# Unicode

## Lecture 5

Adarsh Kumar
Department of Industrial Mining Engineering and ICT (EMIT),
Manresa School of Engineering (EPSEM),
Polytechnic University of Catalonia, Manresa, Barcelona, Spain
adarsh.kumar@upc.edu

May 5, 2026

# What is WordPiece Tokenization?

- WordPiece is a **subword tokenization algorithm** used in NLP.
- Developed for the **BERT model** by Google.
- Splits words into smaller **subwords** to handle:
  - Rare words
  - Morphological variations
  - Out-of-vocabulary words
- Difference from BPE: merges are based on **maximizing likelihood** instead of just frequency.

## Key Characteristics

- **Character-level initialization:** Start with all characters as tokens.
- **Greedy subword merging:** Merge pairs that maximize corpus likelihood.
- **Handling unknown words:** Use ‘‘ prefix for subwords continuing a previous token.
- **Vocabulary:** Contains full words, common subwords, and characters for rare words.

## How WordPiece Works (Conceptually)

1. Initialize vocabulary with all characters.
2. Segment corpus words using existing vocabulary tokens.
3. Merge subword pairs that maximize likelihood.
4. Repeat until desired vocabulary size is reached.
5. Tokenize new text using learned subwords.

# Example: WordPiece Tokenization

**Corpus:** "unaffable"

- Start with characters: `u n a f f a b l e`
- Merge frequent pairs: `a f` → `af`, `f f` → `ff`, `aff` formed
- Merge further: `un` → `un`, `able` → `able`

**Resulting tokens:**

"unaffable" → "un", "aff", "able"

**New word example:** "unaffably"

"unaffably" → "un", "aff", "able", "ly"

(‘‘ indicates continuation from a previous subword)

**Advantages of WordPiece**

- Handles rare and unknown words efficiently.
- Reduces vocabulary size while covering most words and subwords.
- Likelihood-based merging produces better subword units than pure frequency.
- Widely used in transformer-based models like BERT, RoBERTa, and ALBERT.

# What is Unigram LM Tokenization?

- A **probabilistic subword tokenization algorithm** used in NLP.
- Treats tokenization as a **likelihood maximization problem**.
- Unlike BPE, it chooses subwords that maximize the overall likelihood of the corpus.
- Used in models like **SentencePiece (Unigram mode)**, **T5**.

## Key Characteristics

- **Probabilistic approach:** Each subword has a probability; segments maximize likelihood.
- **Subword-based:** Rare words split into smaller subwords, frequent words may remain intact.
- **Vocabulary pruning:** Low-probability subwords removed iteratively.
- **Handles unknown words efficiently:** Unseen words split into known subwords.

# How Unigram LM Tokenization Works (Conceptually)

1. Generate a large **candidate subword vocabulary**.
2. Assign probabilities to each subword.
3. Segment each word into subwords that **maximize corpus likelihood**.
4. Iteratively prune low-probability subwords until reaching desired vocabulary size.
5. Tokenize new text using the final vocabulary.

## Example: Unigram LM Tokenization Corpus: "unaffable unreadable"

- Candidate subwords:

$$["un","aff","able","read","able","ing",...]$$

- Segment "unaffable" $\rightarrow$ "un" + "aff" + "able"
- Segment "unreadable" $\rightarrow$ "un" + "read" + "able"
- Low-probability subwords (unused) are pruned from the vocabulary.

# Advantages of Unigram LM Tokenization

- **Probabilistic segmentation:** Finds subword splits that best represent the corpus.
- **Efficient for rare and unknown words.**
- **Flexible vocabulary size:** Can prune low-probability subwords.
- **Language-independent:** Works for any language or script.
- Used in transformer models like **T5**, **mBERT**, and **SentencePiece (Unigram mode)**.

# What is Byte-Level BPE?

- A variation of BPE that works on **bytes** instead of characters.
- Used in models like **GPT-2** and **GPT-3**.
- Handles **any text**, including emojis, symbols, and non-Latin scripts.
- Avoids issues with unknown or special Unicode characters.

## Key Characteristics

- **Byte-level representation:** Every character, emoji, or symbol is a sequence of bytes.
- **Subword merging using BPE:** Frequent byte pairs are merged iteratively to form subwords.
- **Language-independent:** Works for any language or script.
- **No unknown tokens:** Every input can be represented as bytes.

# How Byte-Level BPE Works (Conceptually)

1. Convert text into **UTF-8 bytes**.
2. Apply **BPE merges** on byte sequences:
   - Merge the most frequent byte pairs iteratively.
3. Encode new text using learned byte-level subwords.
4. Decode tokens by converting bytes back to characters.

