Advanced Methods in Text Analytics

Tokenization

UNIVERSITY OF MANNHEIM
School of Business Informatics and Mathematics

What is Tokenization?

- Often forgotten/neglected/unglamorous aspect of NLP
  - But **still present in all LMs** (from n-grams to LLMs), thus **important topic**!
  - We assume text is *nicely* "chopped up" into segments before processing it
  - Much like printers, some don't care about the details behind this process
  - They just what it to work!
- Generally, **tokenization** is the process of **splitting text into finite strings** that act as **representation of that text for computers to process it**
  - These substrings often referred to as **tokens** (Webster and Kit, 1992)
- For **example**, take the string "Tokenization won't be neglected anymore."
- If we split it based on whitespace separation (each token is underlined):
  - Tokenization won't be neglected anymore. (5 tokens)
- If we further split punctuation marks:
  - Tokenization won ' t be neglected anymore . (8 tokens)
- Using the Llama3-70B tokenizer (center dot represents whitespace):
  - Token ization ·won 't ·be ·neglected ·anymore . (8 *different* tokens)

Goal of Tokenization (1)

- Why are there different tokenization approaches?
  - Because decision on *how* to split text into tokens isn't trivial
  - Highly depends on what we want those tokens to do/represent
- **The linguistic goal**: traditionally, tokens thought of as words
  - Makes sense in some tasks, e.g. POS tagging
  - But often not clear what a word is
  - E.g. word "won't" was tokenized in three different ways in previous slide
- Hence, **in this lecture** we make **distinction between words and tokens**
  - **Token:** these finite substrings we get as output of tokenization process
  - **Word:** string that is, by itself, semantically meaningful in some language
- Still, **linguistically**, we may want **tokens to be approximations of words**
  - E.g. why break word "won't" into meaningless tokens like "won" and "'t"?
  - Many similar examples, e.g. "copy-paste" makes sense by itself
  - Models may learn the semantics of the same representations used by us
  - Producing *word-like* tokens less common today, known as pre-tokenization

Goal of Tokenization (2)

- **The systems goal:** favor system-level requirements over linguistic motivations
  - E.g. **segmenting text into higher number of tokens increases memory consumption of input sequence of a language model** (LM)
  - Slide 2: different tokenizers split same text into different number of tokens
  - So, if using fewer tokens means we can support longer input sequences at the cost of "tokens being approximations of words", then so be it.
  - E.g. use single token for "tokenization" instead of "token" and "ization"
  - **This perspective more common today**, largely motivated by the success of deep learning methods and the need to scale them
- In short, **goal of tokenization not straightforward**
  - **Linguistically**, we want tokens to be approximation of words
  - From a **systems perspective**, we are happy to drop this linguistic motivation in favor of decisions that improve our systems in any way
- In this lecture, we focus on tokenization as part of pipeline of LMs

Outline

1. Tokenization in Language Models

2. Common Tokenization Methods

UNIVERSITY OF MANNHEIM
School of Business Informatics and Mathematics

Tokenization in Language Models

Role of Tokenization in LMs

- **Tokens** are the **atomic units seen/handled by LMs**.
  - Set of tokens known to LM: **vocabulary**
  - **Each token** in vocabulary **mapped to representation vector** of size *d*
  - I.e. the typical (static) embedding table/matrix, impacts model size/costs
- **Input sequences** broken into finite sequence of known tokens
  - E.g. Token ization ·won 't ·be ·neglected ·anymore.
- **Output sequences** are composed of set of these known tokens
  - Next word distribution assigns probability to each of these tokens

