EAI 820 — Speech and Language Technologies

Week 1 Lecture Note

Turning Input into Tokens: Subwords & Discrete Audio Tokens

Jurafsky & Martin, SLP3 (draft, Jan 2026) — Chapter 2 (Words, Tokens & Subwords)

Duration: two hours (2 × ~50 min, with a short break)

Overview

Focus of the week. Turning raw text and transcribed speech into the discrete tokens on which every later module in the course depends.

Every technology studied in this course, from speech synthesis and speech recognition through machine translation and summarization, depends on a single preparatory step that is easy to overlook. Before any model can learn, the raw material of language, whether typed text or transcribed speech, must be converted into a sequence of discrete units that a computer can count, compare, and predict. Week 1 is devoted to that step. It introduces text normalization and tokenization, the processes that turn a stream of characters into the tokens on which every subsequent module rests.

The week opens with the vocabulary of corpora, drawing the distinction between word types and word tokens and showing how simple counts already expose structure in language. It then treats regular expressions as the practical instrument for locating and extracting patterns in text. From there the focus moves to normalization proper, which covers case folding, tokenization, lemmatization, and stemming, and to the deceptively hard question of what counts as a word in writing systems that do not separate words with spaces. The final part of the week presents subword tokenization and works carefully through the Byte-Pair Encoding algorithm, which learns its vocabulary from letter statistics rather than from a fixed dictionary and therefore copes gracefully with words it has never seen.

The decisions made this week are not merely technical housekeeping. The way text is segmented sets the size of a model's vocabulary, governs its handling of rare and unknown words, and shapes the quality of every downstream task. A tokenizer that is well matched to a language and a task is one of the quiet reasons a system works, and a poorly chosen one is a common reason a system fails.

Learning Objectives

By the end of Week 1 you will be able to:

1. Distinguish word types from word tokens and compute basic frequency statistics for a corpus.
2. Read and construct regular expressions to find and extract patterns in text.
3. Describe the stages of text normalization and explain when case folding, lemmatization, and stemming are each appropriate.
4. Explain why word level tokenization struggles with unknown words and how subword tokenization resolves the problem.

5. Trace the Byte-Pair Encoding algorithm by hand, learning an ordered set of merges from a small corpus and applying those merges to new text.
6. Segment text into sentences and account for the ambiguities that make the task harder than it first appears.
7. Explain how the same tokenization idea applies to audio as to text, through discrete audio tokens.

Reading and Preparation

The anchor text for this week is Jurafsky and Martin, *Speech and Language Processing*, third edition draft of January 2026, Chapter 2, titled Regular Expressions, Text Normalization, and Edit Distance. Concentrate on the sections covering words and corpora, text normalization, and tokenization, including the treatment of Byte-Pair Encoding. The minimum edit distance material that closes the same chapter is reserved for Week 2 and may be set aside for now.

In the Laboratory

The accompanying laboratory session puts these ideas into practice in Python. You will normalize and tokenize real text, compare the output of a word level tokenizer against a subword tokenizer, and implement the Byte-Pair Encoding merge loop so that the algorithm you trace by hand in the lecture becomes running code you can inspect and extend. The exercise closes the gap between the statistics on the slide and the behaviour of a working tokenizer.

Looking Ahead

Week 2 takes the tokens produced here and puts them to work. It builds n-gram language models, the first probabilistic account of how likely a sequence of words is, and it defines minimum edit distance, a measure of how far apart two strings are. Every module that follows, from automatic speech recognition to machine translation, consumes tokenized input, so the foundation laid this week is drawn upon continuously for the remainder of the course.

Part A — Words & Tokens (Chapter 2)

A1. Motivation: why start with text processing?

Every speech and language system — a translator, a summarizer, a speech recognizer — ultimately operates on sequences of symbols. Before any model can learn, we must decide what the atomic units of text are and how to measure similarity between strings. Chapter 2 answers three practical questions: what counts as a word, how do we split text into units a model can consume (tokenization), and how do we quantify how different two strings are (edit distance). These foundations reappear in every later week.

