Words and Tokens

Rob Minneker

Administrivia

- A0 is out, no submission, just for practice
- Project specs are live on course website!
  - Teams of 3
  - Various checkpoints throughout the quarter
  - TODO: Read specs, form teams, ask any questions on Ed

PART 1: FOUNDATIONS

Why tokens matter (1/6)

Tokenization: the process of segmenting text into minimal units, or tokens, is foundational to all NLP tasks.

Why tokens matter (2/6)

- Early NLP systems, such as ELIZA, relied on pattern matching over tokens (often words) to create the illusion of conversation.
  - Example: ELIZA used simple patterns like `I need X` and change the words into suitable outputs like `What would it mean to you if you got X?`
  - Token boundaries define what patterns can be matched, impacting system behavior.

ELIZA Example

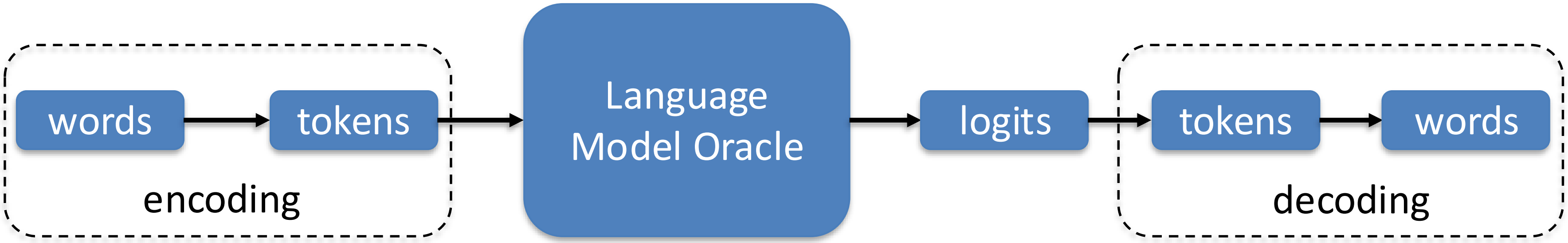
- Rogerian psychotherapist imitation via pattern matching

```
User:   I need some help, that much seems certain.
ELIZA:  WHAT WOULD IT MEAN TO YOU IF YOU GOT SOME HELP
User:   Perhaps I could learn to get along with my mother.
ELIZA:  TELL ME MORE ABOUT YOUR FAMILY
User:   My mother takes care of me.
ELIZA:  WHO ELSE IN YOU FAMILY TAKES CARE OF YOU
User:   My father.
ELIZA:  YOUR FATHER
User:   You are like my father in some ways.
```

Weizenbaum (1966)

Why tokens matter (3/6)

- Tokenization is the first step in most NLP pipelines



Why tokens matter (4/6)

- The definition of a “token” is task-dependent.
  - For language models, punctuation marks (e.g., . , , , !) are typically treated as tokens.
  - For some tasks (e.g., sentiment analysis), splitting contractions or handling hyphenation may be important.

Why tokens matter (5/6)

- **Applications:**
  - Pattern matching: Regular expressions such as `\bword\b` operate over token boundaries.
  - Statistical analysis: Frequency counts and laws (Zipf’s, Heaps’) depend on token definitions.
  - Sequence models: Input and output spaces are defined over tokens, affecting vocabulary size.

Why tokens matter (6/6)

Formally, tokenization defines a function:

$$\text{Tokenize}(T) = [t_1, t_2, \ldots, t_n]$$

where T is the input text and t_i are the resulting tokens.

- The choice of tokenization granularity (word, subword, character) can dramatically influence model capacity, generalization, and robustness.

What is a “word”? (1/6)

A “word” in NLP is a fundamental linguistic unit, but its definition is context- and task-dependent.

What is a “word”? (2/6)

- In written text, a “word” is often defined as a sequence of alphabetic characters separated by whitespace or punctuation.
  - Regular expression example: `\w+` matches contiguous word characters.
  - Formal definition: a “word” is any substring w such that w is maximal and $w \in \Sigma^{+}$, where Σ is the alphabet, and w is bounded by whitespace or punctuation.

