word.

Hands-On Exercises

EXERCISE 1

Explain, using nn1-8's own hidden-state mechanism, why feeding "dog," "bites," "man" into an LSTM in that order produces a genuinely different final hidden state than feeding "man," "bites," "dog."

View solution

EXERCISE 2

Using this chapter's own finding-box, explain why simply averaging nlp1-5's own word embeddings together, without a sequence model, would still fail to distinguish "dog bites man" from "man bites dog."

View solution

EXERCISE 3

Explain why this chapter says nlp1-4's two diagnosed problems are only genuinely solved by combining nlp1-5's own embeddings with this chapter's own sequence model, rather than by either technique alone.

View solution

CHAPTER 6 QUICK REFERENCE

- Real pipeline: preprocessing (nlp1-1) → embeddings (nlp1-5) → LSTM in sequence order (nn1-8) → classifier head (nn1-1/ml1-5)
- Order-dependence follows directly from nn1-8's own hidden-state update depending on both current input and prior state
- Genuinely resolves nlp1-4's "dog bites man" cliffhanger — bag-of-words and averaged embeddings both provably couldn't
- Both of nlp1-4's problems (meaning + order) are only solved by this combined pipeline, not by either fix alone
- Pretrained embeddings slot directly into `nn.Embedding` — previewing nlp1-9
- **Next chapter:** Named Entity Recognition & Part-of-Speech Tagging

CHAPTER 7 OF 10

Named Entity Recognition & Part-of-Speech Tagging

NLP

Chapter 7 · Named Entity Recognition & Part-of-Speech Tagging

`nlp1-6`'s own model produced exactly one output for an entire sequence. This chapter needs one output *per token* instead — and the architectural change required is smaller than it sounds.

Two Real, Practical Tasks

Named Entity Recognition (NER) — identifying and classifying named entities in text:

SARAH	WORKS	AT	GOOGLE	IN	LONDON	.
PER	O	O	ORG	O	LOC	O

Part-of-Speech (POS) tagging — assigning each word its grammatical role:

SARAH	WORKS	AT	GOOGLE	IN	LONDON	.
NOUN	VERB	ADP	PROPN	ADP	PROPN	PUNCT

Both are **sequence labeling** tasks — a genuine, distinct category from `nlp1-6`'s own **sequence classification**: one label for every token, not one label for the whole sequence.

The Architectural Change — Smaller Than It Sounds

`nlp1-6`'s own model discarded every hidden state except the very last one, using it as a single summary of the entire sequence. Sequence labeling keeps *every* hidden state — one per time step — and feeds each one individually through the same small classifier head, producing one prediction per token instead of one prediction total.

```
class SequenceLabeler(nn.Module):
    def __init__(self, vocab_size, embed_dim, hidden_dim, num_tags):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embed_dim)
        self.lstm = nn.LSTM(embed_dim, hidden_dim, batch_first=True, bidirectional=True)
        self.output = nn.Linear(hidden_dim * 2, num_tags)
```

```
    def forward(self, x):
        embedded = self.embedding(x)
        outputs, _ = self.lstm(embedded)     # every hidden state, not just the last (nlp1-6's own change)
        return self.output(outputs)          # one prediction per token, via softmax over num_tags
```

SOFTMAX, NOT NLP1-6'S OWN SIGMOID

`nlp1-6`'s sentiment task was binary — one sigmoid-activated neuron, exactly `nn1-1`'s own logistic regression. NER and POS tagging are usually genuine multi-class problems (several possible entity types or grammatical roles per token) — the output layer here needs `num_tags` outputs with a softmax, not a single sigmoid. A real, honest distinction worth naming rather than silently reusing the wrong output layer.

Bidirectional LSTMs — Why "Later" Words Help Too

`"Washington"` could be a person's surname or a place — often only resolvable by what comes *after* it in the sentence, not before. `nn1-8`'s own plain LSTM only ever sees what came earlier. A **bidirectional LSTM (BiLSTM)** runs two LSTMs over the same sequence — one forward, one backward — and concatenates both hidden states at every position, so each token's own final representation genuinely reflects the whole sentence, not just its own left-hand context.