A2. What is a word? Types vs. instances

A key distinction runs through the whole course. **Word types** are the distinct words in a corpus — the vocabulary size $|V|$. **Word instances** (running words) are the total count N of tokens in the text.

Example — counting

```
They picnicked by the pool, then lay back on
the grass and looked at the stars.
```

Ignoring punctuation, this sentence has **14 types** and **16 instances** ("the" occurs three times, "by/the/then…" repeat).

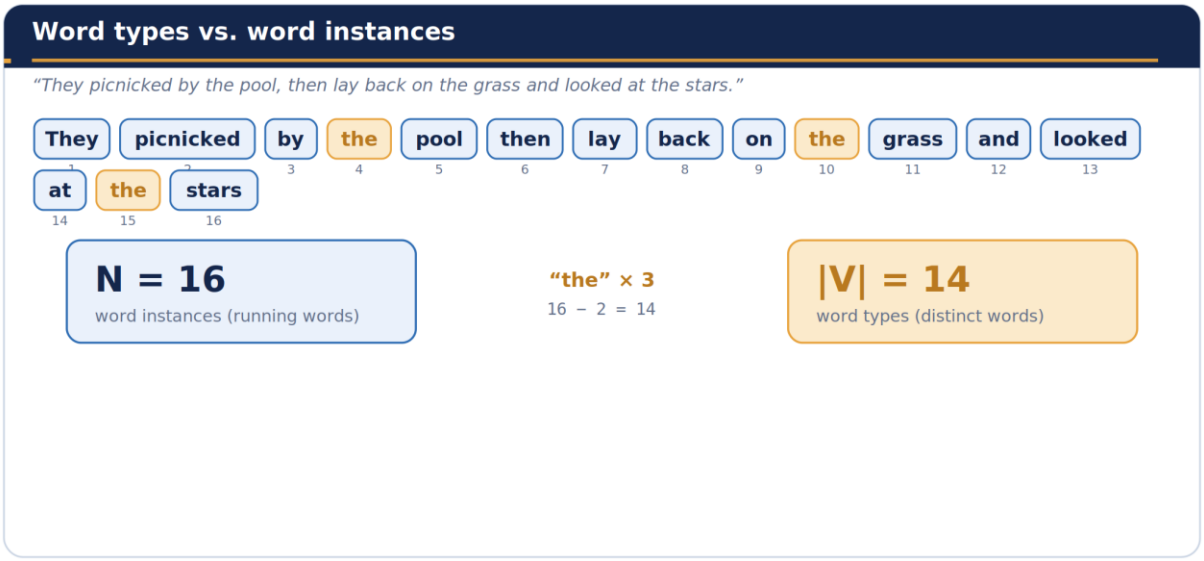


Figure 1. Word types vs. word instances — 16 running words, but only 14 distinct types.

The larger the corpus, the more types we find — vocabulary keeps growing with data. This relationship is **Herdan's Law** (a.k.a. Heaps' Law):

$$|V| = k \cdot N^{\beta}$$

with constants k and $0 < \beta < 1$ (typically $\beta \approx 0.44$–0.56). Roughly, vocabulary grows a little faster than the square root of text length. **Function words** (a, of, the) saturate quickly; **content words** (names, technical terms) keep appearing — which is exactly why fixed word vocabularies are fragile and why we turn to subword tokenization.

