Unicode

Lecture 4

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 Unicode in NLP?

Unicode is a universal standard that assigns a unique number (code point) to every character, symbol, and emoji across all writing systems.

Languages & Scripts:

- **Languages:** Chinese, Arabic, Hindi, Cherokee, Ethiopic, Khmer, N'Ko...
- **Dead ones** like Sumerian cuneiform
- **Invented ones** like Klingon
- **Plus** emojis, currency symbols, etc.
- **150,000+** characters, **168** scripts, all world languages

Why Unicode Matters:

- Enables computers to consistently represent/process multilingual text
- Solves encoding chaos (Shift-JIS, Windows-1256 → mojibake)
- Every character gets one code point: `U+0041` = 'A'

Encoding Chaos

What is Encoding Chaos?

- Text is encoded in one character format
- But decoded using another format

Result:

- Garbled text
- Corrupted characters
- Unreadable output

Common Encodings:

- ASCII
- UTF-8
- ISO-8859-1 (Latin-1)
- UTF-16

Why Does Encoding Chaos Happen?

- Computers store text as **bytes**
- Each encoding standard maps bytes differently

Example Scenario:

- Text encoded in UTF-8
- But decoded as Latin-1

Encoding Chaos

What is Encoding Chaos?

- Text is encoded in one character format
- But decoded using another format

Result:

- Garbled text
- Corrupted characters
- Unreadable output

Common Encodings:

- ASCII
- UTF-8
- ISO-8859-1 (Latin-1)
- UTF-16

Why Does Encoding Chaos Happen?

- Computers store text as **bytes**
- Each encoding standard maps bytes differently

Example Scenario:

- Text encoded in UTF-8
- But decoded as Latin-1

Example: "café"

1. The word **café** is encoded in **UTF-8**.
   The character é becomes two bytes: `C3 A9`.
2. If those same bytes (`C3 A9`) are incorrectly decoded as **Latin-1**,
   - `C3` becomes Ã
   - `A9` becomes ©

So the output becomes:

cafÃ©

This mismatch between encoding and decoding formats causes corrupted text.

Real-World Examples

- café → cafÃ©
- Pokémon → PokÃ©mon
- ₹500 → â‚¹500
- 😊 → ðŸ˜Š

Encoding Chaos → Unicode Solution

The Problem

- **Shift-JIS**: Japanese encoding (1-2 bytes per character)
- **Windows-1256**: Arabic encoding (1 byte per character)
- **Many others**: GBK (Chinese), KOI8-R (Russian), etc.

Each mapped the **same byte values** (0x80-0xFF) to **different characters**.

Example: Same bytes, different results

Bytes: `0x82 0xA0`

System reading as...	Result
Shift-JIS (Japanese)	あ
Windows-1256 (Arabic)	غ
Wrong decoder	â or garbage

Result: Japanese text → Arabic gibberish (mojibake).

Unicode Solution

Unicode gives **one code point per character** regardless of language:

```
あ → U+3042 (Japanese hiragana)
غ → U+063A (Arabic letter ghayn)
```

Unicode Example: Chinese Text

Chinese phrase “ 语言处理 ” (language processing):

- 语 : `U+8BED` → 34726
- 言 : `U+8A00` → 35201
- 处 : `U+5904` → 22810
- 理 : `U+7406` → 29616

`text = " 语言处理 "` → `[34726, 35201, 22810, 29616]`

NLP Unicode Preprocessing: Normalization, Tokenization, Embedding

Raw Text: `"café loves NLP! 你好，世界！"`

1 Normalization (NFC)

- Standardizes Unicode representations
- Example: `"cafe"` → `"café"`

2 Tokenization

- Split text into meaningful units (words/subwords/characters)
- Example (Word-level):
`["café", "loves", "NLP", "!", 你好, "", 世界, ""]`

3 Embedding

- Convert tokens into numeric vectors
- Example (Word2Vec style):
`"café"` → [0.21, -0.05, 0.78, ...]
`" 你好 "` → [0.34, 0.65, -0.12, ...]