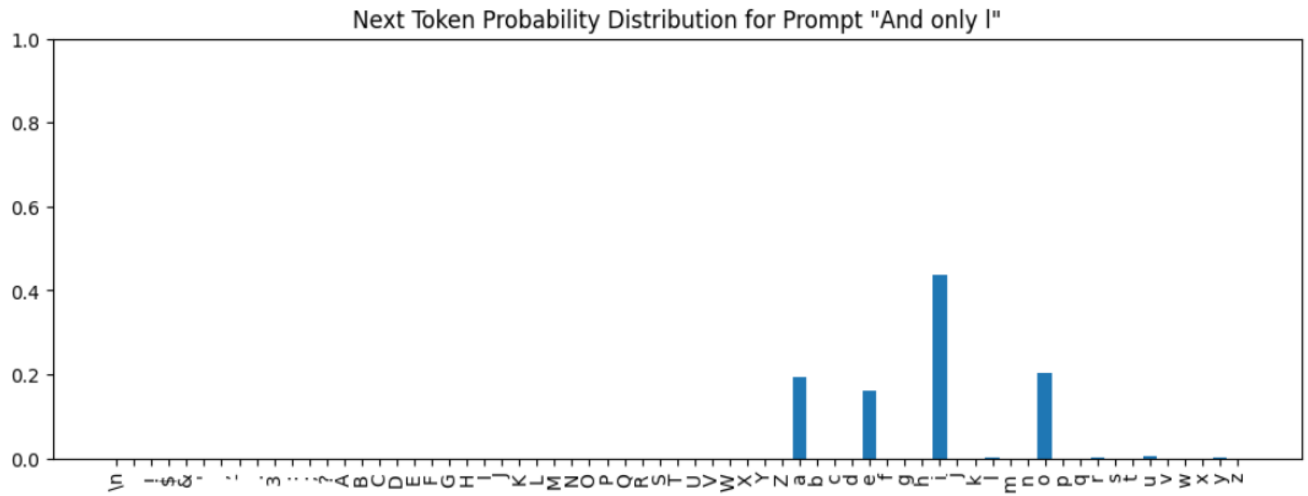
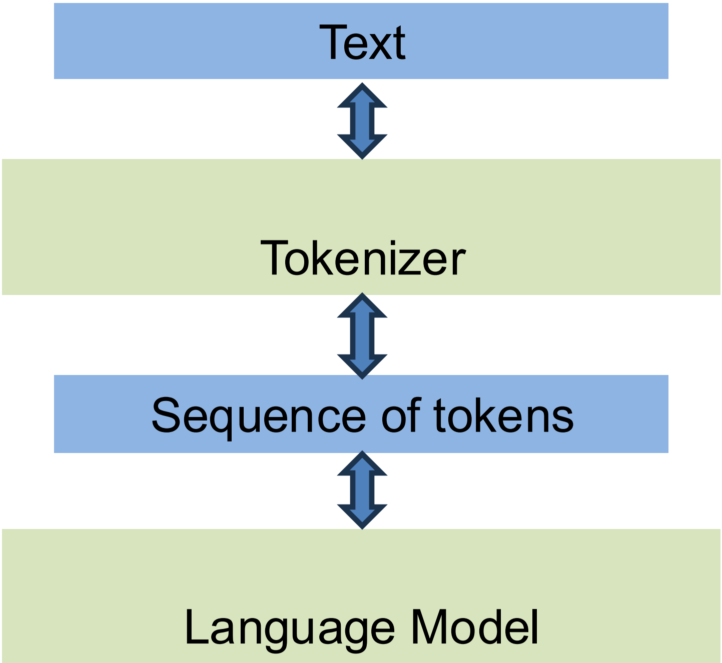


Image source

Tokenizers

- Component of NLP system, e.g. LM, in charge of tokenization process
  - *Bidirectional* mapping between text and corresponding sequence of tokens
- Important: **entire separate system from LM**
  - They can also be "trained", separately, different training data (details soon)
  - After that, they act as encoding/decoding function
- **Encoding**
  - **Input:** text
  - **Output:** corresponding sequence of tokens
- **Decoding**
  - **Input:** sequence of tokens
  - **Output:** corresponding text
- Sequence of tokens = sequence of integers
  - IDs mapped to rows of static embedding matrix, e.g. 123, 4673, 56724



Impact of Tokenization in LMs (1)