**Example: Byte-Level BPE Tokenization Text:** "Hello [wave emoji]"

- Convert to UTF-8 bytes: `H e l l o <space> F0 9F 91 8B` ("[wave emoji]" is 4 bytes)
- Merge frequent byte pairs using BPE: `H e l l o <space> [wave emoji]`
- Tokenization result:

$$[\text{"}Hello\text{"}, \text{""}, \text{"}[waveemoji]\text{"}]$$

**Works for any language, symbol, or emoji without unknown tokens.**

# Advantages of Byte-Level BPE

- Handles **all Unicode characters**, symbols, and emojis.
- **Language-independent**: works for Latin, Chinese, Arabic, etc.
- **No unknown tokens**: every input can be represented.
- **Robust for large-scale models**: used in GPT-2, GPT-3.
- Preserves BPE efficiency while supporting arbitrary text.

# What is Character-Level Tokenization?

- Splits text into **individual characters** as tokens.
- Each character, digit, punctuation, and space is treated as a separate token.
- Simplest and most fine-grained form of tokenization.

## Key Characteristics

- **Fine-grained tokens:** Every character is a token.
- **Language-independent:** Works for any script.
- **Handles unknown words automatically:** No ‘<UNK>‘ tokens.
- **Useful for certain tasks:** OCR, speech recognition, complex morphology, character-level language models.

# How Character-Level Tokenization Works

1. Take input text. Example: "I love NLP!"
2. Split each character individually into tokens:

$$[\text{"}I\text{"}, \text{""}, \text{"}l\text{"}, \text{"}o\text{"}, \text{"}v\text{"}, \text{"}e\text{"}, \text{""}, \text{"}N\text{"}, \text{"}L\text{"}, \text{"}P\text{"}, \text{"}!\text{"}]$$

3. Feed these tokens into a model (RNN, LSTM, or Transformer) for learning character-level patterns.

**Example: Character-Level Tokenization Text:** "ChatGPT"

$$[\text{"}C\text{"}, \text{"}h\text{"}, \text{"}a\text{"}, \text{"}t\text{"}, \text{"}G\text{"}, \text{"}P\text{"}, \text{"}T\text{"}]$$

**Text:** "Hello, world!"

$$[\text{"}H\text{"}, \text{"}e\text{"}, \text{"}l\text{"}, \text{"}l\text{"}, \text{"}o\text{"}, \text{"}, \text{"}, \text{"}, \text{""}, \text{"}w\text{"}, \text{"}o\text{"}, \text{"}r\text{"}, \text{"}l\text{"}, \text{"}d\text{"}, \text{"}!\text{"}]$$

**Spaces and punctuation are separate tokens.**

# Advantages of Character-Level Tokenization

- **No unknown words:** Every word represented as characters.
- **Language-agnostic:** Works for any script, symbols, or emojis.
- **Captures morphology:** Prefixes, suffixes, roots can be learned naturally.
- **Small vocabulary size:** Only characters and punctuation needed.

# What is Moses Tokenizer?

- A widely-used **rule-based word-level tokenizer**.
- Developed for **statistical machine translation (SMT)** as part of the Moses toolkit.
- Splits text into words/tokens using **language-specific rules**.
- Handles punctuation, contractions, and special characters consistently.

## Key Characteristics

- **Rule-based:** Deterministic tokenization using predefined rules.
- **Handles punctuation and symbols:** Example: "Hello, world!" $\rightarrow$ "Hello", ",", "world", "!"
- **Handles contractions:** Example: "don't" $\rightarrow$ "do" + "n't"
- **Language-specific rules:** English, French, German, etc.
- Widely used in NLP pipelines for machine translation and text normalization.

# How Moses Tokenizer Works (Conceptually)

1. Input text: "I don't like NLP!"
2. Apply tokenization rules:
   - Separate punctuation and symbols
   - Split contractions
3. Output tokens:
$$[\text{"}I\text{"}, \text{"}do\text{"}, \text{"}n't\text{"}, \text{"}like\text{"}, \text{"}NLP\text{"}, \text{"}!\text{"}]$$
4. Tokens can then be fed into MT models or NLP pipelines.

**Example: Moses Tokenization** Text: "Hello, world! How's it going?"

- Tokenized output:

$$[\text{"}Hello\text{"}, \text{"},\text{"}, \text{"}world\text{"}, \text{"}!\text{"}, \text{"}How\text{"}, \text{"}'s\text{"}, \text{"}it\text{"}, \text{"}going\text{"}, \text{"}?\text{"}]$$

- Contractions like "How's" $\rightarrow$ "How" + "'s"
- Punctuation is treated as separate tokens.

# Advantages of Moses Tokenizer

- **Deterministic and consistent:** Same input always produces the same tokens.
- Handles punctuation, contractions, and special symbols correctly.
- Applies language-specific rules for better tokenization than whitespace tokenizers.
- Widely used in machine translation preprocessing and NLP pipelines.