Step	Role	Example
Normalization	Standardize Unicode	café → café
Tokenization	Split text into tokens	"I love NLP" → ["I", "love", "NLP"]
Embedding	Convert tokens to vectors	"love" → [0.88, -0.12, 0.41,...]

Workflow Diagram (conceptual):

Raw Text ⟶ Normalization ⟶ Tokenization ⟶ Embedding

Unicode: Universal Character Encoding

One Standard for All Writing Systems

Goals:

- Any character from any language
- Historical and invented scripts
- Symbols, emojis, technical notations
- Future expansion

Numbers:

- 150,000+ characters
- 168+ writing systems
- Range: U+0000 to U+10FFFF
- 1.1 million possible code points

Examples

Latin A	U+0041	0x41
Arabic Alif	U+0627	0x627
Chinese 中	U+4E2D	0x4E2D
Emoji I love NLP 1F60A	U+1F600	0x1F600

Understanding Unicode Encodings (UTF-8, UTF-16, UTF-32)

Example string: A语

- Unicode code points:
  - A → U+0041
  - 语 → U+8BED

Encoding	Bytes per char	Representation of "A语"	Total Bytes
UTF-8	1–4 bytes	41 E8 AE BD	4
UTF-16	2–4 bytes	0041 8BED	4
UTF-32	4 bytes	00000041 00008BED	8

Key Observations:

- UTF-8 is **ASCII compatible** (English = 1 byte)
- UTF-16 balances size and simplicity
- UTF-32 is simple but memory-heavy
- **UTF-8 dominates NLP and Web applications**

Why UTF-8 Dominates NLP and Web Applications

1 **ASCII Compatibility**
- First 128 Unicode code points match ASCII
- English text = 1 byte per character
- Backward compatible with legacy systems

2 **Space-Efficient for Multilingual Text**
- 1–4 bytes per character
- ASCII = 1 byte, most other scripts = 2–4 bytes
- Ideal for English-heavy NLP datasets

3 **Universal and Standardized**
- Supports all Unicode characters: Latin, Chinese, Arabic, Hindi, emojis
- One encoding fits all languages $\rightarrow$ simpler NLP pipelines

4 **Self-Synchronizing and Robust**
- Start of character easily detectable
- Resilient to partial reads $\rightarrow$ important for NLP tokenization

5 **Web-Friendly**
- Default encoding for HTML5, HTTP, APIs, JSON
- Avoids mojibake in web pages and data exchange

Summary Table:

Feature	UTF-8 Advantage
ASCII-compatible	Compact, backward compatible
Variable-length	Efficient for mixed-language text
Universal	Supports all languages + symbols
Robust	Reliable streaming & NLP
Web standard	Default for web, JSON, APIs

UTF-8 Encoding Rules (1–4 Bytes)

UTF-8 is a variable-length encoding: 1 to 4 bytes per character.

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

Examples:

- `A` → U+0041 → 1 byte → `41`
- é → U+00E9 → 2 bytes → `C3 A9`
- 语 → U+8BED → 3 bytes → `E8 AE BD`
- Music Symbol→ U+1D11E → 4 bytes → `F0 9D 84 9E`

UTF-8 Byte Patterns

Every byte in UTF-8 follows a specific visual pattern:

Byte Type	Bit Pattern	Role
ASCII	0xxxxxxx	Single byte (English, numbers)
2-byte start	110xxxxx	Beginning of 2-byte char
3-byte start	1110xxxx	Beginning of 3-byte char
4-byte start	11110xxx	Beginning of 4-byte char
Continuation	10xxxxxx	Middle bytes of multi-byte char

Think of it like: Each byte wears a "uniform" that tells you its role!

Visual Example: Chinese Character “语”

Character: (U+8BED) → 3 bytes in UTF-8

Byte sequence:

`11101000 (start)` `10101110 (cont)` `10111101 (cont)`

Byte 1:
`11101000`
`Start marker`

Byte 2:
`10101110`
`Continuation`

Byte 3:
`10111101`
`Continuation`

Like a train: [ENGINE] [CAR] [CAR]

Analogy: The Train Metaphor

Imagine a train where each car has a colored flag:

- **BLUE flag** (0xxxxxxx) → Single ASCII car
- **RED flag** (110,1110,11110) → Engine car (start)
- **YELLOW flag** (10xxxxxx) → Passenger car (continuation)

Jumping onto a moving train:

Full train: [BLUE] [BLUE] [RED] [YELLOW] [YELLOW] [BLUE] [RED] [YELLOW] [BLUE]

H e l l o 语 W o r l d

You jump here ↓
[YELLOW] [YELLOW] [BLUE] [RED] [YELLOW] [BLUE]

You see YELLOW, walk forward until you find RED or BLUE!

