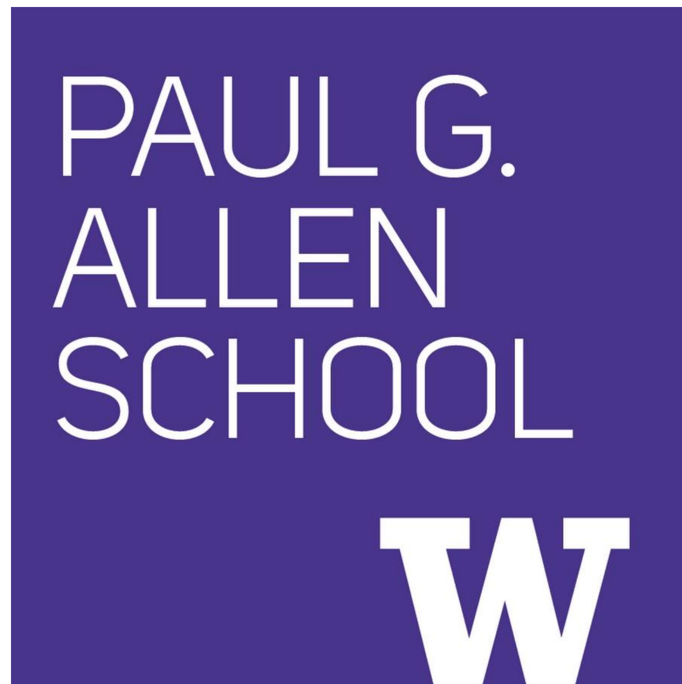


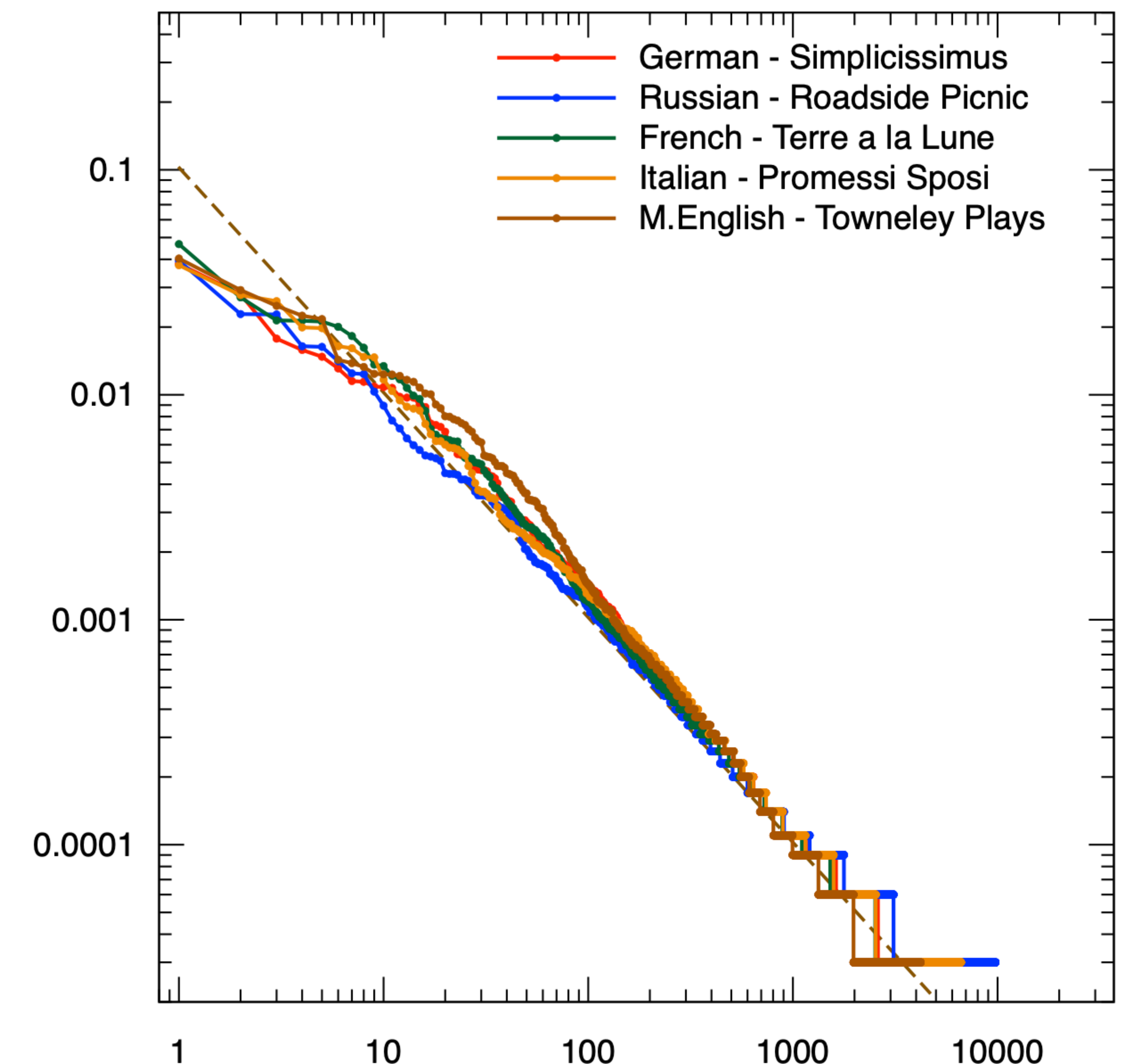
Natural Language Processing

Tokenization

Slides from Taylor Sorensen, Yejin Choi

Vocabulary - Word-Level

- For the n-gram model, our vocabulary $\mathcal{V}$ was comprised of all of the words in a language
- Some problems with this:
  - $|\mathcal{V}|$ **can be quite large** - ~470,000 words Webster's English Dictionary (3rd edition)
  - **Language is changing all of the time** - 690 words were added to Merriam Webster's in September 2023 ("rizz", "goated", "mid")
  - **Long tail of infrequent words**. Zipf's law: word frequency is inversely proportional to word rank
  - **Some words may not appear** in a training set of documents
  - **No modeled relationship between words** - e.g., "run", "ran", "runs", "runner" are all separate entries despite being linked in meaning



Zipf's Law: Word Rank vs. Word Frequency for Several Languages

$$\text{word frequency} \propto \frac{1}{\text{word rank}}$$

https://en.wikipedia.org/wiki/Zipf's_law

Character-level?

What about representing text with characters?

$$V = \{a, b, c, \ldots, z\}$$

(Maybe add capital letters, punctuation, spaces, …)

Pros:

- Small vocabulary size ($|V| = 26$ for English)
- Complete coverage (unseen words are represented by letters)

Cons:

- Encoding becomes very long - # chars instead of # words
- Poor inductive bias for learning