Rough corpus sizes (for scale)

Corpus	Types |V|	Instances N
Shakespeare	31 thousand	884 thousand
Brown corpus	38 thousand	1 million
Switchboard (phone calls)	20 thousand	2.4 million
COCA	2 million	440 million
Google n-grams	13 million	1 trillion

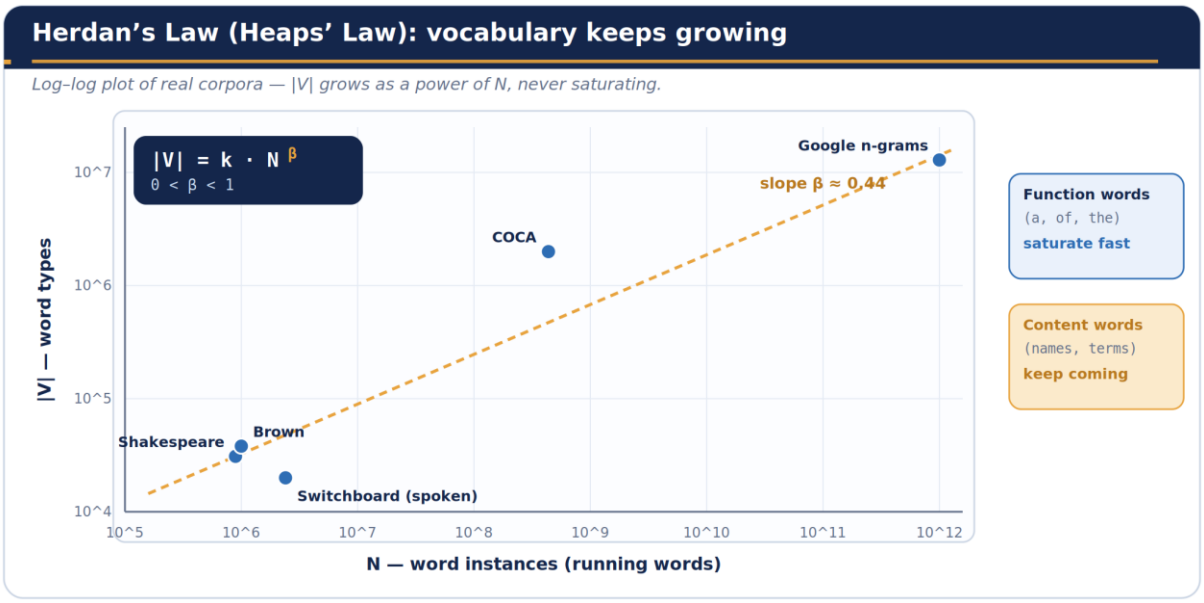


Figure 2. Herdan's (Heaps') Law on real corpora — vocabulary grows as a power of corpus size and never saturates.

Complications the definition must handle: capitalization (They vs. they — task-dependent), disfluencies (uh, um — kept for speech recognition), clitics (I'm = I + am), and languages such as Chinese, Japanese and Thai that **do not put spaces between words at all**. These issues motivate splitting below the word level.

A3. Morphemes: parts of words

> **Definition — morpheme**
>
> A **morpheme** is the minimal meaning-bearing unit in a language; the study of morphemes is **morphology**. "fox" is one morpheme; "cats" is two: *cat* + *-s* (plural).

Segmented example: *Doc work-ed care-ful-ly wash-ing the glass-es*.

We distinguish:

- **Roots** — the central morpheme carrying the main meaning (work, glass).
- **Affixes** — added meanings. **Inflectional** affixes play a grammatical role and are productive/predictable (-s plural, -ed past). **Derivational** affixes are idiosyncratic and often change word class (care → care-ful → care-ful-ly).
- **Clitics** — word-like but reduced and attached to a host (I've, the teacher's, French l'opera).