Hands-On Exercises

EXERCISE 1

Explain the specific architectural difference between this chapter's own sequence-labeling model and nlp1-6's own sequence-classification model, using this chapter's own code and nlp1-6's own code to identify exactly what changed.

View solution

EXERCISE 2

Explain why this chapter's own output layer needs softmax over multiple tags rather than nlp1-6's own single sigmoid neuron, and explain what would go wrong if the sigmoid approach were reused unchanged for NER.

View solution

EXERCISE 3

Using this chapter's own "Washington" example, explain why a plain, forward-only LSTM (nn1-8) can genuinely struggle with NER specifically, and explain what a bidirectional LSTM actually adds to fix it.

View solution

CHAPTER 7 QUICK REFERENCE

- **NER** — classify named entities (PER/ORG/LOC/...) · **POS tagging** — classify grammatical role, both per-token
- **Sequence labeling** (one label per token) vs. nlp1-6's own **sequence classification** (one label per sequence)
- The change: keep every hidden state instead of discarding all but the last, and feed each one through the classifier head
- Softmax over multiple tags, not nlp1-6's own binary sigmoid — a genuine, honest distinction
- **Bidirectional LSTM** — forward + backward passes concatenated, so later context can disambiguate earlier tokens too
- **Next chapter:** From RNNs to Attention — NLP's Own Motivation for Transformers

CHAPTER 8 OF 10

From RNNs to Attention — NLP's Own Motivation for Transformers

NLP

Chapter 8 · From RNNs to Attention — NLP's Own Motivation for Transformers

`nn1-9` previewed self-attention conceptually and deliberately deferred full depth to `llm1`. This chapter doesn't skip ahead of that plan — it grounds the preview in the real, historically accurate problem attention was invented to solve, which happens to be an NLP problem: **machine translation**.

Seq2Seq: nlp1-6's Own Architecture, Turned Toward Generation

Translating a sentence is naturally framed as sequence-to-sequence: an **encoder** LSTM reads the entire source sentence, and a **decoder** LSTM generates the target sentence word by word. Look closely at the encoder's own job, though — it is doing exactly what `nlp1-6`'s classifier did: processing a whole sequence and compressing it down into a *single final hidden state*. The only difference here is what happens to that summary afterward — instead of feeding it to a sigmoid, it's handed to a second LSTM as its own starting point.

MODEL	WHAT THE FINAL HIDDEN STATE IS USED FOR
nlp1-6's sentiment classifier	Fed through a classifier head to produce one label
Seq2seq encoder	Fed to a decoder LSTM as its own initial hidden state

The Fixed-Vector Bottleneck

For a short sentence, squeezing everything into one vector works reasonably well. For a long sentence, it's a genuine problem: the encoder's final hidden state has to somehow carry *everything* the decoder will need — every noun, every clause, every dependency — through one fixed-size vector. Early words in a long sentence get progressively diluted by every step that comes after them, the same accumulating-history mechanism `nlp1-6` relied on to preserve order now works against fidelity at length.

A REAL, MEASURED EFFECT

Translation quality measurably degrades as source-sentence length increases in a plain encoder-decoder setup — not a theoretical concern, but the actual, documented motivation researchers were responding to.

Attention: Let the Decoder Look Back

Bahdanau's 2014 fix (pre-dating the 2017 Transformer paper by three years) was simple in concept: instead of forcing the decoder to work from *one* final encoder hidden state, let it look back at **every** encoder hidden state at every step it generates, and compute a weighted combination — attending more heavily to whichever source words are most relevant to the word being generated right now.

```
# conceptual attention score at one decoder step
scores = [dot(decoder_state, encoder_state_i) for encoder_state_i in encoder_states]
weights = softmax(scores)                    # how much to "attend" to each source word
context = sum(w * s for w, s in zip(weights, encoder_states))   # weighted blend, not one fixed
vector
```

Translating "the black cat" into French, the decoder generating "*noir*" (black) would learn to assign a high attention weight to the source word "black" and comparatively little to "the" or "cat" — a direct, interpretable link between output and source, something the single fixed vector could never expose.