- **Tokenization** is **behind several phenomena observed when using LMs**
  - Sometimes they are a partial explanation for an observation
  - Sometimes they entirely explain an observation
- Let's look at **some examples**
  - This app can help illustrate some of these issues
  - Select tokenizer on top right, each output token has a different color
- **String processing**, e.g. reverse the string "elephant"
  - Llama3's tokenizer breaks word *elephant* into tokens "ele" and "phant"
- **Difficulty in arithmetic**, e.g. "1123 + 245 =" (answer is 1368)
  - Numbers not broken into digits
  - E.g. 1123 split into "112" and "3", 11233 into "1123" & "33"
  - Ongoing research, e.g. Singh et al. 2024
- **Difficulty spelling**, e.g. "how many letters E in the word 'elephant'?"
  - Again, model *sees* tokens "ele" and "phant"

Impact of Tokenization in LMs (2)

- **Performance in languages other than English** usually worse
  - Partly due to lower amount of LM training data in other languages
  - But partly due to lower amount of data when training tokenizer as well
- For example, number of tokens in following words from Llama3-8B:
  - Hello 1 token
  - 안녕하세요 2 tokens
  - नमस्ते 3 tokens
- **Often, words in other languages are split into more tokens**
  - This has to do with the way tokenizers are trained (details soon)
  - Impact?
  - **More memory requirement for same word in different language**
  - **More money spent using LLMs closed behind APIs** (prices per token)
  - Similar arguments behind preferences such as using YAML vs JSON as input
- Next section: we go over some popular tokenization methods.

Common Tokenization Methods

Word-Level Tokenization

UNIVERSITY OF MANNHEIM
School of Business Informatics and Mathematics

- Referred to tokenizers that **produce *word-like* tokens**
  - **Follows linguistic goal** of tokens being approximations of words
- **Today**, this process referred to as **pre-tokenization**
  - Goal of approximating words no longer a priority
  - But still an important part of pre-processing (more later)
  - May include other steps, e.g. normalization, spell correction
- **Main advantage:** interpretable!
  - Tokens are representations of words/concepts we understand.
- **Main disadvantage:** inability to deal with rare or new words
  - Rare words in training replaced with special **UNK token** (from unknown)
  - During inference, UNK used to represent out-of-vocabulary words
  - **Out-of-vocabulary words (OOV) =** words not seen during training
  - Using UNK during inference bad for: text generation, extracting useful features from OOV words, e.g. "desertification" comes from "desert"
- **Another disadvantage:** large/changing vocab. (hundreds of thousands)

Character-Level Tokenization

- Addresses **main disadvantages** of word-level tokenizers
  - If we **assume a finite set of symbols (a script)**, we **can represent OOV words as sequence of its symbols**, plus **size of vocabulary is small/finite**
- For example, if script is English alphabet, we have 26 symbols.
  - Embedding matrix: *26 x d* (plus whatever words you add to vocabulary)
  - If word "elephant" is OOV, represent it as "e" "l" "e" "p" "h" "a" "n" "t"
- **Disadvantages?**
  - In some languages, **characters may not encode meaning** (linguistic goal)
  - **Vectors per character increases length of representation** (systems goal)
  - E.g. Llama3 breaks "elephant" into "ele" and "phant", i.e. *2 x d*
  - If we break it into characters, it's *8 x d*, so more memory consumption, more costly computations, parameters may have to encode more, etc.
  - Also, **vocabulary may still be large** (e.g. Unicode has 150K characters)
- The *currently accepted* intermediate **sweet spot? Subword tokenizers!**
  - But it's ongoing research; systems change, tokenizers may accommodate

Subword-level Tokenization

- **Split *word-like* tokens into smaller units called subwords**
  - Set of all possible subwords is finite, determined from training data
  - Vocabulary also includes all characters, used to represent OOV words
- **Advantages:**
  - Vocabulary size finite, many tokens still semantically meaningful
  - Can handle OOV words well (breaks them into known subwords)
- **Disadvantage:**
  - Many ways to choose subword units, affects performance
  - E.g. manually constructed, linguistically informed rules, may favor morphologically rich languages (lots of inflections, compounding, etc.)
  - More generally, subword segmentation may not favor non-morphologically rich languages, e.g. arabic or hebrew
- **Most common approach today? Byte-Pair-Encoding (BPE)**
  - Input to BPE is output of pre-tokenization process
  - So, let's have a better look at this first