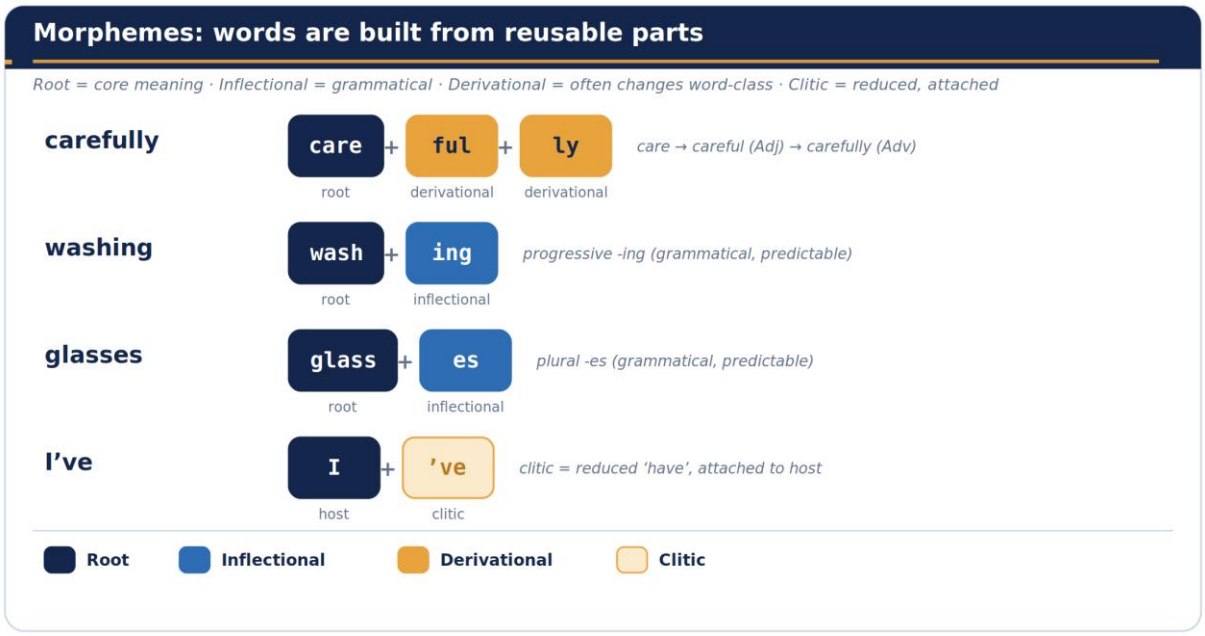


Figure 3. Morphological structure — roots plus inflectional/derivational affixes, and clitics.

Because content words are endlessly productive through morphology, a tokenizer that understands subparts of words generalizes far better than one that memorizes whole words — the bridge to BPE.

A4. Unicode, UTF-8 and normalization

Text is stored as characters encoded as bytes. **Unicode** assigns every character a numeric code point; **UTF-8** is the dominant variable-length byte encoding (ASCII characters stay one byte; others use two to four). Handling bytes correctly matters because modern tokenizers often operate at the byte level, which lets them represent *any* string — emoji, accented letters, scripts without spaces — with no "unknown character". Before tokenizing, systems typically apply **normalization**: consistent Unicode form, and task-dependent choices about case and punctuation.

