# Text Preprocessing and Tokenization

PREPARED BY: AHMAD ALAA ALDINE

## Text Normalization

LOWERCASING, PUNCTUATION REMOVAL, STEMMING VS. LEMMATIZATION.

## How many words in a sentence?

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

17 words

- if we don’t count punctuation marks as words

19 if we count punctuation

## Vocabulary, Type and Instance

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

**Vocabulary**: all unique words in a given text or corpus

**Type**: an element of the vocabulary V

- The number of types is the vocabulary size **|V|**

**Instance**: an instance of that type in a given text.

- 15 types and 17 instances (if we ignore punctuation).

More questions: Are `They` and `they` the same type?

## The Problem: A  Vocabulary Explosion

In Natural Language Processing, treating every word variation as a unique token creates massive, sparse vocabularies.

As text corpora grow, the number of unique word types grows without a clear bound, posing  a huge problem for any computational model.

**Key Takeaway:** To build robust models, we must group different forms of the same word. This process is called **Text Normalization.**

## Lowercasing

Lowercasing in NLP is a text preprocessing step that converts all text to lowercase to standardize words (e.g., "They" becomes "they")

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

$|V| = 15$
$N = 17$

Lowercasing

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

$|V| = 14$
$N = 17$

● Reduce Vocabulary Size

## Proper names may lose identity!

`Apple released a new product in the US`

Lowercasing ↓

`apple released a new product in the us`

● Model can no longer distinguish:

- **Apple (Company)** vs apple (fruit)
- **US (Country)** vs us (pronoun)

# Punctuation Removal

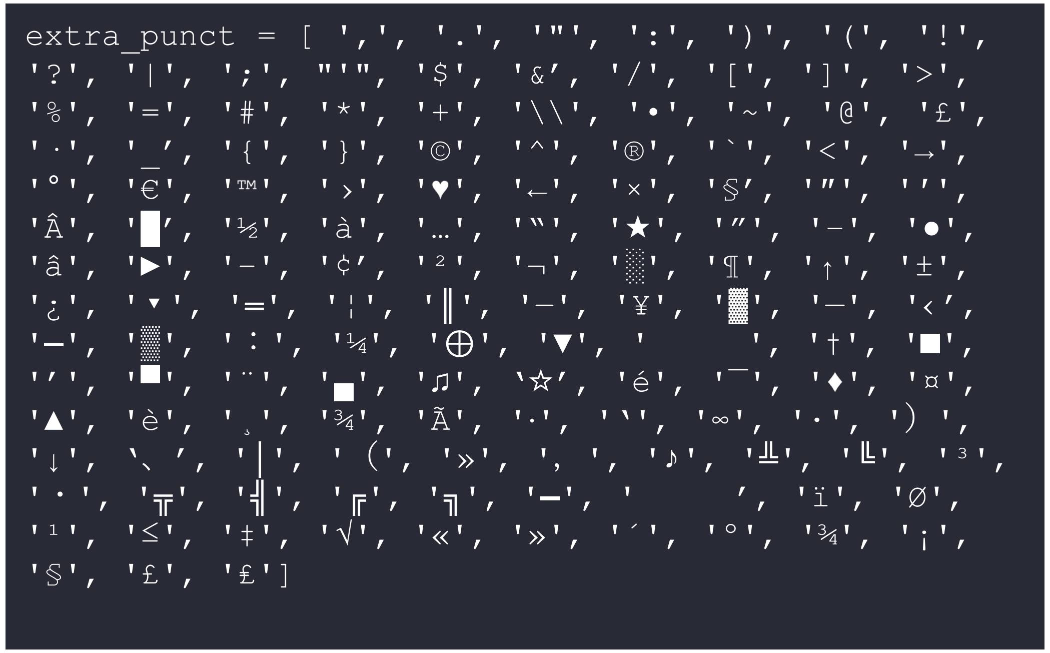
Punctuation removal in NLP is a crucial text preprocessing step that strips characters like periods, commas, and question marks.

We need to carefully choose the list of punctuation which we are going to discard based on the use case.

For example, Python's **string** module contains the following list of punctuation.

```
>>> import string
>>> string.punctuation
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

You can also add other special symbols to list which you want to discard.