What is a “word”? (3/6)

- Counting words can differ based on punctuation handling:
  - “Let’s go to the picnic.” (punctuation excluded: 5 words; included: 6 tokens)
  - Tokenization schemes must specify whether punctuation is a separate token.

What is a “word”? (4/6)

- In spoken language, word boundaries and wordhood are less clear:
  - Disfluencies (e.g., “uh”, “um”), fragments, and filled pauses complicate word segmentation.
  - Example: “I do uh main- mainly business data processing”
  - “uh”: filled pause
  - “main-”: fragment (incomplete word)
  - “mainly”: completed word

What is a “word”? (5/6)

Applications:

- For automatic speech recognition (ASR), retaining filled pauses like “uh” and “um” can be informative:
  - These tokens may signal hesitation, uncertainty, or pragmatic meaning.
  - “uh” vs “um” may have distinct discourse functions (e.g., shorter vs longer pause).

What is a “word”? (6/6)

- The definition of “word” affects downstream tasks:
  - Morphological analysis, syntactic parsing, and language modeling depend on consistent tokenization.
  - Different applications (e.g., IR vs ASR) may require distinct word definitions.
- The choice of word definition is thus a design decision, shaped by data modality, annotation conventions, and task requirements.

Types vs instances (1/4)

A **type** is a unique wordform in a text, while an **instance** (token) is a specific occurrence of a word in the running text.

- Types correspond to the vocabulary set size ($|V|$); instances to the total word count (N).
  - Example: In the sentence "The picnic was a great picnic", `picnic` counts as one type, two instances.

Types vs instances (2/4)

- Formal definitions:
  - Type: $w \in V$, where $V = \{\text{unique wordforms in corpus}\}$
  - Instance: Each position i in the text, w_i, where $1 \leq i \leq N$

Types vs instances (3/4)

- Worked example: For “The picnic was a great picnic”
  - Tokens (instances): [The, picnic, was, a, great, picnic] ($N = 6$)
  - Types: [The, picnic, was, a, great] ($|V| = 5$)

Types vs instances (4/4)

- Case sensitivity decision:
  - “They” vs “they”: Should these count as the same type?
  - Feature: Maintain case to capture proper nouns or sentence-initial position
  - Normalization: Lowercase all to merge types, reducing vocabulary size

Applications:

- Type-token statistics are central to language modeling and corpus linguistics
- Heaps’ law relates vocabulary growth (types) to corpus size (instances): $|V| = kN^{\beta}$

Multilinguality: when “words” aren’t separated by spaces (1/2)

In many languages (e.g., Chinese, Japanese, Thai), text does not use whitespace to separate words, complicating tokenization and downstream NLP tasks.

- Example: Chinese string “姚明进入总决赛 ” (Yao Ming reaches the finals) admits multiple valid segmentations.
- Possible: “姚明 / 进入 / 总决赛 ” vs. “姚 / 明进 / 入决 / 赛”

Multilinguality: when “words” aren’t separated by spaces (2/2)

- The concept of “word” is language-dependent and often ambiguous; segmentation choices can alter meaning and system performance.
- Ambiguity in segmentation:
  - For a character sequence $c_1 c_2 \ldots c_n$, the number of segmentations equals the number of binary partitions:

$$\text{Number of segmentations} = 2^{n-1}$$

→ hard to do, language specific, sensitive to code switching

Vocabulary growth and the “too many words” problem (1/4)

Vocabulary Growth and the “Too Many Words” Problem

- The number of unique word types ($|V|$) in a corpus grows as more tokens (N) are observed.
  - Heaps’ (Herdan’s) Law formalizes this growth:

$$|V| = kN^{\beta}$$

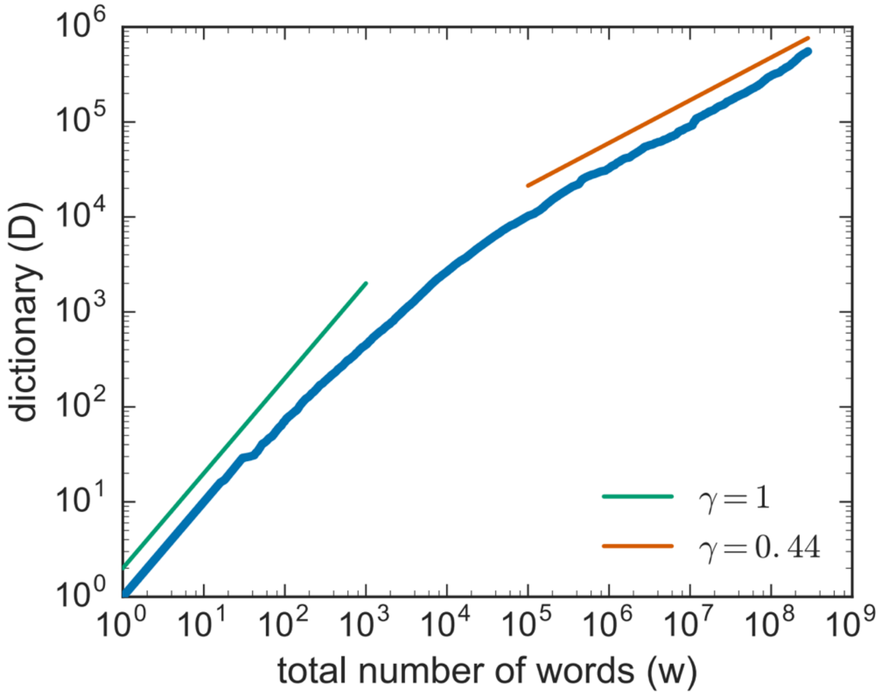
where $k > 0$ and $0 < \beta < 1$.

Intuitively: vocabulary expands sublinearly with corpus size but never saturates.

Vocabulary growth and the “too many words” problem (2/4)

Vocabulary size as a function of text length, computed on the Gutenberg corpus of publicly available books

~ vocabulary size grows little faster than the square root of its length in words



Vocabulary growth and the "too many words" problem (3/4)

- Function words vs. content words:
  - Function words (e.g., `the`, `and`) are frequent and saturate quickly.
  - Content words (e.g., names, technical terms) grow continually, driving $|V|$ upward.
  - Proper nouns and specialized vocabulary lead to an open-ended lexicon.

Vocabulary growth and the “too many words” problem (4/4)

- The “too many words” problem:
  - Large, ever-growing vocabularies make language modeling and NLP tasks challenging.
  - Rare and unseen words (out-of-vocabulary, OOV) are pervasive, especially in open domains.

PART 2: CHARACTERS AND REPRESENTATION (UNICODE + UTF-8)

Unicode basics (1/5)

Unicode Basics

- Unicode is a universal character encoding standard designed to represent text from all writing systems and symbols in a consistent way.
  - Motivation: ASCII encodes only 128 characters, insufficient for global text processing.
  - Unicode enables NLP systems to process multilingual, cross-script, and symbolic data.

Unicode basics (2/5)

- Key distinction:
  - **Code point:** An abstract numerical identifier for a character, written as $U + \mathrm{XXXX}$.
  - Example: $U + 0061$ is the code point for the Latin letter ‘a’.
  - **Glyph:** The visual rendering of a code point, determined by font and style.
  - The same code point may map to different glyphs in different fonts or contexts.

Unicode basics (3/5)

- Each code point is independent of encoding and visual appearance.
  - For example, the code point $U + 00E9$ represents 'é', regardless of how it is displayed. (we'll see this in a moment)

Unicode basics (4/5)

- Code points for selected characters:
  - Latin small letter ‘a’: $U + 0061$
  - Latin small letter ‘é’: $U + 00E9$
  - Chinese character ‘你’: $U + 4F60$
  - Emoji ‘😊’: $U + 1F60A$

Unicode basics (5/5)

- Unicode enables algorithms to manipulate text as sequences of code points, not bytes or glyphs.
- Unicode’s abstraction is foundational for robust, language-independent NLP systems.
  - Currently ~150k characters defined, out of ~1.1M

UTF-8 encoding (1/3)

- Unicode Transformation Format – 8 bit (UTF-8) is a variable-length encoding for Unicode code points, designed to be:
  - Space-efficient for ASCII (one byte per character)
  - Backward-compatible with legacy ASCII systems
  - Capable of representing all Unicode code points (0 to $10FFFF_{16}$)