Pre-Tokenization (1)

- Can be generally described as **classic pre-processing step**
  - Classic: **rule-based**
  - Often forgotten but still present today!
  - Follows normalization step
- **Normalization**:
  - Removing unnecessary whitespaces
  - Turn all characters into lower case
  - Removing accents
  - Etc.
- **Goal of pre-tokenization:** split raw text into word-like segments
  - These segments are foundation to final tokens used by LMs
- **Common pre-tokenization approaches**:
  - Split by whitespace and punctuation (similar to what we do in tutorials)
  - Split by character (e.g. in symbol-based languages like Chinese)
  - More specific rules based on useful linguistic properties

Pre-Tokenization (2)

- Important: **output of pre-tokenization used as input to train tokenizer**
- Examples of rules [used in GPT-2]
  - The following regex pattern:

    's|'t|'re|'ve|'m|'ll|'d|

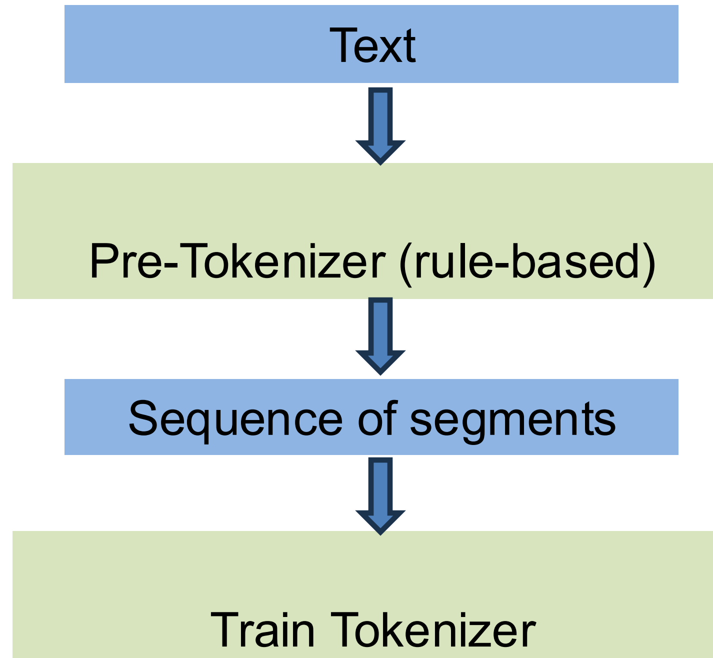
    ?\p{L}+|

    ?\p{N}+|

    ?[^\s\p{L}\p{N}]+

    |\s+(?!\S)|

    \s+



- Details beyond the scope
  - But note rules for contractions, spaces followed by letters or numbers, etc.
- **Important: rules** hand-crafted, **[language dependent]**
  - Character-based languages split by character
  - But [characters often have meaningful parts], require different rules

Byte-Pair-Encoding (1)

- Breakthrough in subword tokenization
  - Originally a data compression algorithm (Gage, 1992)
  - Recently popularized by its use in machine translation (Sennrich, 2016)
- It’s a **fast and simple heuristic**
  - Known to improve downstream performance
- **Idea:** instead of defining tokens a priori, **let data tell us what our tokens should be**
  - A priori: rules, e.g. "each word is a token", or "each character is a token"
  - If we pick the right subwords, we could handle OOV rather well!
- **BPE in short:**
  1. Starting "base vocabulary" V is set of characters
  2. Find most common two-character subword, create token for it, add to V
  3. Replace every instance of two-character subword from Step 1 with its token
  4. Keep finding common and longer subwords until k new tokens are created
- Let's look at this process in more detail.