```
extra_punct = [ ',', '.', '"', ':', ')', '(', '!',
'?', '|', ';', "'", '$', '&', '/', '[', ']', '>',
'%', '=', '#', '*', '+', '\\', '•', '~', '@', '£',
'·', '_', '{', '}', '©', '^', '®', '`', '<', '→',
'°', '€', '™', '›', '♥', '←', '×', '§', '″', '′',
'Â', '█', '½', 'à', '…', '“', '★', '”', '–', '●',
'â', '►', '−', '¢', '²', '¬', '░', '¶', '↑', '±',
'¿', '▾', '═', '¦', '║', '―', '¥', '▓', '—', '‹',
'─', '▒', '：', '¼', '⊕', '▼', '          ', '†', '■',
'′', '▀', '¨', '▄', '♫', '☆', 'é', '¯', '♦', '¤',
'▲', 'è', '¸', '¾', 'Ã', '⋅', '‘', '∞', '∙', '）',
'↓', '、', '│', '（', '»', '，', '♪', '╩', '╚', '³',
'・', '╦', '╣', '╔', '╗', '▬', '          ', 'ï', 'Ø',
'¹', '≤', '‡', '√', '«', '»', '´', 'º', '¾', '¡',
'§', '£', '₤']
```

## Why Remove Punctuation in Text Preprocessing?

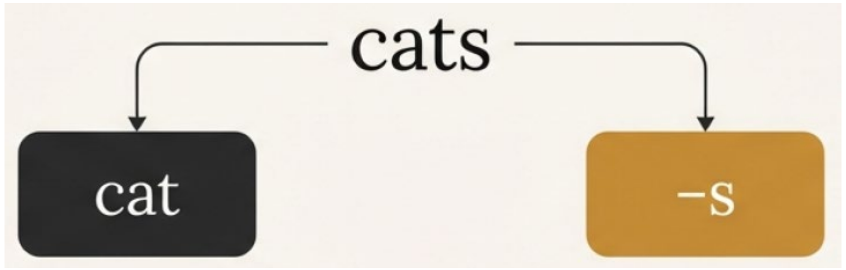
**Reduces Noise:** Punctuation adds characters that often lack meaning.

**Simplifies Data:** Creates fewer unique types, reducing vocabulary size and computational load.

**Improves Accuracy:** Prevents models from learning false distinctions (e.g., "hello" vs. "hello!").

# The Building Blocks of Meaning: Introducing Morphemes

A **morpheme** is the minimal meaning-bearing unit in a language. Words are composed of one or more morphemes.



(the **root** morpheme, supplying the main meaning)

(an **affix,** indicating plural)

Morphemes are everywhere. The sentence below is segmented to show its morphemic structure:

Doc work - ed care - ful - ly wash - ing the glass - es

## Two Classes of Affixes

### Inflectional Morphemes

**Function:** Create different grammatical ***forms* of the same** word. They play a syntactic role (e.g., tense, number).

**Examples:**

**-s** / **-es** for plural nouns (cat cats)

**-ed** for past tense verbs (work worked)

### Derivational Morphemes

**Function:** Create ***new*** words, often of a different grammatical class.

**Examples:**

**-ly** to create an adverb (careful carefully)

**-ation** to create a noun (organize organization)

## Another Type of Morpheme: Clitics

A morpheme that acts syntactically like a word but:

- is reduced in form
- and attached to another word

English: `'ve` in `I've` (`'ve` can't appear alone)

English: `’s` in `the teacher’s book`

French: `l’` in `l’opera`

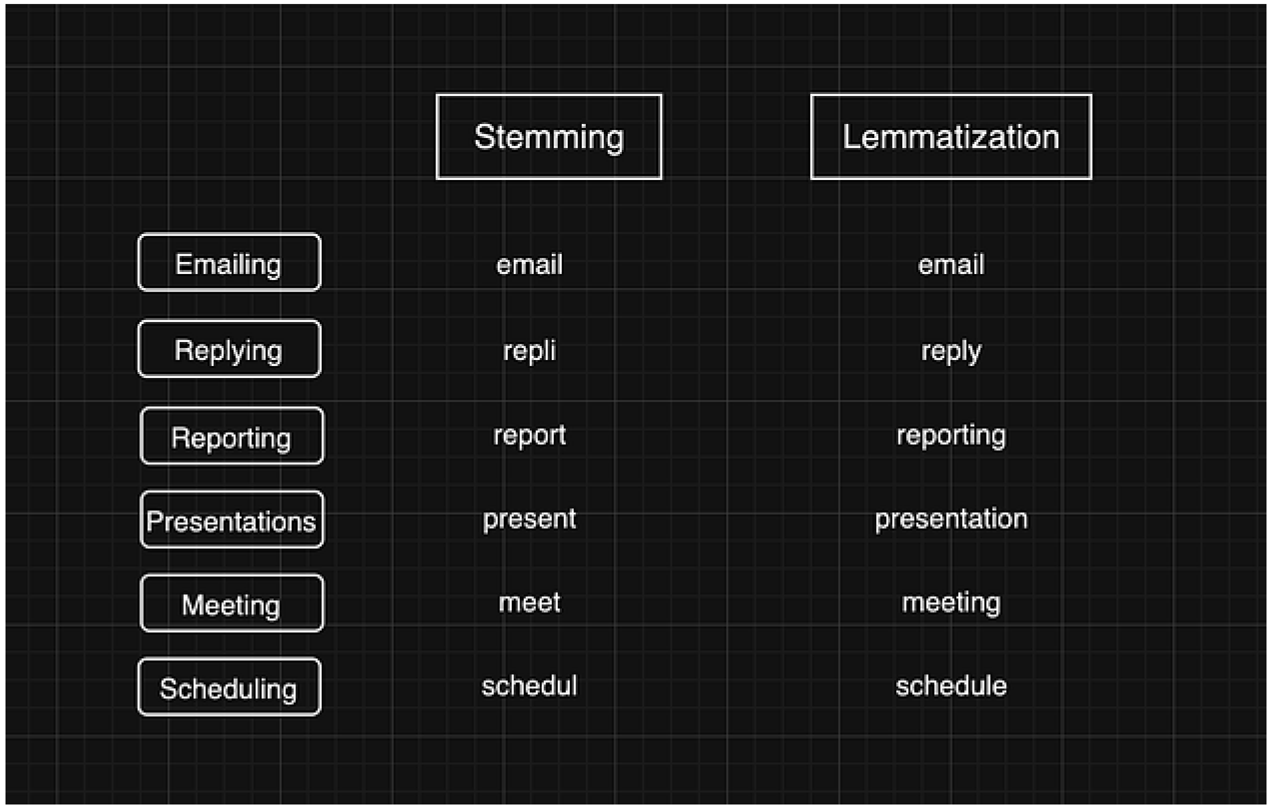
Arabic: ب ‘by/with’, و ‘and’.

## Stemming and Lemmatization

Lemmatization and Stemming are normalization techniques used in NLP to reduce the word to its’ base (root) form.