UTF-8 encoding (2/3)

- Intuition:
  - UTF-32 directly encodes code points as 4 bytes: simple but wasteful for common text
  - UTF-8 uses 1–4 bytes per code point, depending on its value:

Bytes	Bit Pattern	Code Point Range
1	0xxxxxxx	U+0000 to U+007F
2	110xxxxx 10xxxxxx	U+0080 to U+07FF
3	1110xxxx 10xxxxxx 10xxxxxx	U+0800 to U+FFFF
4	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx	U+10000 to U+10FFFF

UTF-8 encoding (3/3)

- Efficient text storage/transmission in NLP pipelines
- Consistent handling of multilingual corpora
- Enables byte-wise compatibility with legacy systems
- Note: $\text{len}(s)$ gives code points, same as “chars”
  - `len(s.encode("utf-8"))` gives bytes used

Tokenization implications (1/6)

- In multilingual NLP, text is often represented in Unicode, encoded in UTF-8 for storage and processing.
- UTF-8 encodes Unicode characters as sequences of 1–4 bytes, but not all byte sequences are valid characters.

Tokenization implications (2/6)

- Byte-Pair Encoding (BPE) and similar algorithms may operate on UTF-8 bytes rather than characters.
  - Merges are performed on byte sequences, not necessarily respecting character boundaries.
  - This ensures all possible byte values ($0–255$) are covered: no “unknown byte” issue.

Tokenization implications (3/6)

- **Potential Issues:**
  - If a merge spans across UTF-8 character boundaries, it may create invalid or uninterpretable byte sequences.
  - For example, merging bytes from different characters may break Unicode validity.
  - Such merges can yield tokens that do not correspond to any real character or grapheme.

Tokenization implications (4/6)

- **Example:**
  - Suppose we have the UTF-8 encoding for ‘é’: `[0xC3, 0xA9]`
  - If BPE merges `0xA9` from ‘é’ with a following ASCII byte, the result is not a valid character.
  - e.g., the merge operation:

merge: `[0xC3, 0xA9]` + `[0x20]` -> `[0xC3, 0xA9, 0x20]`

may produce an invalid token if not aligned to character boundaries.

Tokenization implications (5/6)

- What happens if we build a model expecting "é" (`[0xC3, 0xA9]`) as an output and we get "é " out? `[0xC3, 0xA9, 0x20]`
- What about if we are looking for output begins with "Yes" or "No"?

Tokenization implications: curveball

- What happens if we build a model expecting “é” (`[0xC3, 0xA9]`) as an output and we get “é ” out? `[0xC3, 0xA9, 0x20]`
- What about if we are looking for output begins with “Yes” or “No”?
  - What if the model starts outputting “Notor”?

Tokenization implications: curveball solution

- What happens if we build a model expecting “é” (`[0xC3, 0xA9]`) as an output and we get “é ” out? `[0xC3, 0xA9, 0x20]`
- What about if we are looking for output begins with “Yes” or “No”?
  - What if the model starts outputting “Notor”?
    - Option 1: Don’t use fragile string matching, use logits directly
    - Option 2: If you can’t use logits, go for token ids and rigorously test/validate the stack
    - Option 3: Use structured outputs if available (can be combined with others)

Tokenization implications (6/6)

- **Applications:**
  - Robust tokenization strategies must account for Unicode encoding to avoid generating ill-formed tokens.
  - Many modern models (e.g., GPT-2/3) use byte-level BPE to simplify the vocabulary and avoid out-of-vocabulary (OOV) issues.
  - Ensuring merges only occur within valid UTF-8 boundaries is essential for text integrity and reversibility.

PART 3: TOKENIZATION STRATEGIES OVERVIEW

Three candidates for units (1/5)

Tokenization: the process of segmenting text into units ("tokens") for downstream NLP tasks.

Three candidates for units (2/5)

- Three primary candidates for tokenization units:
  - **Words:**
    - Pros: Intuitive and meaning-rich; often correspond to linguistic units.
    - Cons: Ambiguous boundaries. Hard to define consistently across languages.
  - **Morphemes:**
    - Linguistically motivated sub-word units (smallest meaning-bearing elements).
    - Pros: Capture internal structure of words, useful for morphology-rich languages.
    - Cons: Requires complex, language-specific analyzers. Not always uniquely defined.
  - **Characters:**
    - Atomic, well-defined units.
    - Pros: Language-independent; no segmentation ambiguity.
    - Disadvantages: Too small for most semantic tasks; long sequences, reduced efficiency.