Byte-Pair-Encoding (2)

- The algorithm runs "inside words", i.e. no merges across words
  - These "words" first determined by pre-tokenization step
  - I.e. word boundaries depend pre-tokenization rules
  - Boundary represented by added special symbol, e.g. an underscore _
- So, for a tiny input corpus of 18 word tokens, we have the following frequencies for each word, as well as our base vocabulary:

corpus		**vocabulary**
5	l o w _	_, d, e, i, l, n, o, r, s, t, w
2	l o w e s t _	
6	n e w e r _	
3	w i d e r _	
2	n e w _	

- The algorithm then counts all pairs of *adjacent* symbols
  - Most common is "er" (appears in *newer* and *wider*, so 9 times in corpus)
- It then merges the two symbols into new token, adds it to vocabulary

Byte-Pair-Encoding (3)

- We count adjacent tokens again (note "er" is now a token)

corpus		vocabulary
5	l o w _	_, d, e, i, l, n, o, r, s, t, w, er
2	l o w e s t _	
6	n e w er _	
3	w i d er _	
2	n e w _	

- Most common adjacent tokens are now "er" and "_", which we add to *V*

corpus		vocabulary
5	l o w _	_, d, e, i, l, n, o, r, s, t, w, er, er_
2	l o w e s t _	
6	n e w er_	
3	w i d er_	
2	n e w _	

- The process continues until we add *k* tokens to our base vocabulary
  - *k* is a hyperparameter,
  - This loop often referred to as **training the tokenizer**

Byte-Pair-Encoding (4)

- In the end, for *k = 8*, we would derive the following merge rules:

merge	current vocabulary
(ne, w)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new
(l, o)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo
(lo, w)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo, low
(new, er_)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo, low, newer_
(low, _)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo, low, newer_, low_

  - Note that we added two full words: *newer_* and *low_*.
  - In real-world settings, *k* is in the thousands, so many words get full tokens.
- Then, for **encoding a word into tokens**, we proceed as follows:
  - We **break input sequence into characters**
  - **Apply merges in learned order**, e.g. first merge "e" and "r" into "er"
- Note that encoding ensures we use tokens of highest possible level
  - Thus, **many words tokenized as full word**
  - **Only OOV words would be represented by subword tokens**
  - E.g. "lower" in our toy setting would be "low" and "er_"

Byte-Pair-Encoding (5)

- The algorithm for "learning" the tokenizer is thus the following.

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

- Several methods exist that are similar to BPE.
- **WordPiece** (Schuster and Nakajima, 2012) focused on Japanese and Korean (can't be relied on space-separated tokens)
  - Instead of merging most common pairs of tokens, **merged pairs that increase data likelihood of an n-gram LM** trained with this updated vocabulary

UnigramLM and SentencePiece

- **Another common approach:** UnigramLM (Kudo, 2018)
    - Adopts same idea of evaluating subword candidates by impact on LM
- **UnigramLM in short:**
    - Starts with very large vocabulary, much larger than what we would want
    - At every iteration, trains unigram LM on current vocabulary, then drops lowest probability items from vocabulary
    - Process is repeated until desired vocabulary size is reached
- Their **probabilistic approach** allowed for **interesting observations**:
    - A string can be broken down into different equally probable segmentations
    - They found that using "sampled segmentation" instead of a deterministic mapping between string and set of tokens improved performance on MT
    - Similar idea adopted by **BPE-Dropout** (Provilkov et al. 2020): token merges are randomly skipped to produce segmentation variety.
- **SentencePiece** (Kudo and Richardson, 2018):
    - Software library for BPE and UnigramLM, thus often ambiguous reference

The Tokenization Spectrum

UNIVERSITY OF MANNHEIM
School of Business Informatics and Mathematics

- **Word-level tokens**:
  - Meaningful representations, interpretable, **often helps LM performance**
  - Why? Intuition: easier for models to learn since **representation units closer to language being learned**, learned relations between representations similar to relations between words
  - Larger, potentially infinite vocabulary, difficult to handle OOV words
- **Character-level tokens**:
  - Smaller vocabulary but potentially meaningless representations, often associated with lower LM performance
  - **May require higher capacity models to learn to use representations well**
  - BPE-like processes require more merges/data (i.e. larger vocabulary, more parameters) to get to meaningful subwors/words