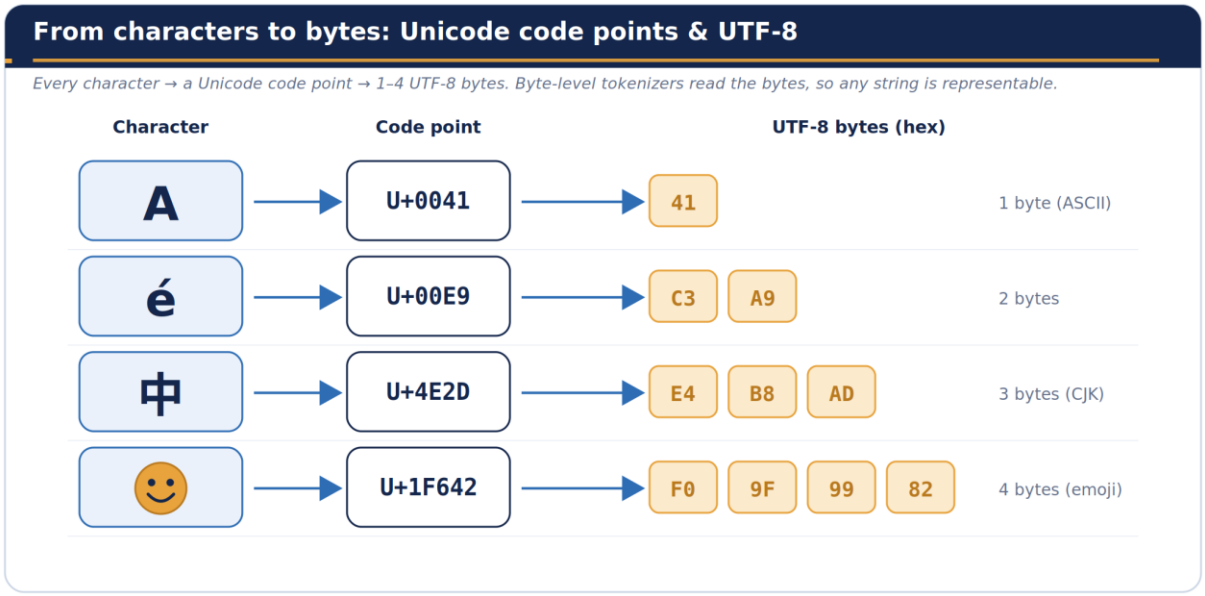


Figure 4. From characters to bytes: each character maps to a Unicode code point, then to 1–4 UTF-8 bytes.

A5. Subword tokenization: Byte-Pair Encoding (BPE)

Word tokenization breaks on unknown words (out-of-vocabulary), and character tokenization makes sequences painfully long. **Subword tokenization** is the compromise used by essentially all modern models: frequent words become single tokens, while rare words are split into meaningful pieces. BPE has two parts — a **token learner** (run once on a corpus to induce a vocabulary) and a **token segmenter** (applies the learned merges to new text).

BPE token learner (pseudocode)

```
function BYTE-PAIR-ENCODING(strings C, num merges k) -> vocab V
  V <- all unique characters in C          # start from characters
  for i = 1 to k:                           # do k merges
     (t_L, t_R) <- most frequent adjacent pair of tokens in C
     t_NEW <- t_L + t_R                     # new merged token
     V <- V + t_NEW
     replace every (t_L, t_R) in C with t_NEW
  return V
```

Merges are learned **within words** (text is first split on whitespace/punctuation), so a merge never crosses a word boundary.

Worked trace — corpus "set new new renew reset renew"

Start (word : count), vocab = { _, e, n, r, s, t, w }:

```
n e w        2      r e n e w    2
s e t        1      r e s e t    1
```

Most frequent adjacent pair is **n e** (in *new* ×2 and *renew* ×2 = 4). Merge → add **ne**.

```
ne w   2     r e ne w   2     s e t   1     r e s e t   1
```

Next **ne w** (=4) → **new**; then **r e** → **re**. The learner has discovered the prefix *re-* and the whole word *new*. Continuing yields tokens like *renew, se, set*.