Real-World Example: Streaming Text

Scenario: Receiving "Hello语World" in two network packets

Packet 1: "Hello" + first byte of 语

H	e	l	l	o	[start]
41	65	6C	6C	6F	E8

What phone sees:

- "Hello" $\rightarrow$ 5 ASCII characters (each starts with 0)
- E8 $\rightarrow$ binary 11101000 starts with "1110"
- **Decision:** "This is start of 3-byte character, need 2 more bytes"
- Phone waits patiently for Packet 2

Real-World Example: Streaming Text (Cont.)

Packet 2: Rest of 语 + "World"

[cont]	[cont]	W	o	r	l	d
AE	BD	57	6F	72	6C	64

Assembly:

1. Phone receives `AE` and `BD`
2. Both start with "10" → correct continuation bytes!
3. Assembles: `E8 AE BD` = 语
4. Then continues with "World"

Result: Perfect reconstruction despite packet boundaries!

Encoding Each Character: A

Character: A (U+0041)

- Falls in 1-byte range
- UTF-8 encoding: `41`

Encoding Character: 语 (U+8BED)

- Binary of U+8BED: 1000101111101101 (16 bits)
- UTF-8 requires 3 bytes (U+0800–U+FFFF)

Encoding Steps:

- UTF-8 3-byte template: 1110xxxx 10xxxxxx 10xxxxxx
- Fill in bits: 11101000 10101110 10111101
- Hexadecimal: `E8 AE BD`

Scenario 1: Radio Static Interference

Original broadcast: "The price is $100"

Static hits middle: "The prce is $100" (= corrupted bytes)

UTF-8 (Good)

- Lose ASCII characters only
- See: "The pr" + "ce is $100"
- Still understandable!
- No garbage inserted

UTF-16 (Bad)

- Corrupted bytes create new characters
- See: "The prˆce is $100"
- Garbage symbols appear
- Harder to understand

Chinese example: "价格是100元"
With static: "价□□是100元" → still understandable!

Scenario 2: Damaged Document

Original: “Hello 语 World”

Damaged: “Hello □ World” (□ = damaged bytes)

UTF-8 Reader

1. Sees damaged bytes
2. Checks first bit: continuation without start?
3. “Invalid! Skip to next valid start”
4. Finds space character (ASCII)
5. Continues with “World”
6. Result: “Hello World”

Bad Encoding Reader

1. Tries to interpret damaged bytes
2. Creates random symbols
3. Result: “Hello I И и World”
4. Garbage pollutes the text

Scenario 3: Search Engine Indexing

Searching for "World" in a large document:

Document: "... Hello 语 World ..."

UTF-8 byte view:

```
...  48 65 6C 6C 6F E8 AE BD 57 6F 72 6C 64 ...
```

- Search engine scans byte by byte
- Sees E8 → starts with 1110 → 3-byte character, skip 3 bytes
- Next byte: 57 → starts with 0 → ASCII 'W'
- "Found 'World' at position X!"

No need to decode everything — just follow the patterns!