Stemming uses a heuristic approach to reduce the word to its’ root form.

Lemmatization uses linguistic knowledge (dictionaries, context) to find the actual dictionary word (lemma).



| | Stemming | Lemmatization |
|---|---|---|
| Emailing | email | email |
| Replying | repli | reply |
| Reporting | report | reporting |
| Presentations | present | presentation |
| Meeting | meet | meeting |
| Scheduling | schedul | schedule |

# Stemming vs. Lemmatization

## Stemming

Method: Heuristic, rule-based suffix removal (e.g., Porter Stemmer).

Output: A “root" that might not be a real word (e.g., "caring" -> "car", "studies" -> "studi").

Pros: Fast, efficient for large datasets, lower computational cost.

Cons: Lower accuracy, can lose semantic meaning.

## Lemmatization

Method: Uses dictionaries (lexicon) and Part-of-Speech (POS) tagging to find the canonical form (lemma) (e.g., WordNet Lemmatizer).

Output: A valid, meaningful dictionary word (e.g., "caring" -> "care", "better" -> "good").

Pros: High accuracy, preserves meaning, consistent output.

Cons: Slower, computationally intensive, requires more resources (dictionaries).

POS tagging: assigning a grammatical category to each word.
To be covered later →

## The Core Trade-Off: Speed vs. Accuracy

**Stemming: FAST & SIMPLE**

Reduces feature space quickly.

Output is not linguistically correct or easily interpretable.

**Lemmatization: ACCURATE & INTERPRETABLE**

Produces valid words, enabling deeper semantic analysis.

Computationally expensive and requires more linguistic resources (lexicon, POS tags).

Your choice depends entirely on the requirements of your specific NLP application.

# Tokenization Strategies

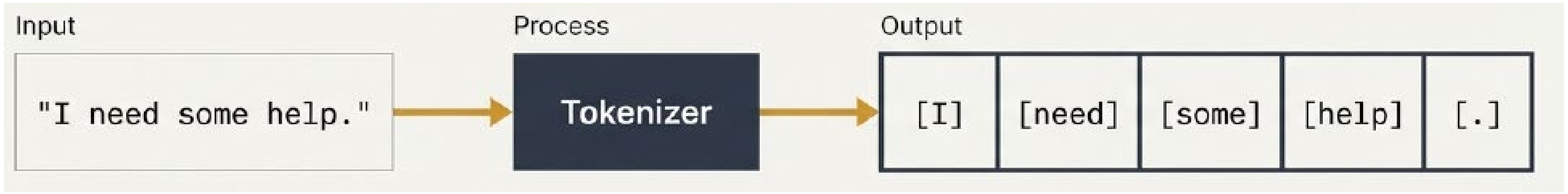
A JOURNEY FROM RULE-BASED METHODS TO THE SUBWORD REVOLUTION

# Text is unstructured
# Machine understand numbers

The first step in any modern NLP pipeline is converting raw text into a sequence of  discrete units, called tokens.

This fundamental task is called tokenization.

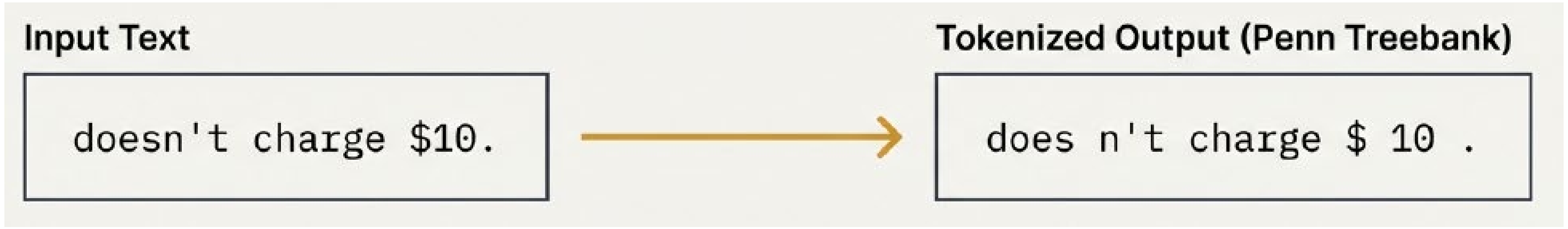
It is the bridge between human language and machine understanding.



## Treat Words as Tokens: Early methods

Early methods are rule-based methods that rely on handcrafted rules to define word boundaries.

The **Penn Treebank** tokenization standard, for example, uses a sophisticated set of rules to handle punctuation and clitic contractions in English.