Three candidates for units (3/5)

Applications:

- Word-based models (e.g., classical bag-of-words, word2vec) often struggle with unseen or rare words.
- Character-level models are robust to out-of-vocabulary items but less semantically informative.

Three candidates for units (4/5)

- **Practical compromise: subwords**
  - Subword units (e.g., via Byte-Pair Encoding, Unigram LM) combine the advantages of words and characters.
  - Recombine to represent unseen words, reducing out-of-vocabulary rates.
  - Allow efficient vocabulary size control:

Subword vocabulary: V = {`un`, `##seen`, `word`}

Three candidates for units (5/5)

→ Can generate `unseenword` as `un` + `##seen` + `word` - Widely adopted in modern NLP architectures (e.g., BERT, GPT).

- **Key insight:** The choice of tokenization unit directly impacts model vocabulary, data sparsity, and cross-lingual applicability.

PART 4: SUBWORD TOKENIZATION — BYTE-PAIR ENCODING (BPE)

Big picture: trainer + encoder (1/5)

Byte-Pair Encoding (BPE) employs a two-stage architecture: a trainer builds the merge rules (vocabulary), and an encoder applies these merges to segment new text.

Big picture: trainer + encoder (2/5)

- **BPE Trainer:** Learns a sequence of symbol merges from a training corpus.
  - Begins with a vocabulary of single characters.
  - Iteratively finds the most frequent adjacent symbol pair and merges it.
  - The process is repeated for N steps to build a vocabulary of frequent subwords.

Big picture: trainer + encoder (3/5)

- **BPE Encoder:** Segments input text by greedily applying the learned merges.
  - Given a word, repeatedly merges symbol pairs according to the learned order.
  - Produces a sequence of subword units present in the final BPE vocabulary.

Big picture: trainer + encoder (4/5)

- **Formalization:**

  At each step: $(x^*, y^*) = \underset{(x,y)}{\operatorname{argmax}} \operatorname{freq}(x, y)$

  Merge $(x^*, y^*) \rightarrow$ new symbol $z = x^* y^*$

- **Key Insights:**
  - The trainer determines which subword units are represented, balancing vocabulary size with corpus coverage.
  - The encoder is deterministic: the same input string always yields the same segmentation under fixed merges.

Big picture: trainer + encoder (5/5)

Applications:

- Used in modern NLP models (e.g., GPT) to handle rare words, reduce vocabulary size, and improve generalization.

BPE training: core algorithm intuition (1/5)

Byte-Pair Encoding (BPE) Training: Core Algorithm Intuition

- BPE is an unsupervised, greedy algorithm that constructs a subword vocabulary by iteratively merging the most frequent pair of adjacent symbols in a corpus.
- Initial vocabulary consists of all characters in the corpus.
  - Each word is represented as a sequence of characters (with special end-of-word marker).

BPE training: core algorithm intuition (2/5)

- At each iteration:
  - Identify the most frequent adjacent symbol pair (e.g., `n e` in `new`).
  - Merge this pair into a new symbol (e.g., `ne`), updating both the corpus and the vocabulary.

BPE training: core algorithm intuition (3/5)

- Example: Corpus = `set new new renew reset renew`
  - Step 0: Words are tokenized as sequences of characters:
  - `s e t`, `n e w`, `n e w`, `r e n e w`, `r e s e t`, `r e n e w`
  - Step 1: Count adjacent pairs; most frequent is `n e`.
    - Merge: `n e` → `ne`
    - Update: `ne w`, `r e ne w`, etc.
  - Step 2: Next frequent pair is `ne w`.
    - Merge: `ne w` → `new`
    - Update: `new`, `r e new`, etc.
  - Further merges may create higher-level subwords (e.g., `re-` prefix).