UTF-8: Smart Variable-length Encoding

UTF-8 Encoding Strategy

Character Type	Bytes	Example
ASCII	1 byte	`h` (U+0068 → `0x68`)
European scripts	2 bytes	`ñ` (U+00F1 → `0xC3B1`)
Chinese/Japanese/Korean	3 bytes	中 (U+4E2D → `0xE4B8AD`)
Rare/Emoji	4 bytes	(U+1F600 → `0xF09F9880`)

Advantages:

- Backward compatible with ASCII
- Space efficient
- Self-synchronizing
- Web standard (97%+)

Example String: "`helloñ中 `"

h e l l o	$(5 \times 1\text{B} = 5\text{B})$
ñ	(2B)
中	(3B)
	(4B)
Total:	14 bytes

Evolution of Tokenization Approaches

From Words to Subwords: The NLP Journey

Early Days (1990s)
Words

- Space-based splitting
- Simple but limited
- OOV problems

Intermediate (2000s)
Characters + N-grams

- Handle OOV better
- Long sequences
- Poor semantics

Modern (2010s+)
Subwords (BPE)

- Best of both worlds
- Controlled vocabulary
- Current standard

The Right Level of Granularity

Words are too big, characters are too small,
subwords are just right for most NLP tasks

Popular Tokenization Algorithms

- Whitespace Tokenization
- Rule-Based Tokenization
- Regex Tokenization
- Treebank Tokenization
- Byte Pair Encoding (BPE)
- WordPiece
- SentencePiece
- Unigram Language Model Tokenization
- Byte-Level BPE
- Character-Level Tokenization
- Byte-Level Tokenization
- Moses Tokenizer
- Jieba Tokenizer
- spaCy Tokenizer
- NLTK Tokenizer

Popular Model - Tokenizer Mapping

Model Family	Tokenizer Used
GPT-1, GPT-2, GPT-3, GPT-4	Byte-Level BPE
BERT (base, large)	WordPiece
RoBERTa	BPE
T5, mT5	SentencePiece (Unigram)
XLNet	SentencePiece (Unigram + BPE)
ALBERT	SentencePiece (Unigram)
LLaMA, CodeLLaMA	BPE
Claude (Anthropic)	SentencePiece
Gemini (Google)	SentencePiece
Falcon	BPE
Mistral	BPE
Bloom	BPE
XLM-RoBERTa	SentencePiece
ELECTRA	WordPiece

Variants of Byte Pair Encoding (BPE)

Variant	Unit Level	Supports Multilingual	Notes
Original BPE	Byte	No	For compression
Subword BPE	Character	Limited	For NMT & NLP
Byte-Level BPE	Byte	Yes	Robust for Unicode & code
SentencePiece BPE	Unicode / char	Yes	Often combined with unigram LM
LLM Custom BPE	Byte / subword	Yes	Includes special tokens, larger vocab

Table: Comparison of BPE variants used in NLP and LLMs

Example Outputs Comparison

Tokenizer	Output for: "I love NLP! 我爱处理中文"
Whitespace	['I', 'love', 'NLP!', ' 我爱处理中文']
Treebank	['I', 'love', 'NLP', '!', ' 我爱处理中文']
Moses	['I', 'love', 'NLP', '!', ' 我爱处理中文']
Jieba	['I', 'love', 'NLP', '!', '我', '爱', '处理', '中文']
spaCy	['I', 'love', 'NLP', '!', ' 我爱处理中文']
BPE	['I', 'Ġlove', 'ĠN', 'LP', '!', 'ĠæĢģ', 'åĤĤ', 'çĥĥä', 'ä¸æĖĖ']
SentencePiece	['I', 'love', 'N', 'LP', '!', '', '我爱', '处理', '中文']
Character	['I', ' ', 'l', 'o', 'v', 'e', ' ', 'N', 'L', 'P', '!', ' ', '我', '爱', '处', '理', '中', '文']
Byte-Level	[73, 32, 108, 111, 118, 101, 32, 78, 76, 80, 33, 32, 230, 136, 145, 231, 136, 177, 229, 164, 132, 231, 144, 134, 228, 184, 173, 230, 150, 135]

Tokenization Algorithms Overview

Algorithm	Type	Vocabulary	Language Support	Use Case
Whitespace Tokenization	Rule-Based	None	All (space-separated)	Quick & dirty splitting
Rule-Based Tokenization	Rule-Based	None	Language-specific	Simple preprocessing
Regex Tokenization	Rule-Based	None	Customizable	Pattern-based splitting
Treebank Tokenizer	Rule-Based	None	English primarily	Penn Treebank projects
Moses Tokenizer	Rule-Based	None	Multilingual (30+ langs)	Machine translation
Jieba Tokenizer	Dictionary-Based	Pre-built + Custom	Chinese	Chinese NLP