Subword tokenization!

How can we combine the high coverage of character-level representation with the efficiency of word-level representation?

Subword tokenization! (e.g., Byte-Pair Encoding)

- Start with character-level representations
- Build up representations from there

Original BPE Paper (Sennrich et al., 2016)
https://arxiv.org/abs/1508.07909

Byte-pair encoding - algorithm

Required:

- Documents D
- Desired vocabulary size N (greater than characters in D)

Algorithm:

- Pre-tokenize D by splitting into words (split before whitespace/punctuation)
- Initialize V as the set of characters in
- Convert D into a list of tokens (characters)
- While $| V | < N$:
  - Get counts of all bigrams in D
  - For the most frequent bigram v_i, v_j (breaking ties arbitrarily)
    - Make new token v’ by concatenating v_i and v_j
    - Change all v_i, v_j bigrams in D to v’ and add v’ to V

Byte-pair encoding - Example

Required:
- Documents D
- Desired vocabulary size N

Algorithm:
- Pre-tokenize D by splitting into words
- Initialize V as the set of characters in
- Convert D into a list of tokens (characters)
- While | V | < N :
  - Get counts of all bigrams in D
  - For the most frequent bigram v_i, v_j
    - Make new token v’ by concatenating v_i and v_j
    - Change all v_i, v_j bigrams in D to v’ and add v’ to V

$$\mathcal{D} = \{\text{“i hug pugs”}, \text{“hugging pugs is fun”}, \text{“i make puns”}\}$$