# What is Jieba Tokenizer?

- A popular Python-based word segmentation tool for **Chinese text**.
- Handles languages without spaces between words.
- Combines **dictionary-based** and **statistical methods** for tokenization.
- Widely used in NLP pipelines and search engines.

## Key Characteristics

- **Dictionary-based segmentation:** Uses a pre-built word dictionary.
- **HMM for unknown words:** Predicts new or rare words based on character sequences.
- **Multiple modes:**
  - Precise mode: Most accurate segmentation.
  - Full mode: Finds all possible words.
  - Search mode: Optimized for search indexing.
- Efficient for Chinese text without spaces.

# How Jieba Tokenizer Works (Conceptually)

1. Input Chinese text: 我喜欢学习 NLP
2. Apply dictionary and HMM-based segmentation.
3. Output tokens:
   我 / 喜欢 / 学习 NLP
4. Full Mode may produce overlapping tokens:
   我 / 喜欢 / 学 / 学习 / 习 NLP

**Example: Jieba Tokenization** Text: 结巴分词是中文分词工具

- Precise Mode: 结巴 / 分词 / 是 / 中文 / 分词 / 工具
- Full Mode: 结巴 / 分词 / 是 / 中 / 中文 / 文分词 / 工具
- Search Mode: 结巴 / 分词 / 是 / 中文 / 分词 / 工具

# Advantages of Jieba Tokenizer

- Handles **Chinese text without spaces** efficiently.
- Supports both **dictionary-based and statistical segmentation**.
- Flexible tokenization modes: Precise, Full, Search.
- Recognizes **new words** using HMM.
- Widely used in **Chinese NLP preprocessing, search engines, and text analytics**.

# What is spaCy Tokenizer?

- A modern, rule-based tokenizer used in the **spaCy NLP library**.
- Designed for industrial-strength NLP pipelines.
- Primarily word-level tokenization with linguistic awareness.
- Supports multiple languages (English, German, Spanish, etc.).

## Key Characteristics

- **Rule-based with exceptions**
  - Prefix rules
  - Suffix rules
  - Infix rules
  - Special-case exceptions
- Handles punctuation and contractions.
- Language-specific tokenization rules.
- Integrated with POS tagging, parsing, NER.

# How spaCy Tokenizer Works (Conceptually)

1. Input text: "I don't like NLP!"
2. Apply prefix, suffix, and infix rules.
3. Apply special-case exceptions.
4. Output tokens:
   I / do / n't / like / NLP / !

**Example 1** Text: "Hello, world!"

- Tokenized output:
  Hello / , / world / !
- Punctuation is separated as independent tokens.

# Example 2

Text: "U.S.A. is a country."

- Tokenized output:
  U.S.A. / is / a / country / .
- Abbreviations handled using special-case rules.

## Advantages of spaCy Tokenizer

- Fast and efficient (optimized in Cython).
- Linguistically informed tokenization.
- Easily customizable rules.
- Integrated with full NLP pipeline.
- Suitable for production-level applications.

## Comparison with Other Tokenizers

- Regex Tokenizer → Simple pattern matching.
- Moses → Rule-based for machine translation.
- Jieba → Chinese segmentation.
- spaCy → Industrial NLP pipeline tokenizer.

# What is NLTK Tokenizer?

- Part of the **Natural Language Toolkit (NLTK)** library.
- One of the most widely used NLP libraries for teaching and research.
- Provides multiple tokenization methods:
  - Word Tokenizer
  - Sentence Tokenizer
  - Regex Tokenizer
  - Treebank Tokenizer

## Key Characteristics

- Primarily rule-based tokenization.
- Handles punctuation and contractions.
- Sentence segmentation using pre-trained models (Punkt).
- Easy to use for educational purposes.

# Word Tokenization Example

Text: "I don't like NLP!"

- Output tokens:
  I / do / n't / like / NLP / !
- Based on Penn Treebank-style rules.

# Sentence Tokenization Example

Text: "NLP is interesting. It is widely used in AI."

- Sentence tokenizer output:
  Sentence 1: NLP is interesting.
  Sentence 2: It is widely used in AI.
- Uses Punkt sentence segmentation model.

# Different Tokenizers in NLTK

- **word_tokenize()** → General word tokenizer.
- **sent_tokenize()** → Sentence segmentation.
- **RegexpTokenizer** → Custom regex-based rules.
- **TreebankWordTokenizer** → Penn Treebank rules.

## Advantages of NLTK Tokenizer

- Excellent for teaching NLP concepts.
- Easy to experiment with different tokenizers.
- Lightweight and flexible.
- Large academic community support.

## Comparison with Other Tokenizers

- Regex → Simple pattern-based splitting.
- Moses → Machine translation preprocessing.
- spaCy → Industrial production NLP pipeline.
- NLTK → Educational and research-focused toolkit.