Self-Attention: The 2017 Leap

The real leap in "Attention Is All You Need" (2017) — the paper `historyai3-6` already namechecked — was applying this exact same mechanism *within* a single sequence, not just from decoder back to encoder. In **self-attention**, every token computes attention weights against every *other* token in the same sequence, building a representation of each word informed by every other word it's relevant to — no recurrence required at all.

> **RESOLVING NN1-9'S OWN NAMED BOTTLENECK**
>
> `nn1-8`'s LSTM must process tokens one at a time, in strict sequence — inherently serial, exactly the limitation `nn1-9` flagged. Self-attention computes every pairwise token relationship *at once*, which is why it's genuinely parallelizable in a way recurrence structurally cannot be.

What's Still Deferred

`nn1-9` already flagged this honestly, and it's still true here: self-attention has no inherent sense of token order — swap two tokens and the set of pairwise relationships computed is identical. A real Transformer needs **positional encoding** to reintroduce order information, and the full multi-head, multi-layer Transformer architecture built from self-attention is genuinely substantial. Both are deliberately left for `llm1`'s own full depth, exactly as planned.

Hands-On Exercises

EXERCISE 1

Explain why a plain seq2seq encoder's own final hidden state is structurally the same idea as nlp1-6's own final hidden state, and explain specifically why this becomes a bigger problem as sentence length grows.

View solution

EXERCISE 2

Using this chapter's own "the black cat" example, explain what attention weights actually represent and why they give the decoder something the single fixed encoder vector could never provide.

View solution

EXERCISE 3

Explain why self-attention resolves nn1-9's own named sequential-computation bottleneck, and explain what specific limitation self-attention still has that a full Transformer must separately address.

View solution

CHAPTER 8 QUICK REFERENCE

- **Seq2seq** — encoder LSTM compresses a sentence into one final hidden state, decoder LSTM generates from it — nlp1-6's own architecture, turned toward generation
- **The bottleneck** — one fixed-size vector must carry an entire sentence's meaning, degrading measurably as sentences get longer
- **Bahdanau attention (2014)** — the decoder attends to every encoder hidden state, weighted by relevance, instead of just the final one
- **Self-attention (2017)** — the same mechanism applied within one sequence; resolves nn1-9's own named serial-computation bottleneck by computing all pairwise relationships at once
- Still deferred to **llm1**: positional encoding, and the full multi-head, multi-layer Transformer
- **Next chapter:** Pretrained Embeddings & Transfer Learning

CHAPTER 9 OF 10

Pretrained Embeddings & Transfer Learning

NLP

Chapter 9 · Pretrained Embeddings & Transfer Learning

`nlp1-5` trained word2vec embeddings from scratch, on whatever corpus happened to be on hand. This chapter asks the obvious next question: what if, instead, someone else already trained embeddings on a corpus vastly larger than any single project could gather — and all that's needed is to reuse them?

The Cold-Start Problem nlp1-5 Quietly Assumed Away

Word2vec's own skip-gram/CBOW training (`nlp1-5`) needs a large corpus and real training time to learn good relationships — "excellent" only ends up near "great" in vector space after seeing both words in enough similar contexts, many times over. A small project's own dataset — the spam examples from `nlp1-2`, or the toy sentences carried through `nlp1-6`/`nlp1-7` — is nowhere near large enough to learn quality embeddings from scratch. Training on too little text produces noisy, unreliable vectors no matter how correct the algorithm is.