Tokenization Algorithms Comparison (2)

Algorithm	Type	Vocabulary	Language Support	Use Case
spaCy Tokenizer	Rule-Based + ML	None	Multilingual (70+ models)	Production NLP
NLTK Tokenizer	Rule-Based	None	Multilingual	Education & research
Byte-Level BPE	Subword	Learned (bytes)	Any language	GPT models, Code LLMs
Character-Level	Character	Character set	Any language	Character-level models
Byte-Level	Byte	256 bytes	Any language	Raw byte processing

Whitespace Tokenization

Definition:
Whitespace Tokenization splits text into tokens based only on spaces, tabs, or newline characters.

Rule:

- Split the sentence wherever whitespace appears.

Example 1:

```
Input:  I love NLP
Output: ["I", "love", "NLP"]
```

Example 2 (with punctuation):

```
Input:  I love NLP.
Output: ["I", "love", "NLP."]
```

Advantages:

- Very simple
- Fast
- Easy to implement

Limitations:

- Does not separate punctuation
- Does not handle contractions
- Not suitable for languages without spaces

Rule-Based Tokenization

Definition:
Rule-Based Tokenization splits text using manually defined linguistic rules (e.g., punctuation handling, contractions, abbreviations).

How it works:

- Split punctuation from words
- Handle contractions (e.g., don't → do + n't)
- Apply language-specific grammar rules

Example 1:

```
Input:  I love NLP.
Output: ["I", "love", "NLP", "."]
```

Example 2 (contractions):

```
Input:  I don't like this.
Output: ["I", "do", "n't", "like", "this", "."]
```

Advantages:

- Better handling of punctuation
- Handles contractions properly
- More linguistically informed

Limitations:

- Language-dependent
- Requires handcrafted rules
- Not scalable for all languages

What is Regex Tokenization?

- Tokenization splits text into smaller units called **tokens**.
- **Regex Tokenization:** Uses **regular expressions** (patterns) to define how to split text.
- Useful when text is more complex than words separated by spaces.

Why Use Regex Tokenization?

- Handles special cases that simple space splitting cannot:
  - Emails: `adarsh@example.com`
  - URLs: `https://example.com`
  - Numbers with decimals: `95.5`
  - Special symbols: `#hashtag`, `@mention`
  - Contractions: `"don't"`, `"isn't"`
- Ensures precise and clean token extraction.

How Regex Tokenization Works (Conceptually)

1. Define a **pattern** describing what counts as a token:
   - Letters only → words
   - Apostrophes inside words → contractions
   - Numbers (integers or decimals)
   - Special symbols like #, @
2. Apply the pattern to the text.
3. Every match becomes a token.

Example 1: Simple Words Text: "I love NLP!"

- Pattern: sequences of letters
- Tokens extracted:

$$["I", "love", "NLP"]$$

- Punctuation like "!" is ignored.

Examples

Example 2: Handling Contractions **Text:** "I can't do this."

- Pattern allows apostrophes inside words
- Tokens extracted:

$$["I", "can't", "do", "this"]$$

- Keeps contractions like "can't" as a single token.

Example 3: Numbers, Hashtags, Mentions Text: "Follow me @adarsh and check NLP. Score: 95.5"

- Pattern recognizes:
  - Mentions: `@adarsh`
  - Hashtags: `#NLP`
  - Numbers: `95.5`
- Tokens extracted:

$$["Follow", "me", "@adarsh", "and", "check", "NLP", "Score", "95.5"]$$

What is Treebank Tokenization?

- Treebank Tokenization follows the **Penn Treebank conventions**.
- It is more sophisticated than whitespace or regex tokenization.
- Ensures tokens are consistent with linguistic rules for words, punctuation, and contractions.

Key Characteristics

- **Punctuation Handling:** Commas, periods, quotes, and parentheses are split into separate tokens.
- **Contraction Splitting:** ‘"don’t"‘ → ‘"do"‘ + ‘"n’t"‘, ‘"I’m"‘ → ‘"I"‘ + ‘"’m"‘.
- **Special Symbols:** Dollar signs, percentages, brackets are treated as separate tokens when appropriate.
- **Parser-Friendly:** Designed to align with syntactic parsers and treebanks.