BPE training: core algorithm intuition (4/5)

- BPE discovers recurring subword patterns (e.g., `re-`) without linguistic supervision.
- Pseudocode (from J&M Fig. 2.6):

function BYTE-PAIR ENCODING(strings C, number of merges k) **returns** vocab V

$V \leftarrow$ all unique characters in C # initial set of tokens is characters
for $i = 1$ **to** k **do** # merge tokens k times
  $t_L, t_R \leftarrow$ Most frequent pair of adjacent tokens in C
  $t_{NEW} \leftarrow t_L + t_R$ # make new token by concatenating
  $V \leftarrow V + t_{NEW}$ # update the vocabulary
  Replace each occurrence of t_L, t_R in C with t_{NEW} # and update the corpus
return V

BPE training: core algorithm intuition (5/5)

Applications:

- Produces subword units that capture morphological structure and rare word forms.
- Widely used in neural machine translation and large language models.

BPE “in practice” (1/6)

- BPE is applied over token sequences, often at the byte level for maximal coverage.
  - Byte-level BPE operates on UTF-8 byte sequences, ensuring compatibility with any text.
  - Illegal or rare byte sequences are typically filtered to prevent encoding errors.
  - This approach yields robust, language-agnostic tokenization, but may ignore linguistic boundaries.

BPE "in practice" (2/6)

- **Pretokenization with Regex:**
  - Pretokenization splits raw text into initial fragments using regular expressions.
  - Common regex patterns: whitespace (`\s+`), punctuation (`[.,!?]`), and digit chunking (`\d+`).
  - Clitic handling: patterns like `(\w+)'(s|re|ve)` to preserve contractions as units.
  - Formally, given input x, pretokenizer P maps $x \rightarrow (w_1, w_2, \ldots, w_n)$, where each w_i is a fragment.

BPE “in practice” (3/6)

- Multilingual tokenization introduces vocabulary imbalances.
  – BPE trained on English-heavy corpora oversegments other languages:
  – Non-English words split into longer subword sequences.
  – Leads to longer input sequences and higher computational cost for underrepresented languages.

BPE "in practice" (4/6)

- **Unigram LM Tokenization (Alternative):**
  - Unigram LM defines a probabilistic model over possible subword segmentations:

$$\text{Segment}(x) = \underset{S \in \mathcal{S}(x)}{\arg\max} P(S)$$

BPE “in practice” (5/6)

- S: segmentation, $P(S)$: product of subword probabilities.
  - Contrast: BPE is deterministic and greedy; Unigram LM is probabilistic and can sample multiple segmentations.
  - Linguistic pros/cons:
    - Unigram LM can encode alternative morphological analyses.
    - BPE is preferred for efficiency, lossless compression, and deterministic decoding.

BPE “in practice” (6/6)

Applications:

- Engineering: BPE is favored for large-scale, multilingual systems due to speed and simplicity.
- Research: Unigram LM tokenization supports richer linguistic modeling, especially for morphologically complex languages.

PART 5: RULE-BASED TOKENIZATION (WHEN YOU WANT “WORDS”)

Why rule-based still matters (1/4)

Rule-based tokenization refers to the use of handcrafted patterns or algorithms to segment text into word-like units, as opposed to relying solely on statistical or neural models.

- Many NLP tasks require precise, linguistically motivated tokens:
  - Syntactic parsing, morphological analysis, and social science studies often demand word-like units that respect linguistic conventions.

Why rule-based still matters (2/4)

- English tokenization desiderata highlight why rules are still essential:
  - Separate most punctuation marks from words (e.g., “hello!” → “hello”, “!”)
  - Preserve internal punctuation in abbreviations or acronyms (e.g., `U.S.A.`, `e.g.`, `co-op`)
  - Retain entities with internal structure as single tokens:
  - Monetary amounts: `$45.55`
  - Dates: `12/31/2023`
  - URLs: `https://www.example.com`
  - Hashtags and emails: `#NLP`, `user@example.com`

Why rule-based still matters (3/4)

- Rule-based approaches handle language-specific conventions more robustly:
  - Numeric formats differ (e.g., English `1,000.5` vs. German `1.000,5`)
  - Rule-based tokenizers can use regular expressions such as

`\$[0-9]+(\.[0-9]{2})?`

for monetary amounts, ensuring correct treatment.