GloVe — A Genuinely Different Training Approach

GloVe (Global Vectors, Stanford) is the other major pretrained-embedding family alongside word2vec, and it gets there differently. Word2vec learns from local context windows, one sliding prediction task at a time. GloVe instead builds a full word-by-word co-occurrence matrix across an entire corpus (how often does word A appear near word B, counted globally) and factorizes that matrix directly to produce vectors — a genuinely different mathematical approach reaching a similar kind of representation. Trained on massive corpora (Wikipedia, Common Crawl — billions of words), the resulting vectors are freely downloadable and ready to use without any training of your own.

Transfer Learning — A Pattern, Not Just an NLP Trick

Reusing embeddings trained on one (huge, general) task for a different (smaller, specific) task is an instance of **transfer learning** — a general pattern, not unique to text. `nn1-7`'s own AlexNet, trained on ImageNet's millions of labeled images, produces internal features so generally useful that they get reused as a starting point for entirely different image tasks the network was never trained on. Pretrained word embeddings are the

same idea in a different modality: a representation learned once, on a huge general corpus, reused as a head start rather than relearned from nothing.

Frozen vs. Fine-Tuned

There are two honest ways to use a pretrained embedding layer:

APPROACH	WHAT HAPPENS DURING TRAINING	WHEN IT'S THE RIGHT CALL
Frozen	Pretrained vectors loaded, never updated	Small task-specific dataset — not enough data to safely specialize further
Fine-tuned	Pretrained vectors loaded as a starting point, then updated during training	Enough task-specific data to adapt vectors to this domain without losing what was already learned

```
# frozen — the classic transfer-learning move for a small dataset
embedding = nn.Embedding.from_pretrained(glove_vectors, freeze=True)

# fine-tuned — start from GloVe, keep adapting during training
embedding = nn.Embedding.from_pretrained(glove_vectors, freeze=False)
```

Compare this to `nlp1-5`'s own `nn.Embedding(vocab_size, embed_dim)` — randomly initialized, learned entirely from this project's own limited data. Loading GloVe's own pretrained values instead means training starts from vectors that already encode real semantic relationships, rather than from random noise.

The Honest Limitation: Out-of-Vocabulary Words

A FIXED, PRETRAINED VOCABULARY

GloVe's own vocabulary is fixed at whatever it was trained on. Domain-specific jargon, brand names, typos, or genuinely novel words simply aren't in it — these **out-of-vocabulary (OOV)** words get no meaningful vector at all, typically falling back to a generic "unknown" placeholder that carries none of the real word's own meaning. A pretrained embedding is a head start, not a complete solution for every dataset.

The Bridge Into llm1

THE SAME SHAPE, AT A MUCH LARGER SCALE

This chapter's own frozen-vs-fine-tuned distinction is the exact same shape as the pretrain-then-finetune paradigm `llm1` covers in full — a model trained once on a massive, general task, then either used as-is or further adapted on a smaller, specific one. The only real difference is scale: here it's a lookup table of word vectors; there it's an entire deep

network with billions of parameters. The underlying logic — don't relearn from nothing what's already been learned well elsewhere — is identical.

Hands-On Exercises

EXERCISE 1

Explain why nlp1-5's own from-scratch word2vec training would likely produce noisy, unreliable vectors if trained only on the small spam or toy datasets used earlier in this course, and explain specifically what pretrained embeddings like GloVe fix about this.

View solution

EXERCISE 2

Explain the genuine parallel between this chapter's own use of pretrained GloVe embeddings and nn1-7's own use of AlexNet features trained on ImageNet, and explain why both count as transfer learning despite operating on entirely different kinds of data.

View solution

EXERCISE 3

Explain when freezing pretrained embeddings is the right choice versus fine-tuning them, and explain specifically why this chapter's own frozen-vs-fine-tuned distinction is described as the same underlying shape as llm1's own pretrain-then-finetune paradigm.