```
doesn't charge $10.
```

```
does n't charge $ 10 .
```

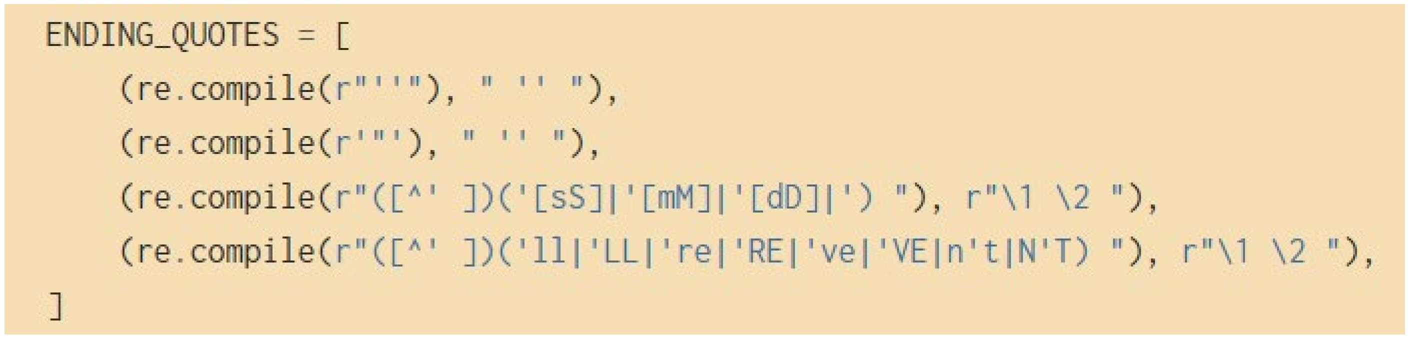
# The Penn Treebank

The **Penn Treebank tokenizer** uses regular expressions to tokenize text.

This tokenizer performs the following steps:

- split standard contractions, e.g. ``don't`` -> ``do n't`` and ``they'll`` -> ``they 'll``
- treat most punctuation characters as separate tokens
- split off commas and single quotes, when followed by whitespace
- separate periods that appear at the end of line

Examples of **Penn Treebank** regular expressions to handle clitics:



```
ENDING_QUOTES = [
    (re.compile(r"''"), " '' "),
    (re.compile(r'"'), " '' "),
    (re.compile(r"([^' ])('[sS]|'[mM]|'[dD]|') "), r"\1 \2 "),
    (re.compile(r"([^' ])('ll|'LL|'re|'RE|'ve|'VE|n't|N'T) "), r"\1 \2 "),
]
```

# Beyond Rule-Based Tokenization

## RULE-BASED TOKENIZATION

**Penn Treebank Tokenization**
- Rule-based
- Deterministic
- Language-specific

**Limitations**
- Hard-coded rules
- Difficult to adapt to new languages
- Edge cases remain

## STATISTICAL-BASED TOKENIZATION

- Learns token boundaries from data
- Uses context and frequency
- Adapts better across domains

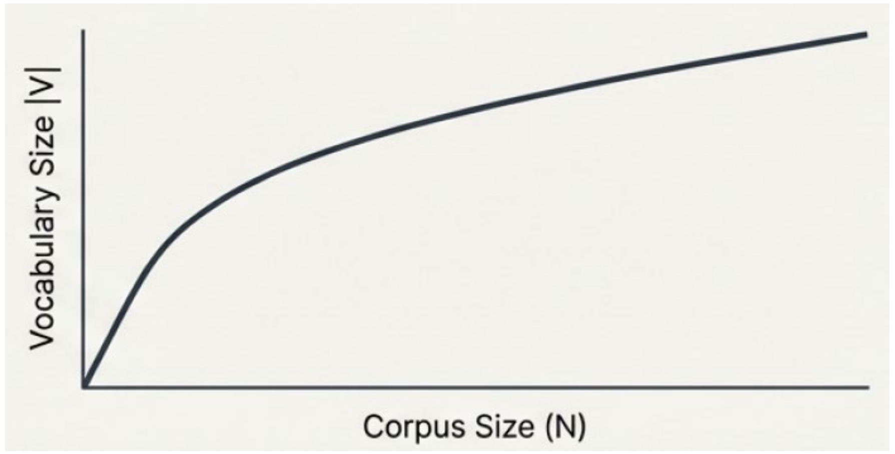
**Examples**
- Learned tokenizers in modern NLP tools (**BPE**)
- Punkt tokenizer (sentence-level)
- HMM / CRF-based word segmentation

# Word-Level Tokenization Limitations

Relying on words as tokens presents two critical failures that limit **scalability** and **cross-lingual applicability**.

## Scalability

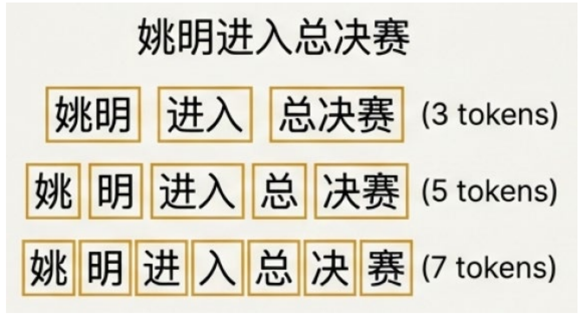
The vocabulary explosion. The number of unique words in a language grows unboundedly as we process more text.



## Cross-lingual Applicability

How do you segment languages without spaces?

In Chinese, 姚明进入总决赛 “Yao Ming reaches the finals” has multiple valid word tokenization.



姚明进入总决赛
姚明 进入 总决赛 (3 tokens)
姚 明 进入 总 决赛 (5 tokens)
姚 明 进 入 总 决 赛 (7 tokens)

## [UNK] – Uknown Words

Even a massive training corpus will not contain every  word.

UNK represents a word that is not in the model’s vocabulary, also known as outof-vocabulary (oov) words.

The UNK token serves as a universal placeholder for all such words, allowing the model to process any input text without encountering errors.

Training Vocabulary = {i, love, nlp}

New Input Sentence: "I love transformers“

After Tokenization: ["i", "love", "UNK"]

"transformers" was never seen during training

Problem caused by [UNK]

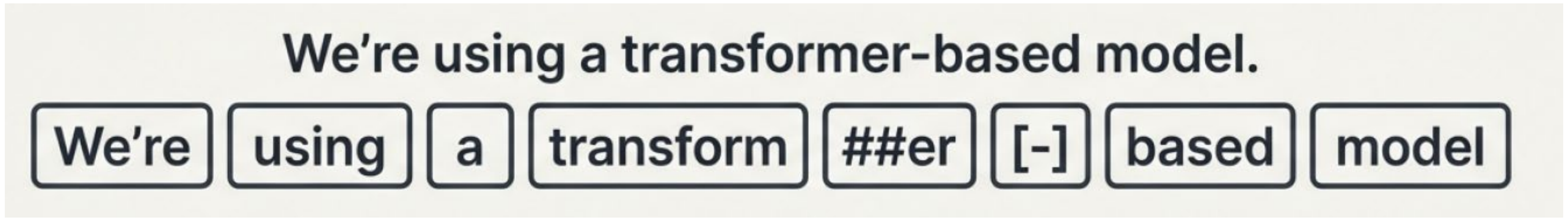
- Loss of semantic information
- Different words become indistinguishable

## Solution: Subword Tokenization

Subword tokenization: represent words with sequences of sub-units.

It addresses the failures of word-based methods by creating a fixed-size vocabulary of reusable parts.

These subwords are often morphemes (meaningful units like '-er', 're-') or other frequently occurring character sequences.



# Subword Tokenization: Advantages

Eliminates '[UNK]’

- Any new or rare word can be constructed from known subword pieces

Fixed Vocabulary