$$N = 20$$

$$\mathcal{D} = \{\text{“i”}, \text{“ hug”}, \text{“ pugs”}, \text{“hugging”}, \text{“ pugs”}, \text{“ is”}, \text{“ fun”}, \text{“i”}, \text{“ make”}, \text{“ puns”}\}$$

$$\mathcal{V} = \{\text{‘ ’}, \text{‘a’}, \text{‘e’}, \text{‘f’}, \text{‘g’}, \text{‘h’}, \text{‘i’}, \text{‘k’}, \text{‘m’}, \text{‘n’}, \text{‘p’}, \text{‘s’}, \text{‘u’}\}, |\mathcal{V}| = 13$$

$$\mathcal{D} = \{[\text{‘i’}], [\text{‘ ’}, \text{‘h’}, \text{‘u’}, \text{‘g’}], [\text{‘ ’}, \text{‘p’}, \text{‘u’}, \text{‘g’}, \text{‘s’}], [\text{‘h’}, \text{‘u’}, \text{‘g’}, \text{‘g’}, \text{‘i’}, \text{‘n’}, \text{‘g’}], [\text{‘ ’}, \text{‘p’}, \text{‘u’}, \text{‘g’}, \text{‘s’}], [\text{‘ ’}, \text{‘i’}, \text{‘s’}], [\text{‘ ’}, \text{‘f’}, \text{‘u’}, \text{‘n’}], [\text{‘i’}], [\text{‘ ’}, \text{‘m’}, \text{‘a’}, \text{‘k’}, \text{‘e’}], [\text{‘ ’}, \text{‘p’}, \text{‘u’}, \text{‘n’}, \text{‘s’}]\}$$

Example inspired by: https://huggingface.co/docs/transformers/tokenizer_summary

Byte-pair encoding - Example

Required:

- Documents D
- Desired vocabulary size N

Algorithm:

- Pre-tokenize D by splitting into words
- Initialize V as the set of characters in
- Convert D into a list of tokens (characters)
- While | V | < N :
  - Get counts of all bigrams in D
  - For the most frequent bigram v_i, v_j
    - Make new token v' by concatenating v_i and v_j
    - Change all v_i, v_j bigrams in D to v' and add v' to V

$$\mathcal{D} = \{ [\text{`i'}], [\text{` '}, \text{`h'}, \text{`u'}, \text{`g'}], [\text{` '}, \text{`p'}, \text{`u'}, \text{`g'}, \text{`s'}], [\text{`h'}, \text{`u'}, \text{`g'}, \text{`g'}, \text{`i'}, \text{`n'}, \text{`g'}], [\text{` '}, \text{`p'}, \text{`u'}, \text{`g'}, \text{`s'}], [\text{` '}, \text{`i'}, \text{`s'}], [\text{` '}, \text{`f'}, \text{`u'}, \text{`n'}], [\text{`i'}], [\text{` '}, \text{`m'}, \text{`a'}, \text{`k'}, \text{`e'}], [\text{` '}, \text{`p'}, \text{`u'}, \text{`n'}, \text{`s'}] \}$$

Bigram	Count
'u','g'	4
'p', 'u'	3
' ', 'p'	3
'h', 'u'	2
…	…

$$v_{14} := \text{concat}(\text{`u'}, \text{`g'}) = \text{`ug'}$$

Byte-pair encoding - Example

Required:

- Documents D
- Desired vocabulary size N

Algorithm:

- Pre-tokenize D by splitting into words
- Initialize V as the set of characters in
- Convert D into a list of tokens (characters)
- While | V | < N :
  - Get counts of all bigrams in D
  - For the most frequent bigram v_i, v_j
    - Make new token v' by concatenating v_i and v_j
    - Change all v_i, v_j bigrams in D to v' and add v' to V

$$\mathcal{D} = \{ [\text{`i'}], [\text{` '}, \text{`h'}, \text{`u'}, \text{`g'}], [\text{` '}, \text{`p'}, \text{`u'}, \text{`g'}, \text{`s'}], [\text{`h'}, \text{`u'}, \text{`g'}, \text{`g'}, \text{`i'}, \text{`n'}, \text{`g'}], [\text{` '}, \text{`p'}, \text{`u'}, \text{`g'}, \text{`s'}], [\text{` '}, \text{`i'}, \text{`s'}], [\text{` '}, \text{`f'}, \text{`u'}, \text{`n'}], [\text{`i'}], [\text{` '}, \text{`m'}, \text{`a'}, \text{`k'}, \text{`e'}], [\text{` '}, \text{`p'}, \text{`u'}, \text{`n'}, \text{`s'}] \}$$