View solution

CHAPTER 9 QUICK REFERENCE

- **The cold-start problem** — nlp1-5's own from-scratch training needs a large corpus; small task-specific datasets can't reliably supply one
- **GloVe** — global co-occurrence-matrix factorization, a genuinely different training approach from word2vec, trained on massive corpora and freely downloadable
- **Transfer learning** — reusing a representation learned on a huge general task for a smaller specific one; nn1-7's ImageNet-trained AlexNet features are the same pattern in a different modality
- **Frozen** — pretrained vectors never updated, best for small datasets · **Fine-tuned** — pretrained vectors continue updating, best with enough task-specific data
- **OOV words** — domain jargon/novel words missing from GloVe's fixed vocabulary get no meaningful vector

- This chapter's frozen-vs-fine-tuned split is the same shape, at far smaller scale, as **llm1**'s own pretrain-then-finetune paradigm
- **Next chapter:** Capstone — Building a Real NLP Pipeline & Why LLMs Are Different

CHAPTER 10 OF 10

Capstone: Building a Real NLP Pipeline & Why LLMs Are Different

NLP

Chapter 10 · Capstone: Building a Real NLP Pipeline & Why LLMs Are Different

Nine chapters, each solving one specific, honestly-diagnosed problem. This capstone assembles the pieces that actually survive into a real, working sentiment-classification pipeline — then closes with the question this whole course has been quietly building toward: what does an LLM actually do differently?

The Pipeline

```
class SentimentClassifier(nn.Module):
    def __init__(self, glove_vectors, hidden_dim):
        super().__init__()
        # nlp1-9: pretrained GloVe, frozen — this dataset is small, per nlp1-9's own guidance
        self.embedding = nn.Embedding.from_pretrained(glove_vectors, freeze=True)
        # nlp1-6: LSTM sequence model, resolving nlp1-4's order-and-meaning problem together
        self.lstm = nn.LSTM(glove_vectors.shape[1], hidden_dim, batch_first=True)
        # nlp1-6: classifier head reading only the final hidden state — one label per sequence
        self.output = nn.Linear(hidden_dim, 1)

    def forward(self, x):
        embedded = self.embedding(x)              # nlp1-1: x is already tokenized/cleaned text
        _, (hidden, _) = self.lstm(embedded)
        return torch.sigmoid(self.output(hidden[-1]))   # binary sentiment, per nlp1-6
```

Every line here has a specific origin. Text arrives already tokenized and cleaned (`nlp1-1`), is converted to vectors using pretrained, frozen GloVe embeddings rather than training from scratch on this project's own small dataset (`nlp1-9`, resolving the exact cold-start problem that chapter diagnosed), processed in order by an LSTM so word order genuinely affects the result (`nlp1-6`, resolving `nlp1-4`'s own two-problem cliffhanger together with the embeddings), and classified from the final hidden state through a single sigmoid output (`nlp1-6`'s own binary classifier head).

Chapter Attribution Table

CHAPTER	WHAT IT CONTRIBUTED
nlp1-1	Tokenization, stopword removal, lemmatization — the cleaning step every input passes through first
nlp1-2 / nlp1-3	Bag-of-words and TF-IDF — early vectorization approaches, superseded in this pipeline per nlp1-4's own diagnosis, but the first real proof that text could be turned into numbers at all
nlp1-4	Proved bag-of-words loses both word order and word meaning — the two-problem diagnosis this whole pipeline exists to resolve
nlp1-5	Introduced word embeddings and word2vec — the meaning-aware vector idea this capstone uses via nlp1-9's pretrained version instead of training from scratch
nlp1-6	The LSTM sequence-classification architecture used directly in this capstone's own model
nlp1-7	Extended the same architecture to per-token labeling (NER/POS) — a genuinely different pipeline from this capstone's own sequence classifier, not used here
nlp1-8	Attention and self-attention, motivated by machine translation — conceptual foundation only; not implemented in this capstone (see scope note)
nlp1-9	Pretrained GloVe embeddings, used frozen in this capstone's own embedding layer, directly per that chapter's own small-dataset guidance

Why LLMs Are Different

Look honestly at what this course actually built. `nlp1-6`'s sentiment classifier and `nlp1-7`'s NER tagger share the same underlying LSTM mechanism — but they are genuinely **different pipelines**: different output layers, different training objectives, separately trained models. Every task this course covered got its own purpose-built, hand-assembled architecture. That's not a flaw in how the course was taught — it's an honest reflection of how NLP actually worked before LLMs.