Why rule-based still matters (4/4)

- **Applications:**
  - Preprocessing for downstream tasks that assume word boundaries (e.g., part-of-speech tagging)
  - Social media and domain-specific texts, where statistical models may lack coverage
  - Linguistic research requiring faithful segmentation of complex forms
  - Statistical and neural tokenizers may over-split or under-split without explicit rules, making rule-based methods indispensable for accuracy in many scenarios.

Tokenization standards (1/4)

- Tokenization speed is critical:
  - Tokenization is a prerequisite for downstream tasks (e.g., parsing, tagging).
  - Inefficient tokenization can bottleneck large-scale NLP workflows.
  - Real-world systems often require processing millions of words per second.

Tokenization standards (2/4)

- Standard approaches leverage regular expressions and finite-state automata:
  - Regex-based tokenizers (e.g., NLTK’s regexp_tokenize) use patterns such as +|[^\w\s]+ to separate words from punctuation.
  - Formally, let be the input alphabet; a finite-state automaton (FSA) defines a set of states and transitions to efficiently recognize token boundaries.
  - Example FSA for whitespace tokenization:

Tokenization standards (3/4)

- Intuition: Tokenization standards must balance linguistic accuracy with computational efficiency.
  - Simple whitespace or punctuation-based splitting misses edge cases (e.g., “can’t”).
  - Overly complex patterns may slow processing and reduce maintainability.

Applications:

- Preprocessing for language modeling, parsing, and information retrieval.

Tokenization standards (4/4)

- Standardized tokenization ensures comparability across corpora and experiments.

Regex tokenizer example (NLTK-style) (1/4)

- Rule-based tokenization with regular expressions enables fine-grained control over how input text is segmented into tokens, crucial for downstream NLP tasks.
- Key idea: A regex tokenizer applies pattern-matching rules to extract tokens, rather than relying solely on whitespace or built-in language rules.
- Example input: That U.S.A. poster-print costs $12.40…
- Typical regex pattern components (NLTK-style):

Regex tokenizer example (NLTK-style) (2/4)

Component	Example Pattern	Description
Abbreviations	`[A-Z]\.(?:[A-Z]\.)+`	Match acronyms (e.g., U.S.A.)
Hyphenated words	`\w+(?:-\w+)+`	Match hyphenated words (e.g., poster-print)
Currency and numbers	`\\d+(?:\.\d+)?`	Match currency and numbers (e.g., \12.40)
Ellipsis	`\.{3}`	Match ellipsis (...)
Punctuation	`[.,!?;:]`	Match punctuation marks

Regex tokenizer example (NLTK-style) (3/4)

- Output token list for the example:

  [That, U.S.A., poster–print, costs, $12.40, …]

Applications:

- Adapting regex patterns allows for custom tokenization, such as:
  - Preserving emails as single tokens: add `[\w\.-]+@[\w\.-]+` to the pattern
  - Handling URLs, hashtags, or domain-specific entities

Regex tokenizer example (NLTK-style) (4/4)

- Limitations:
  - Rule-based tokenizers may miss linguistic subtleties (e.g., contractions, ambiguous cases)
  - Maintenance and extensibility can be challenging for highly variable data

PART 6: CORPORA, VARIATION, AND TOKENIZATION CHOICES

Why corpora context matters (1/5)

Corpora context refers to the social, linguistic, and situational factors surrounding the creation of text. These dimensions crucially impact linguistic analysis, annotation, and tokenization decisions.

Why corpora context matters (2/5)

- Text is always situated: speaker, dialect, time, and communicative purpose influence content
  - Example: Transcripts from therapy sessions (e.g., ELIZA dialogues) differ from newswire
  - Speaker intent and formality shape lexical choice, syntax, and segmentation

Why corpora context matters (3/5)

- Variation dimensions affect language data:
  - Language: Over 7000 languages, each with unique orthography and morphology
  - Dialects/Varieties: Features in African American English (AAE) may alter word boundaries or spelling (e.g., “gon’” for “going to”)
  - Genre: Tokenization differs for news, fiction, medical notes, or conversational transcripts
  - Medical notes: “bp120/80” (blood pressure) vs. standard prose

Why corpora context matters (4/5)