- The vocabulary size is a fixed hyperparameter, preventing uncontrolled growth.

Captures Morphology

- Related words like 'low 'lower and' lowest' can share the 'low' subword token, allowing the model to see their connection.

## Modern Tokenizers

### Byte-Pair Encoding (BPE)

- Iteratively merges the most frequent pairs of characters or character sequences into a single token.
- It is the standard for models like **GPT** and **LLaMA**.

### WordPiece

- Similar to BPE but merges pairs based on maximizing the likelihood of the training data rather than just frequency.
- It is used by **BERT** and **DistilBERT**.

### Unigram Language Modeling (ULM)

- A probabilistic approach that starts with a large vocabulary and removes tokens that least impact the overall likelihood of the training set.

# Byte-Pair Encoding (BPE)

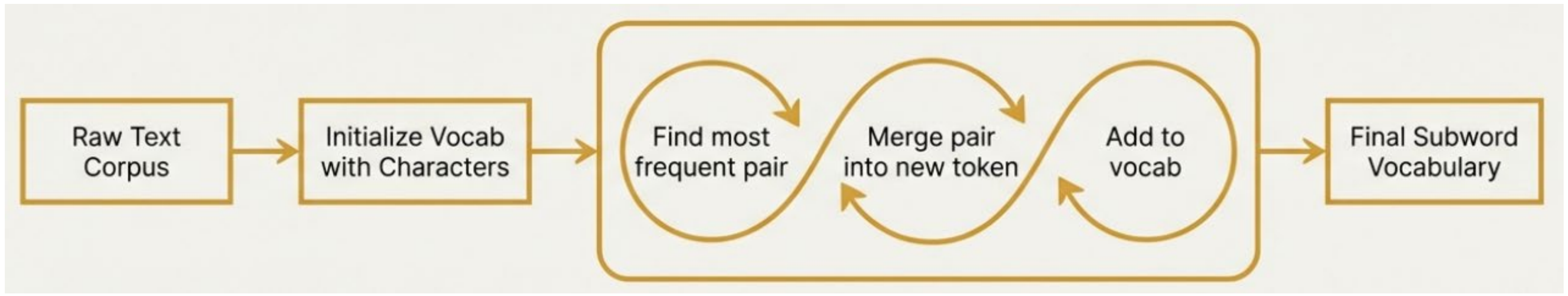
BPE builds a vocabulary by learning the most efficient ''text compression‘’.

Originally a data compression algorithm, it was adapted for NLP tokenization.

Its core principle is powerful and requires no linguistic expertise.

Start with individual characters, and iteratively merge the most frequently occurring adjacent pair of tokens.

The process is repeated for a predetermined number of merges, building up a vocabulary of common subwords.



## BPE Training Algorithm

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

## Training the Vocabulary from Data (Part 1)

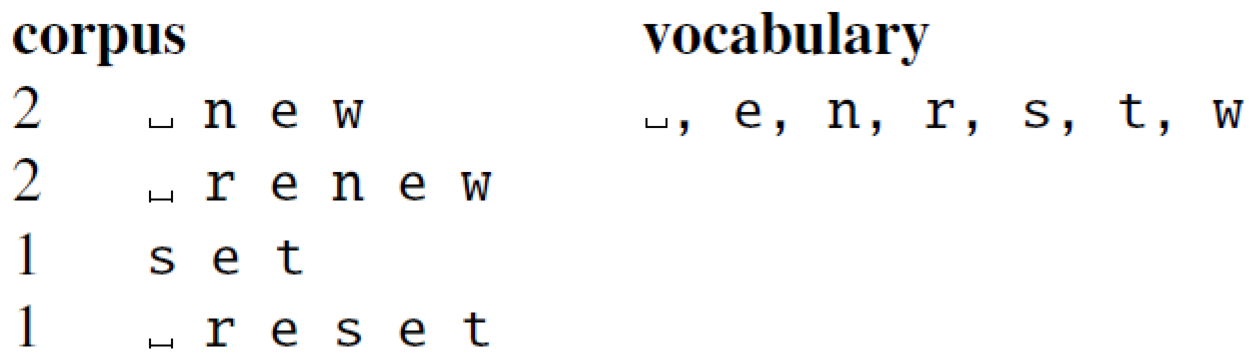
Let's trace the BPE training process on a small, synthetic corpus.

**Corpus**: “set new new renew reset renew”

First, break up the corpus into words, with leading whitespace, together with their counts.

Merges are only allowed within the initial word boundaries.

**Initial State:**

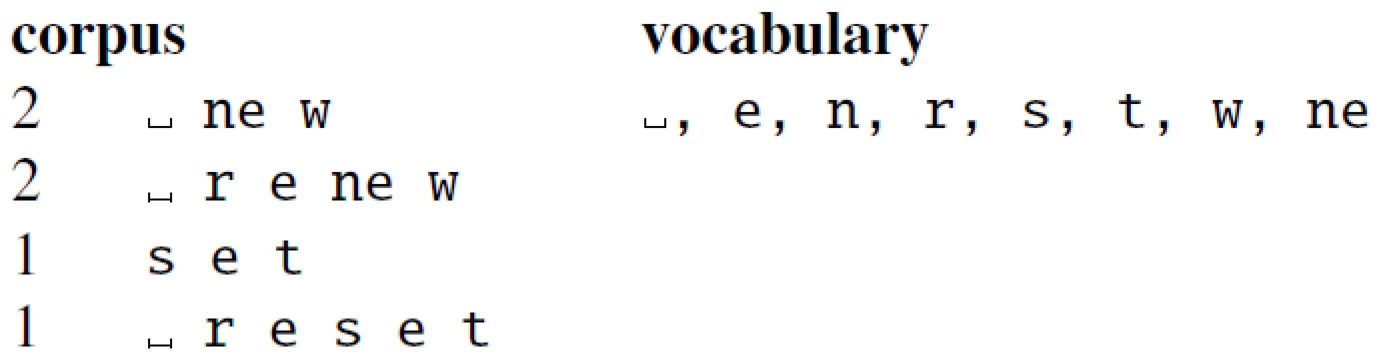


| corpus | | vocabulary |
|---|---|---|
| 2 | ␣ n e w | ␣, e, n, r, s, t, w |
| 2 | ␣ r e n e w | |
| 1 | s e t | |
| 1 | ␣ r e s e t | |

# Training the Vocabulary from Data (Part 2)

**Step 1: First merge**

The BPE training algorithm first counts all pairs of adjacent symbols:
the most frequent is the pair n e because it occurs in “**new”** (frequency of 2) and “**renew”** (frequency of 2) for a total of 4 occurrences.
We then merge these symbols, treating **ne** as one symbol, and count again:

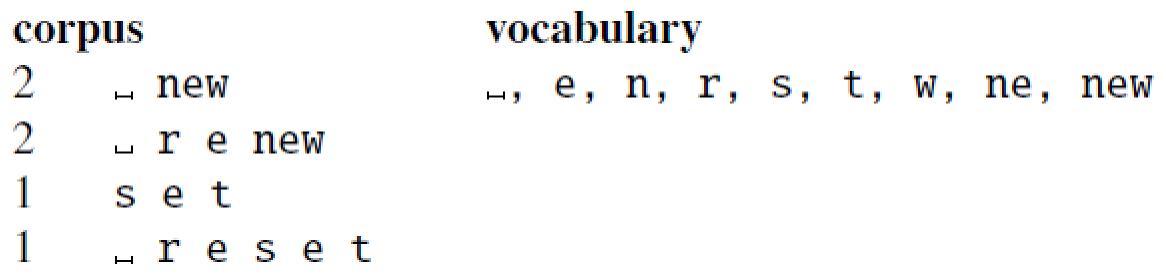


| corpus | | vocabulary |
|---|---|---|
| 2 | ␣ ne w | ␣, e, n, r, s, t, w, ne |
| 2 | ␣ r e ne w | |
| 1 | s e t | |
| 1 | ␣ r e s e t | |