THREE REAL DIFFERENCES, NOT JUST "BIGGER"

- **Scale.** `nlp1-9`'s GloVe was trained on billions of words to produce a static lookup table of word vectors. An LLM is trained on comparably massive (often larger) corpora to learn an entire deep network end to end — not a fixed table of word meanings, but a full model of how language behaves in context.
- **Self-supervised pretraining, taken further.** Word2vec and GloVe already used a self-supervised signal — predict a word from its context, with no hand-labeled data required (`nlp1-5`, `nlp1-9`). LLMs scale that exact idea up to predicting the next token across a massive corpus, and that single pretraining objective turns out to be enough to learn grammar, facts, sentiment, and style all at once — capabilities this course had to teach as separate chapters.
- **One flexible architecture vs. many task-specific pipelines.** This capstone's own model cannot do NER; `nlp1-7`'s own model cannot do sentiment classification the way this capstone does. An LLM, once pretrained, can be steered toward sentiment classification, NER, translation, and more — often through prompting alone, sometimes through the same kind of fine-tuning `nlp1-9` introduced for embeddings, but applied to an entire pretrained network rather than just a lookup table.

Put plainly: this course taught how to hand-build the right tool for each job. An LLM is closer to one very large, very capable tool that has already seen enough language to be pointed at most jobs without being rebuilt from scratch each time. Neither approach makes the other obsolete to understand — knowing exactly what a purpose-built pipeline is doing, and why, is exactly what makes it possible to reason honestly about what a much larger, more opaque model might be doing instead.

Hands-On Exercises

EXERCISE 1

Using this capstone's own code, trace each line back to the specific chapter it came from, and explain why nlp1-2/nlp1-3's own bag-of-words and TF-IDF approaches are listed in the attribution table but not actually used in the final pipeline.

View solution

EXERCISE 2

Explain why nlp1-6's own sentiment classifier and nlp1-7's own NER tagger count as genuinely different pipelines despite sharing the same underlying LSTM mechanism, and explain what this reveals about how NLP worked before LLMs.

View solution

EXERCISE 3

Explain each of this chapter's own three named differences between this course's own techniques and LLMs (scale, self-supervised pretraining taken further, one flexible architecture vs. many task-specific pipelines), using a specific example from earlier in this course for each one.

View solution

Scope Note — What This Capstone Deliberately Doesn't Do

HONESTLY OUT OF SCOPE

- No self-attention or Transformer implementation — `nlp1-8` covered the motivation only; the full mechanism is deliberately deferred to `llm1`.
- No NER/POS pipeline — `nlp1-7`'s own bidirectional, per-token architecture is a genuinely separate model, not folded into this sentiment classifier.
- No fine-tuning of the GloVe embeddings — frozen only, per `nlp1-9`'s own guidance for a small dataset.

- No production deployment, serving, or monitoring — this capstone is a working pipeline, not a shipped product.

CHAPTER 10 QUICK REFERENCE — COURSE SUMMARY

- **The pipeline:** tokenize/clean (1) → pretrained frozen embeddings (9, resolving 5) → LSTM sequence model (6, resolving 4) → sigmoid classifier head (6)
- **What was diagnosed and fixed:** word order and word meaning as two genuinely independent problems (4), solved separately by sequence modeling (6) and embeddings (5/9)
- **What generalizes beyond this capstone:** the same LSTM idea, extended to per-token output for NER/POS (7); attention as NLP's own real motivation for what becomes the Transformer (8)
- **Why LLMs are different:** far greater scale, self-supervised pretraining pushed to learn many capabilities at once, and one flexible architecture in place of many hand-built pipelines
- This completes the NLP course (10 chapters) — the direct bridge into **llm1**'s own full transformer/LLM coverage