Tokenization

By

Tokenization

- Tokenization is the process of converting raw text into **tokens** (units) that a model can handle (typically IDs via a vocabulary lookup).

- In NLP, tokens may be words, sub-words, characters, or bytes — depending on design.

- The goal is to map text (strings) → discrete IDs → embeddings → model input.

Basic pipeline steps :

- **Normalization**: e.g., lower-casing, Unicode normalization.
- **Pre-tokenization**: e.g., splitting by whitespace/punctuation into words or “pre-tokens”.
- **Model (vocabulary / tokenization algorithm)**: mapping pre-tokens into tokens/sub-tokens.
- **Post-processing**: adding special tokens (e.g., [CLS], [SEP]), possibly handling sequence-pairs, etc.
- **Mapping to IDs**: each token gets an integer ID from the vocabulary, for feeding into model embeddings.

Basic/Word-level Tokenizers

- A “WordLevel” tokenizer simply splits text into words (based on whitespace, punctuation) and maps each full word to an ID.
- This is the simplest type of tokenization: each distinct word (token) is in the vocabulary, unseen words map to [UNK] (unknown) or some fallback.

Example: “The quick brown fox” → ["The", "quick", "brown", "fox"].

Cons:

- Vocabulary must include almost all words → large memory size, many rare tokens.
- Rare or unseen words get [UNK], losing detailed information.
- Doesn’t capture sub-word structure: e.g., "unhappiness" vs "happiness" would be separate tokens even though they share roots.
- simplest word-level tokenization has limitations (vocab size, unknown tokens)

❖ Might be okay for restricted vocabulary tasks (small domain, limited words).

Sub-Word Tokenization

- Word level Tokenization issue : if you treat every word as a token, you need a huge vocabulary and will struggle with rare/unseen words.

- Sub-word or byte-level tokenization helps handle rare words, morphological variation, and unseen vocabulary more gracefully.

- Some of the algorithms to perform Sub-Word Tokenization

  - Byte-Pair Encoding (BPE)
  - WordPiece
  - Unigram (via SentencePiece)

Byte Pair Encoding

- **Idea:** Start with characters → repeatedly merge most frequent adjacent pairs → build subword tokens.
- **Training:** Merge most frequent pairs until desired vocab size.
- **Tokenization:** Apply learned merges to new words.
- Used by a lot of Transformer models, including GPT, GPT-2, RoBERTa, BART, and DeBERTa

Byte Pair Encoding

Let’s say our corpus uses these five words:

```
("hug", 10), ("pug", 5), ("pun", 12), ("bun", 4), ("hugs", 5)
```

The base vocabulary will then be ["b", "g", "h", "n", "p", "s", "u"]

```
("h" "u" "g", 10), ("p" "u" "g", 5), ("p" "u" "n", 12), ("b" "u" "n", 4), ("h" "u" "g" "s", 5)
```

("u", "g"), which is present in "hug", "pug", and "hugs", for a grand total of 20 times in the vocabulary.

```
Vocabulary: ["b", "g", "h", "n", "p", "s", "u", "ug"]
Corpus: ("h" "ug", 10), ("p" "ug", 5), ("p" "u" "n", 12), ("b" "u" "n", 4), ("h" "ug" "s", 5)
```

Second merge rule learned is ("u", "n") -> "un". Adding that to the vocabulary and merging all existing occurrences

```
Vocabulary: ["b", "g", "h", "n", "p", "s", "u", "ug", "un"]
Corpus: ("h" "ug", 10), ("p" "ug", 5), ("p" "un", 12), ("b" "un", 4), ("h" "ug" "s", 5)
```

Now the most frequent pair is ("h", "ug"), so we learn the merge rule ("h", "ug") -> "hug",

```
Vocabulary: ["b", "g", "h", "n", "p", "s", "u", "ug", "un", "hug"]
Corpus: ("hug", 10), ("p" "ug", 5), ("p" "un", 12), ("b" "un", 4), ("hug" "s", 5)
```

we continue like this until we reach the desired vocabulary size.

Word Piece

like BPE, WordPiece learns merge rules. The main difference is the way the pair to be merged is selected.
Instead of selecting the most frequent pair, WordPiece computes a score for each pair, using the following formula:

$$\text{score} = \frac{\text{freq}(\text{pair})}{\text{freq}(\text{first}) \times \text{freq}(\text{second})}$$

By dividing the frequency of the pair by the product of the frequencies of each of its parts, the algorithm prioritizes the merging of pairs where the individual parts are less frequent in the vocabulary. **Means if the individual parts are frequent , less likely to merge**

- ❖ Frequent parts → **less** likely to merge
- ❖ Infrequent parts → more likely to merge

Google developed to pretrain BERT, Used in Transformer models based on BERT, such as DistilBERT, MobileBERT, and MPNET etc.

Word Piece

```
("hug", 10), ("pug", 5), ("pun", 12), ("bun", 4), ("hugs", 5)
```

It identifies subwords by adding a prefix (like ## for BERT) initial vocabulary will be ["b", "h", "p", "##g", "##n", "##s", "##u"]

```
("h" "##u" "##g", 10), ("p" "##u" "##g", 5), ("p" "##u" "##n", 12), ("b" "##u" "##n", 4),("h" "##u" "##g" "##s", 5)
```

The most frequent pair is ("##u", "##g") (present 20 times), but the individual frequency of "##u" is very high, so its score is not the highest (it’s 1 / 36). so the best score goes to the pair ("##g", "##s") — the only one without a "##u" — at 1 / 20, and the first merge learned is ("##g", "##s") -> ("##gs").

```
Vocabulary: ["b", "h", "p", "##g", "##n", "##s", "##u", "##gs"]
Corpus: ("h" "##u" "##g", 10), ("p" "##u" "##g", 5), ("p" "##u" "##n", 12), ("b" "##u" "##n", 4), ("h" "##u" "##gs", 5)
```

```
Vocabulary: ["b", "h", "p", "##g", "##n", "##s", "##u", "##gs", "hu"]
Corpus: ("hu" "##g", 10), ("p" "##u" "##g", 5), ("p" "##u" "##n", 12), ("b" "##u" "##n", 4), ("hu" "##gs", 5)
```

```
Vocabulary: ["b", "h", "p", "##g", "##n", "##s", "##u", "##gs", "hu", "hug"]
Corpus: ("hug", 10), ("p" "##u" "##g", 5), ("p" "##u" "##n", 12), ("b" "##u" "##n", 4), ("hu" "##gs", 5)
```

continue like this until we reach the desired vocabulary size.

Sentence Piece

It starts from a big vocabulary and removes tokens from it until it reaches the desired vocabulary size.

SentencePiece addresses the fact that not all languages use spaces to separate words. The Unigram algorithm is used in combination with Sentence Piece

Unigram algorithm computes a loss over the corpus given the current vocabulary. Then, for each symbol in the vocabulary, the algorithm computes how much the overall loss would increase if the symbol was removed.

Symbols have a lower effect on the overall loss over the corpus, so in a sense they are “less needed” and are the best candidates for removal.

Used by models like AlBERT, T5, mBART, Big Bird, and XLNet

Thank you