# Training the Vocabulary from Data (Part 3)

**Step 2: Second merge**

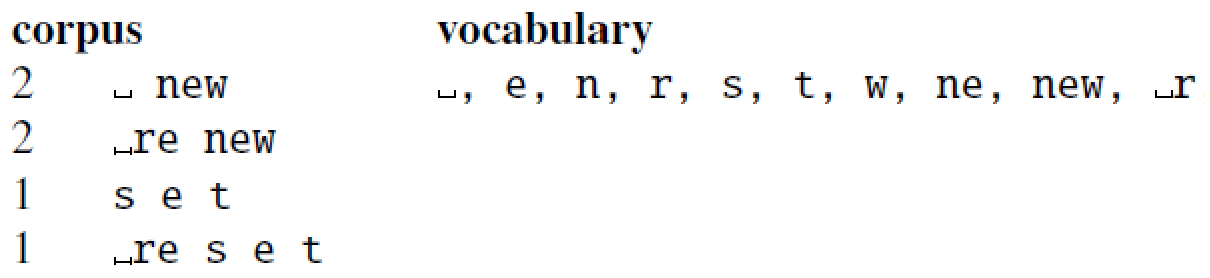
Now the most frequent pair is ne w (total count=4), which we merge.



| corpus | | vocabulary |
|---|---|---|
| 2 | ␣ new | ␣, e, n, r, s, t, w, ne, new |
| 2 | ␣ r e new | |
| 1 | s e t | |
| 1 | ␣ r e s e t | |

**Step 3: Third merge**

Next ␣ and r (total count of 3) get merged to ␣r.



| corpus | | vocabulary |
|---|---|---|
| 2 | ␣ new | ␣, e, n, r, s, t, w, ne, new, ␣r |
| 2 | ␣re new | |
| 1 | s e t | |
| 1 | ␣re s e t | |

# Training the Vocabulary from Data (Part 4)

**Step 4: Fourth merge**

Next ␣r and e (total count of 3) get merged to ␣re.

| corpus | | vocabulary |
|---|---|---|
| 2 | ␣ new | ␣, e, n, r, s, t, w, ne, new, ␣r, ␣re |
| 2 | ␣re new | |
| 1 | s e t | |
| 1 | ␣re s e t | |

If we continue, the next merges are:

| merge | current vocabulary |
|---|---|
| (␣, new) | ␣, e, n, r, s, t, w, ne, new, ␣r, ␣re, ␣new |
| (␣re, new) | ␣, e, n, r, s, t, w, ne, new, ␣r, ␣re, ␣new, ␣renew |
| (s, e) | ␣, e, n, r, s, t, w, ne, new, ␣r, ␣re, ␣new, ␣renew, se |
| (se, t) | ␣, e, n, r, s, t, w, ne, new, ␣r, ␣re, ␣new, ␣renew, se, set |

# BPE Encoder

Once we've learned our vocabulary, the BPE encoder is used to tokenize a test sentence.

The encoder just runs on the test data the merges we have learned from the training data.

It runs them in the order we learned them.

Example:

Input word "**newest**" , Merge List: `␣, e, n, r, s, t, w, ne, new, ␣r, ␣re, ␣new, ␣renew, se, set`

**Initial representation**: Each character is a token.

**Tokens**: "n" "e" "w" "e" "s" "t"

**Step 1:** Apply Merge Rule #1 (n + e -> ne)

**Tokens:** "ne" "w" "e" "s" "t"

**Step 2:** Apply Merge Rule #2 (ne + w -> new)

**Tokens:** "new" "e" "s" "t"

**No more merge rules to apply**

## Sentence Segmentation

## From Tokens to Sentences

Tokenization is not only about words.

Many NLP tasks operate at the sentence level such as Translation.

Sentence segmentation is the task of identifying sentence boundaries in text.

Dr. Smith arrived at 5 p.m. He left early.

Naive split fails

Intelligent segmentation required

## Why Sentence Segmentation is Hard

Sentence segmentation depends on the language and the genre.

The most useful cues for segmenting a text into sentences in English written text tend to be punctuation, like periods ".", question marks "?", and exclamation points "!".

Question marks "?" and exclamation points "!" are relatively unambiguous markers of sentence boundaries, and simple rules can segment sentences when they appear.