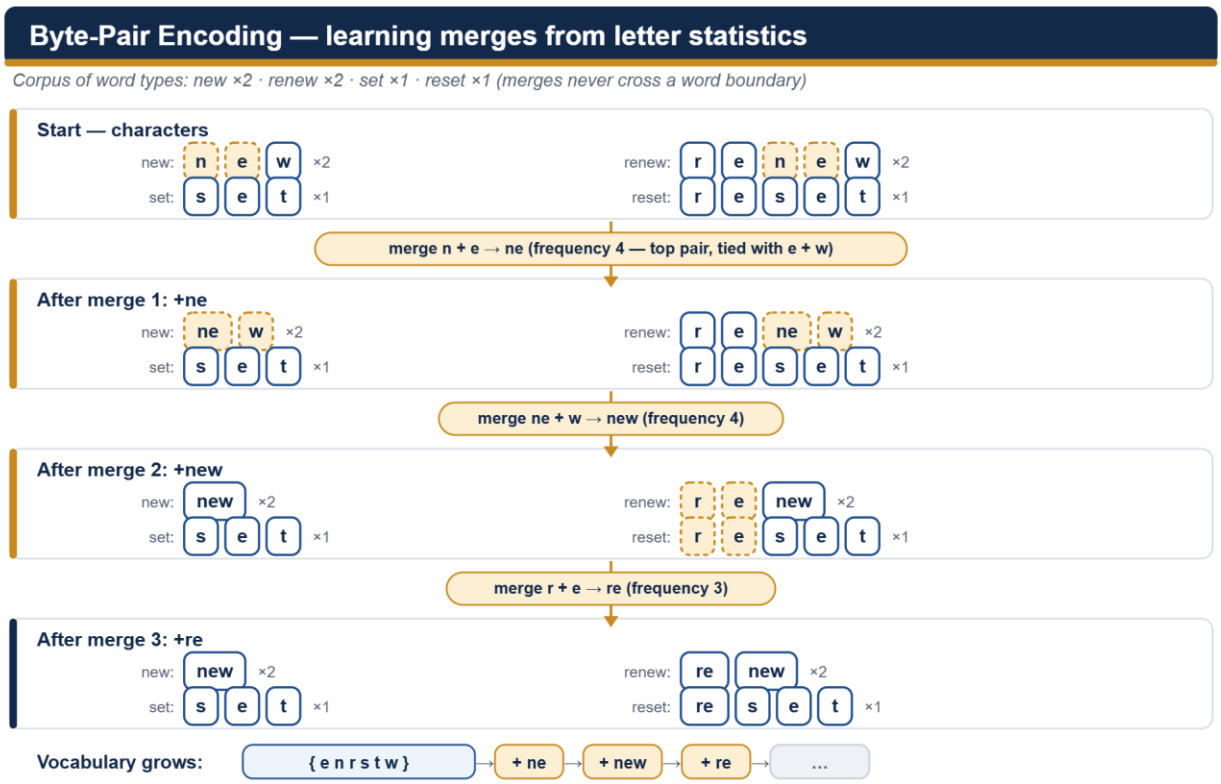


Figure 5. BPE token learner — iterative merging of the most frequent adjacent pair grows the vocabulary.

Takeaway: BPE uses nothing but letter-sequence statistics to build a vocabulary that balances coverage against sequence length — and it never produces a truly unknown token when run over bytes.

A6. Corpora, regular expressions, and simple tokenizers

A **corpus** is a curated body of text (or speech). Responsible practice asks who produced it, when, why, and with what biases — a "datasheet" for the data. **Regular expressions** are the workhorse for specifying string patterns (character classes, ranges, disjunction, capture groups, substitutions) and are the first tool used to split text and to write rule-based tokenizers. A minimal whitespace/punctuation tokenizer can even be built from **Unix tools** alone:

```
tr -sc 'A-Za-z' '\n' < file.txt | sort | uniq -c | sort -nr
```

This pipeline splits on non-letters, then counts and ranks word frequencies — a surprisingly useful baseline and a good mental model for what a tokenizer does.

Part B — Audio as Tokens: the Neural Codec

Everything so far has concerned text. Audio poses the same problem in a different key: a microphone delivers a continuous stream of amplitude samples — typically sixteen thousand every second — and no sequence model can consume that raw. The parallel to byte-pair encoding is the neural audio codec, which learns a vocabulary of discrete audio tokens by training an encoder to compress a waveform into a short sequence of symbols and a decoder to reconstruct it.

The intellectual move is identical to BPE: turn an open-ended signal into a finite alphabet. Once it has been made, the same modelling machinery applies to sound as to text — the single observation that merged two

fields kept apart for half a century, and the reason this course can treat speech synthesis and machine translation as relatives rather than strangers. Weeks 6 and 7 build directly on it.