$$v_{14} := \text{concat}(\text{`u'}, \text{`g'}) = \text{`ug'}$$

$$\mathcal{D} = \{ [\text{`i'}], [\text{` '}, \text{`h'}, \text{`ug'}], [\text{` '}, \text{`p'}, \text{`ug'}, \text{`s'}], [\text{`h'}, \text{`ug'}, \text{`g'}, \text{`i'}, \text{`n'}, \text{`g'}], [\text{` '}, \text{`p'}, \text{`ug'}, \text{`s'}], [\text{` '}, \text{`i'}, \text{`s'}], [\text{` '}, \text{`f'}, \text{`u'}, \text{`n'}], [\text{`i'}], [\text{` '}, \text{`m'}, \text{`a'}, \text{`k'}, \text{`e'}], [\text{` '}, \text{`p'}, \text{`u'}, \text{`n'}, \text{`s'}] \}$$

$$\mathcal{V} = \{ \text{` '}, \text{`a'}, \text{`e'}, \text{`f'}, \text{`g'}, \text{`h'}, \text{`i'}, \text{`k'}, \text{`m'}, \text{`n'}, \text{`p'}, \text{`s'}, \text{`u'}, \text{`ug'} \}, |\mathcal{V}| = 14$$

Byte-pair encoding - Example

Required:

- Documents D
- Desired vocabulary size N

Algorithm:

- Pre-tokenize D by splitting into words
- Initialize V as the set of characters in
- Convert D into a list of tokens (characters)
- While | V | < N :
  - Get counts of all bigrams in D
  - For the most frequent bigram v_i, v_j
    - Make new token v' by concatenating v_i and v_j
    - Change all v_i, v_j bigrams in D to v' and add v' to V

$$\mathcal{D} = \{ [\text{'i'}], [\text{' '}, \text{'h'}, \text{'ug'}], [\text{' '}, \text{'p'}, \text{'ug'}, \text{'s'}], [\text{'h'}, \text{'ug'}, \text{'g'}, \text{'i'}, \text{'n'}, \text{'g'}], [\text{' '}, \text{'p'}, \text{'ug'}, \text{'s'}], [\text{' '}, \text{'i'}, \text{'s'}], [\text{' '}, \text{'f'}, \text{'u'}, \text{'n'}], [\text{'i'}], [\text{' '}, \text{'m'}, \text{'a'}, \text{'k'}, \text{'e'}], [\text{' '}, \text{'p'}, \text{'u'}, \text{'n'}, \text{'s'}] \}$$

Bigram	Count
' ', 'p'	3
'p', 'ug'	2
'ug', 's'	2
'u', 'n'	2
...	...

$$v_{15} := \text{concat}(\text{' '}, \text{'p'}) = \text{' p'}$$

Byte-pair encoding - Example

Required:

- Documents D
- Desired vocabulary size N

Algorithm:

- Pre-tokenize D by splitting into words
- Initialize V as the set of characters in
- Convert D into a list of tokens (characters)
- While | V | < N :
  - Get counts of all bigrams in D
  - For the most frequent bigram v_i, v_j
    - Make new token v' by concatenating v_i and v_j
    - Change all v_i, v_j bigrams in D to v' and add v' to V

$$\mathcal{D} = \{ [\text{'i'}], [\text{' '}, \text{'h'}, \text{'ug'}], [\text{' '}, \text{'p'}, \text{'ug'}, \text{'s'}], [\text{'h'}, \text{'ug'}, \text{'g'}, \text{'i'}, \text{'n'}, \text{'g'}], [\text{' '}, \text{'p'}, \text{'ug'}, \text{'s'}], [\text{' '}, \text{'i'}, \text{'s'}], [\text{' '}, \text{'f'}, \text{'u'}, \text{'n'}], [\text{'i'}], [\text{' '}, \text{'m'}, \text{'a'}, \text{'k'}, \text{'e'}], [\text{' '}, \text{'p'}, \text{'u'}, \text{'n'}, \text{'s'}] \}$$