# Text Normalization Approaches in NLP

- Lowercasing
- Uppercasing
- Removing Punctuation
- Removing Numbers
- Removing Stopwords
- Stemming
- Lemmatization
- Accent/Diacritic Removal
- Expanding Contractions (e.g., "don't" $\rightarrow$ "do not")
- Removing Extra Whitespaces
- Unicode Normalization (NFC, NFD, NFKC, NFKD)
- Token Cleaning (removing URLs, emails, hashtags)
- Case Folding
- Text Standardization (e.g., British $\rightarrow$ American English)
- Spell Correction / Spelling Normalization

# Text Normalization Approaches in NLP

## Examples:

- **Lowercasing:** "NLP is Fun" → "nlp is fun"
- **Uppercasing:** "nlp is fun" → "NLP IS FUN"
- **Removing Punctuation:** "Hello, world!" → "Hello world"
- **Removing Numbers:** "I have 2 cats" → "I have cats"
- **Removing Stopwords:** "I love natural language processing" → "love natural language processing"
- **Stemming:** "playing, played, plays" → "play, play, play"
- **Lemmatization:** "playing, played, plays" → "play, play, play"
- **Accent/Diacritic Removal:** "café, résumé" → "cafe, resume"
- **Expanding Contractions:** "don't" → "do not"
- **Removing Extra Whitespaces:** "I love NLP" → "I love NLP"
- **Unicode Normalization:** ensures "\u00E9" (é) is consistent
- **Token Cleaning:** "Check https://example.com" → "Check"
- **Case Folding:** "NASA" → "nasa"
- **Text Standardization:** "colour" → "color"
- **Spell Correction:** "langauge" → "language"

# Tokenization vs Segmentation in NLP

**Definitions:**

- **Tokenization:** Splitting text into smaller units (words, subwords, characters).
- **Segmentation:** Dividing text into larger meaningful units (sentences, paragraphs, topics).

**Example Text:**
`I love NLP. It is very useful.`

**Tokenization (word-level):**
`["I", "love", "NLP", ".", "It", "is", "very", "useful", "."]`
**Segmentation (sentence-level):**
`["I love NLP.", "It is very useful."]`
**Subword Tokenization:**
`["I", "love", "NL", "P", ".", "It", "is", "very", "useful", "."]`
**Paragraph Segmentation (document-level):**
`["Paragraph 1: ...", "Paragraph 2: ..."]`

**Key Points:**

- Tokenization = micro-level, prepares text for NLP models.
- Segmentation = macro-level, organizes text for analysis, summarization, retrieval.
- Tokenization often occurs *inside* a segment.

# Text Segmentation Approaches in NLP

- Sentence Segmentation (Sentence Boundary Detection)
- Word Segmentation
- Subword Segmentation
- Tokenization-Based Segmentation
- Rule-Based Segmentation
- Statistical Segmentation
- Dictionary-Based Segmentation
- Byte Pair Encoding (BPE) Segmentation
- Unigram Language Model Segmentation
- Neural Network / Deep Learning Segmentation
- Maximum Matching (Greedy) Segmentation
- Viterbi-Based Segmentation
- CRF (Conditional Random Fields) Segmentation
- Semantic / Topic-Based Segmentation
- Paragraph / Discourse Segmentation

# Use of Regular Expressions (Regex) in NLP

**Definition:**
Regex are patterns used to match, search, and manipulate text. In NLP, they help in preprocessing, extraction, and normalization.

**Key Uses:**

- **Tokenization:** Split text into words or sentences
  `Text: "I love NLP." → Regex: \w+ → ["I","love","NLP"]`
- **Text Cleaning:** Remove URLs, emails, numbers
  `Text: "Visit https://example.com" → Regex: https?://§+ → "Visit"`
- **Pattern Matching / Extraction:** Extract structured info
  Phone: `\d{3}-\d{3}-\d{4}` Email: `§+@§+`
- **Normalization:** Standardize text
  Replace multiple spaces: `\s+ → " "`
- **Stemming / Lemmatization Assistance:**
  Suffix removal example: `ing$ → "playing" → "play"`
- **Feature Engineering:** Extract patterns for ML features
  Example: hashtags in tweets: `+`
- **Language Detection / Token Filtering:**
  Remove non-alphabetic: $[^{a} - zA - Z]$

**Advantages:** Powerful, flexible, language-agnostic.
**Limitations:** Can be complex, rule-based, limited semantic understanding.

# Looking Forward

## Current Trends

- **Unigram LM tokenization**: Alternative to BPE
- **Multilingual models**: Shared subword vocabularies
- **Character-level models**: For certain tasks
- **Byte-level approaches**: Eliminate Unicode complexities

## Active Research Areas

- Optimal vocabulary size for different tasks
- Language-specific vs. multilingual tokenizers
- Tokenization for low-resource languages
- Efficient tokenization for large-scale models

**Thank You! Questions?**