Examples

Example 1: Punctuation Handling Text: "Hello, world!"

- Tokens:

$$["Hello", ",", "world", "!"]$$

- Commas and exclamation marks become separate tokens.

Example 2: Contractions Text: "I can't go."

- Tokens:

$$["I", "ca", "n't", "go", "."]$$

- Contractions like "can't" are split according to Penn Treebank rules.

Examples

Example 3: Complex Sentences Text: "He said, 'I'll be there at 5:30 pm.' "

- Tokens:

$$["He","said",",","``","I","'ll","be","there","at","5:30","pm",".","''"]$$

- Opening and closing quotes standardized as `` and ''.
- Times and numbers preserved as single tokens.

Advantages of Treebank Tokenization

- **Standardized:** Matches Penn Treebank conventions widely used in NLP.
- **Parser-friendly:** Works well with syntactic and semantic analysis tools.
- **Handles punctuation contractions correctly:** Reduces errors in downstream tasks.

Byte Pair Encoding (BPE): The NLP Standard

Statistical Subword Tokenization

Core Idea

Start with characters, iteratively merge most frequent pairs

Training Phase:

1. Initialize: characters as tokens
2. Count all adjacent pairs
3. Merge most frequent pair
4. Repeat until desired vocab size

Application Phase:

1. Start with character tokens
2. Apply learned merges
3. Greedy longest-match

Used by: GPT, BERT, T5, and most modern NLP models

Byte Pair Encoding (BPE) in NLP

What is BPE?

- BPE is a **subword tokenization** technique
- Starts with **characters as tokens** and iteratively merges the **most frequent pairs**
- Handles **rare words**, reduces vocabulary size, and captures morphology

Step 1: Start with characters
low: l o w </w>
lower: l o w e r </w>
newest: n e w e s t </w>
widest: w i d e s t </w>

Step 2: Count most frequent pairs and merge iteratively
After merging frequent pair `e s` → es:
newest: n e w es t </w>
widest: w i d es t </w>

Step 3: Final subword vocabulary
low, er, new, est, wid, es, t
Example Tokenization:

- "lower" → `low er`
- "newest" → `new est`
- Rare/unseen words can be tokenized into known subwords

BPE Class Exercises

Exercise 1: Given the words: `cat`, `cats`, `catch` Tokenize them into characters and perform **one BPE merge step** by combining the most frequent pair.

Exercise 2: Given the words: `play`, `played`, `player` Show how BPE can split the rare/unseen word `playing` using subwords learned from the above vocabulary.

BPE Class Exercises: Solutions

Solution 1:
Step 1: Initial character tokens with </w>
cat: c a t </w>
cats: c a t s </w>
catch: c a t c h </w>
Step 2: Count pairs - Most frequent pair: `c a` or `a t` (suppose we choose `a t`)
Step 3: Merge pair `a t` → `at`
cat: c at </w>
cats: c at s </w>
catch: c at c h </w>
Step 4: Vocabulary after merge `c, at, s, ch, h, </w>`
—
Solution 2:
Vocabulary learned: `play, ed, er`
- Rare/unseen word: `playing` - Tokenize using subwords: `play ing` → uses known subword `play` + new subword `ing`
BPE allows handling OOV words using subword units.

Why BPE Works Well

Solves Key Problems:

1. **OOV Words:**
   Can build from subwords
2. **Cross-linguistic:**
   Works for any writing system
3. **Vocabulary Control:**
   Choose desired size
4. **Morphology:**
   Captures word structure

Practical Benefits:

1. **Efficiency:**
   Balanced granularity
2. **Transfer Learning:**
   Shared subwords across languages
3. **Rare Words:**
   Better handling than word-based
4. **Standardization:**
   Widely adopted

Comparison

	Words	Characters	BPE
Vocabulary	Infinite	Small	**Controlled**
OOV Handling	Poor	Good	**Excellent**
Sequence Length	Short	Long	**Medium**
Meaning Capture	High	Low	**Good**