- Code-switching and mixed-language phenomena complicate tokenization
  - Example: Social media post mixing English and Spanish ("Estoy happy today!")
  - Algorithms must handle multilingual word boundaries and hybrid grammar

Applications:

- Tokenizer design must be corpus-aware; generic rules may fail on nonstandard or domain-specific text

Why corpora context matters (5/5)

- Annotation guidelines often require adaptation to the sociolinguistic context of the corpus
- Zipf’s Law and Heaps’ Law depend on corpus context:
  - i.e. empirical statistics of the corpus

RECAP: PUTTING IT ALL TOGETHER

Tokenization design decision framework (1/4)

Tokenization is the process of segmenting text into units (tokens) for downstream NLP tasks. Selecting an appropriate tokenization strategy is critical, as it affects model performance, fairness, and linguistic adequacy.

Tokenization design decision framework (2/4)

The optimal tokenization scheme depends on the downstream task and linguistic context.

- For parsing or syntax-sensitive applications:
  - Rule-based or word-level tokenization is preferred, possibly augmented with clitic handling.
  - Example: English contractions (“don’t” → “do” + “n’t”) require splitting to preserve syntactic structure.
- For language models and open-vocabulary generation:
  - Subword tokenization (e.g., Byte-Pair Encoding, Unigram LM) balances vocabulary size and coverage.
  - Byte-level tokenization ensures safety for rare or unseen scripts.
  - Example: BPE learns frequent subword units such that common words are single tokens, rare words split into smaller units.
- For multilingual fairness:
  - Measure and minimize oversegmentation, especially in morphologically rich or low-resource languages.
  - Evaluate per-language token efficiency (e.g., average tokens per word, coverage rate).

Tokenization design decision framework (3/4)

Applications:

- When designing a tokenizer, analyze the trade-off between vocabulary size, out-of-vocabulary (OOV) rate, and token sequence length.

Tokenization design decision framework (4/4)

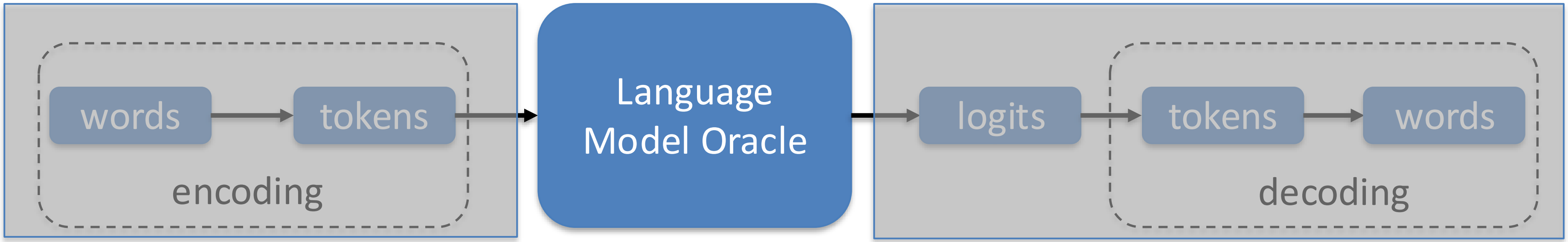
Formally, let T be the tokenizer, V the vocabulary, and S the set of input sentences:

$$\min_{T} \mathbb{E}_{s \in S}\left[\text{len}\big(T(s)\big)\right] \quad \text{subject to} \quad |V| \leq N, \quad \text{OOV}(T, S) < \epsilon$$

where N is a vocabulary size constraint, and ϵ is a target OOV rate.

- Consider sociolinguistic factors; a tokenization scheme should not systematically disadvantage any language or dialect.

Next time...


words
tokens
encoding
Language Model Oracle
logits
tokens
words
decoding

Sources

Content derived from: J&M Ch. 2

Appendix: Code snippet

```
print(chr(0x00E9))
print(chr(0x0065) + chr(0x0301))
print(bytes.fromhex("c3 a9").decode("utf-8"))
print(bytes.fromhex("c3 a9 20").decode("utf-8"))
print(chr(0xFFFD))
print(bytes.fromhex("c3 a9 a9").decode("utf-8"))
```