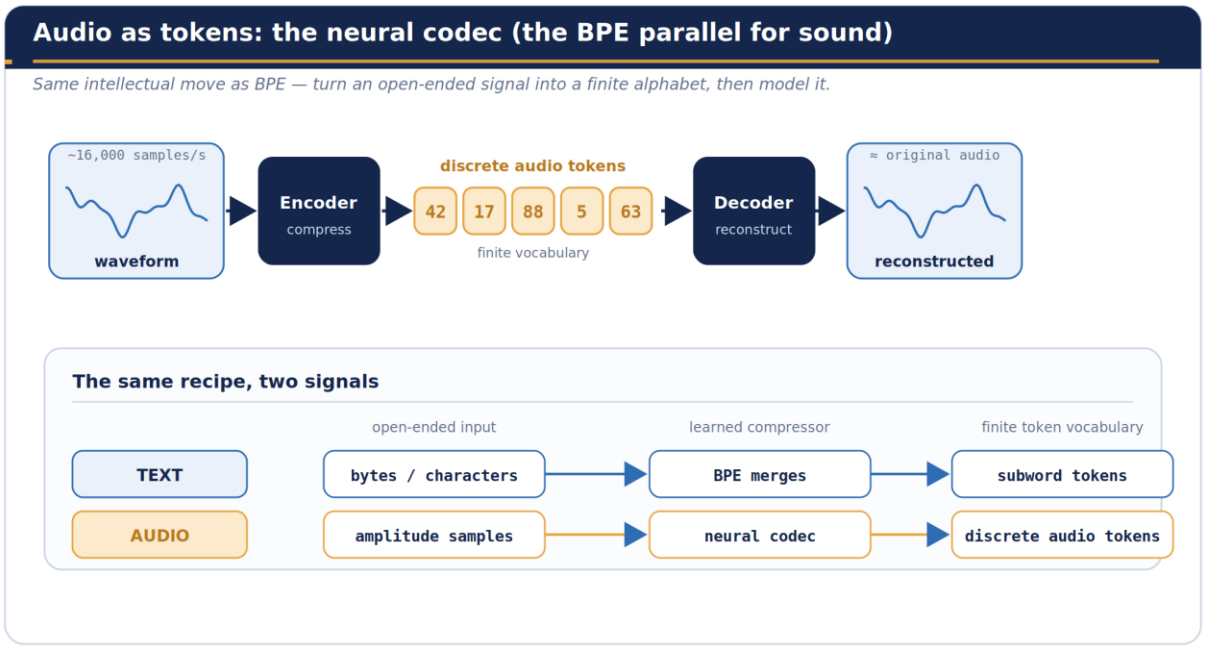


Figure 6. The neural audio codec mirrors BPE: an open-ended signal becomes a finite token vocabulary.

Wrap-up

Key takeaways

- Words are slippery; types ≠ instances, and vocabularies grow without bound (Herdan/Heaps).
- Morphology explains why words have reusable parts — motivating subword tokenization.
- BPE induces a subword vocabulary from pure letter-sequence statistics via iterative merges.
- Audio becomes tokens the same way text does: a neural codec learns a finite vocabulary of discrete audio tokens, placing speech and text under one kind of model.

Connection to this week's labs

Concept in lecture	Where you implement it
Byte-Pair Encoding (2.4)	Implement BPE merges from scratch in plain Python
Tokenization / word counts (2.1, 2.7)	The Unix pipeline above; extend it into a simple tokenizer

Exercises & discussion

- **1. Count the word types and instances in a paragraph of your choice, with and without lowercasing. How much does the count change?**
- **2. Hand-run two more BPE merges on "set new new renew reset renew" beyond the trace above. What token appears next?**
- **3. In one or two sentences, explain why a neural audio codec is the audio counterpart of byte-pair encoding: what does each turn into a finite vocabulary, and why does that matter downstream?**

Reading & references

- Primary: Jurafsky & Martin, Speech and Language Processing (3rd ed. draft, Jan 2026), Chapter 2 (Words, Tokens and Subwords).
- Supplementary: Sennrich et al. (2016) on BPE for NMT; the course introduction (§3, Idea 1) on discrete audio tokens.
- Next week (Week 2): counting language models, smoothing and perplexity, and minimum edit distance — Chapters 3 and 2.9.

Note. *Section numbers and worked examples follow the January 2026 SLP3 draft; as a living draft, minor numbering may shift in later versions.*