Period "." is quite ambiguous

- Sentence boundary
- Abbreviations like Inc. or Dr.
- Numbers like .02% or 4.3

# Sentenene Segmentation Methods

Many English sentence segmentation methods work by first deciding whether a period is part of the word or is a sentence-boundary marker.

- Often based on deterministic rules (Stanford CoreNLP sentence splitter)
- But sometimes via machine learning (NLTK Punkt tokenizer)

**Key Takeaway:** Rule-based methods are fast and simple, but they are language-specific. Machine Learning methods generalize better.

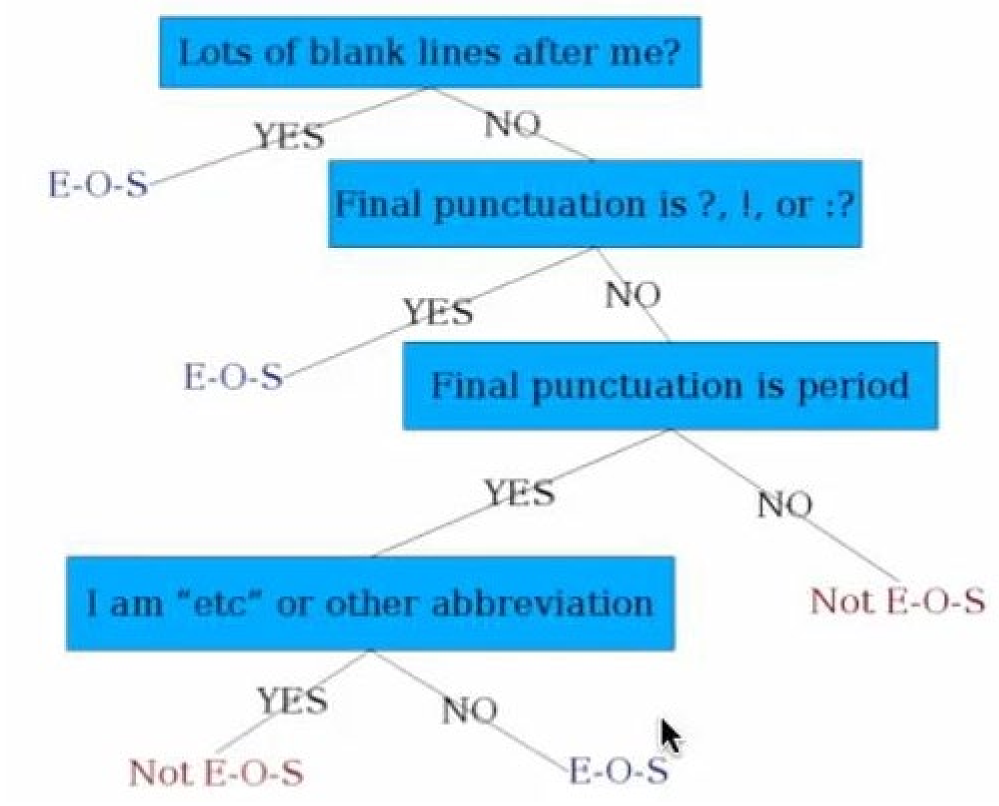
# The Simplest Method: Decision Tree

*A Decision Tree is simply an if-else statement.*

The interesting research in a decision tree is choosing the features.

*Lots of modifications can be made to make a more sophisticated decision tree such as:*

- Case of word with ".": Upper, Lower, Cap, Number
- Case of word after ".": Upper, Lower, Cap, Number
- Length of word with "."


Lots of blank lines after me?
YES
NO
E-O-S
Final punctuation is ?, !, or :?
YES
NO
E-O-S
Final punctuation is period
YES
NO
Not E-O-S
I am "etc" or other abbreviation
YES
NO
Not E-O-S
E-O-S

## Stanford CoreNLP Sentence Splitter: Core Rules

A sentence *may* end at: “.”, “!” and “?”

Periods do NOT end a sentence if they belong to:

- Titles
- Common Abbreviations
- Acronyms

Periods inside numbers are ignored.

Periods after single-letter initials do not trigger a split.

- e.g: John F. Kennedy was elected.

Ignore sentence boundaries inside quotes and parentheses

- e.g: She said, "Stop."

## Punkt Sentence Tokenizer

Punkt is an **unsupervised** statistical sentence segmentation algorithm.

- Learns abbreviations and sentence boundaries
- No manually labeled data required

Punkt decides based on context and frequency however a period "." is:

- Sentence end
- Abbreviation
- Decimal point

## Training Phase & Decision Phase

### What Punkt Learns Automatically

- Common abbreviations (Dr., Mr., etc.)
- Typical sentence starters
- Token context around punctuation

### Decision Rules

- Is the token frequently abbreviated?
- Does the next token look like a sentence start?
- Is capitalization consistent?

**Punkt assigns probability, not hard rules**

# Strengths and Limitations of Punkt

## STRENGTHS

Language-adaptive

No labeled data

Handles abbreviations well

## LIMITATIONS

Needs sufficient training text

Domain-specific abbreviations may fail

Not designed for informal text (tweets, chats)