Character-level tokens

Word-level tokens

Handling OOV Words

- How does BPE handle OOV words?
  - Breaks them down into subwords
  - Can go as far "down" as base vocabulary
- Thus, **BPE dependent on base vocabulary to handle OOV words**
  - If we still can't identify a character in a word, we assign UNK token
- Common choices of base vocabulary:
  - Unicode (database for all symbols in all languages and beyond)
  - Raw Bytes (used by OpenAI's models, META's Llama models, etc.)
- **Unicode**:
  - **PROs:** lookup table for "all" symbols (or subsets thereof), ~150K entries
  - **CONs:** still a finite lookup table, changes over time, e.g. emojis were added
- **Raw Bytes**:
  - **PROs:** tiny base vocabulary, impossible to find OOV words
  - **CONs:** meaningless base vocabulary, likely requires stronger models
- Why impossible to find OOV? Let's discuss this in a bit more detail

Falling Back to Bytes

- **Unicode**:
  - Mapping from real-world symbol to unique ID

- **UTF-8** (stands for **U**nicode **T**ransformation **F**ormat):
  - How to encode Unicode IDs into bytes, i.e. blocks of 8-bits
  - English alphabet dominant due to US role in Computer Science history
  - Thus, such characters typically use single byte: *a-z, A-Z, 0-9,* etc.
  - Languages with other scripts added later
  - Thus, single characters from other languages often require more bytes, e.g. japanese symbol か used to make questions (i.e. common) uses 3 bytes
- So, starting with raw bytes as base vocabulary is a clever idea
  - All text boils down to byte patterns established by both Unicode and UTF-8
  - I.e. base vocabulary only has 256 tokens
  - This approach taken by GPT-2 and other OpenAI models, many LLMs since
  - Note: for fixed vocabulary size *K*, English has benefits
- We'll come back to this in future lectures

How to Select Number of Merges?

UNIVERSITY OF MANNHEIM
School of Business Informatics and Mathematics

- Do we simply choose a desired vocabulary size?
  - Usually the standard approach
  - But how to reason about this?
- More and larger subwords may lead to more memorization, less fundamental understanding (Kharitonov et al. 2021)
- Optimal choice may depend on task and language (Mielke et al. 2019)
  - Always challenging to develop general solutions
- This **choice directly relates to** where in **tokenization spectrum** we lie

Can We Get Rid of Tokenization?

- E.g. by simply using characters as tokens, i.e. we never merge anything
  - Something like Unicode encodes all symbols from past and present
  - Some language models designed around processing bytes
- But what about semantics?
  - Deep-enough transformers may be able to pick up semantics enough to outperform subword-based models (Al-Rfou et al. 2019)
  - But more work is needed to understand this better
- And what about input length?
  - Lots of characters still need longer input sequence in terms of memory/parameters/cost
  - But this may become less of an issue with recent work on infinite attention
- **Another alternative:** learn tokens based on visual representations, i.e. symbols (e.g. Rust et al. 2022)

Summary

UNIVERSITY OF MANNHEIM
School of Business Informatics and Mathematics

- **Tokenization** is often neglected, described as unglamorous
- But it has a **clear impact on the representations learned by models**
  - Thus, also a **clear impact on downstream applications**
- Several types of tokenization approaches exist
  - Word-level
  - Character-level
  - Subword-level
- Difficult to learn/advocate for one single approach
  - **Subword is most common today** (esp. **BPE** and UnigramLM)
  - But **research is ongoing to better understand impact** of different methods
- Tokenization has a **direct impact** on **evaluating model performance** and on models' ability to **perform in different languages**
  - More in future lectures

References

- Speech and Language Processing, Jurafsky et al., 2024
  - Chapter 2
- Between words and characters: A Brief History of Open-Vocabulary Modeling and Tokenization in NLP, Mielke et al, 2021
- Let's build the GPT Tokenizer, Andrej Karpathy, 2024
- References linked in corresponding slides