$$v_{15} := \text{concat}(\text{' '}, \text{'p'}) = \text{' p'}$$

$$\mathcal{D} = \{ [\text{'i'}], [\text{' '}, \text{'h'}, \text{'ug'}], [\text{' p'}, \text{'ug'}, \text{'s'}], [\text{'h'}, \text{'ug'}, \text{'g'}, \text{'i'}, \text{'n'}, \text{'g'}], [\text{' p'}, \text{'ug'}, \text{'s'}], [\text{' '}, \text{'i'}, \text{'s'}], [\text{' '}, \text{'f'}, \text{'u'}, \text{'n'}], [\text{'i'}], [\text{' '}, \text{'m'}, \text{'a'}, \text{'k'}, \text{'e'}], [\text{' p'}, \text{'u'}, \text{'n'}, \text{'s'}] \}$$

$$\mathcal{V} = \{ \text{' '}, \text{'a'}, \text{'e'}, \text{'f'}, \text{'g'}, \text{'h'}, \text{'i'}, \text{'k'}, \text{'m'}, \text{'n'}, \text{'p'}, \text{'s'}, \text{'u'}, \text{'ug'}, \text{' p'} \}, |\mathcal{V}| = 15$$

Byte-pair encoding - Example

Required:

- Documents D
- Desired vocabulary size N

Algorithm:

- Pre-tokenize D by splitting into words
- Initialize V as the set of characters in
- Convert D into a list of tokens (characters)
- While | V | < N :
  - Get counts of all bigrams in D
  - For the most frequent bigram v_i, v_j
    - Make new token v' by concatenating v_i and v_j
    - Change all v_i, v_j bigramand add v' to s in D to v' V

Repeat until …

$$\mathcal{V} = \{\text{` ', `a', `e', `f', `g', `h', `i', `k', `m',`n', `p', `s', `u',}$$
$$\text{`ug', ` p', `hug', ` pug', ` pugs', `un', ` hug'}\},$$

$$|\mathcal{V}| = 20$$

$$\mathcal{D} = \{\, [\text{`i'}]\,, [\text{` hug'}]\,, [\text{` pugs'}]\,,$$
$$[\text{`hug', `g', `i', `n', `g'}]\,, [\text{` pugs'}]\,,$$
$$[\text{` ', `i', `s'}]\,, [\text{` ', `f', `un'}]\,, [\text{`i'}]\,,$$
$$[\text{` ', `m', `a', `k', `e'}]\,, [\text{` p', `un', `s'}]\}$$

CHANGES FROM START

Byte-pair encoding - Example

CHANGES FROM START

Questions:

- Is every token we made used in the corpus? Why or why not?
- How much memory (#tokens) have we saved for each document?
- What would happen if you kept adding vocabulary until you couldn't anymore?

$$\mathcal{V} = \{\text{‘ ’, ‘a’, ‘e’, ‘f’, ‘g’, ‘h’, ‘i’, ‘k’, ‘m’, ‘n’, ‘p’, ‘s’, ‘u’,}$$
$$\text{‘ug’, ‘ p’, ‘hug’, ‘ pug’, ‘ pugs’, ‘un’, ‘ hug’}\},$$
$$|\mathcal{V}| = 20$$

$$\mathcal{D} = \{\text{[‘i’], [‘ hug’], [‘ pugs’],}$$
$$\text{[‘hug’, ‘g’, ‘i’, ‘n’, ‘g’], [‘ pugs’],}$$
$$\text{[‘ ’, ‘i’, ‘s’], [‘ ’, ‘f’, ‘un’], [‘i’],}$$
$$\text{[‘ ’, ‘m’, ‘a’, ‘k’, ‘e’], [‘ p’, ‘un’, ‘s’]}\}$$

Byte-pair encoding - Tokenization/Encoding

With this vocabulary, can you represent (or, tokenize/encode):

- “apple”?
  - No, there is no ‘l’ in the vocabulary
- “huge”?
  - Yes - [16, 4]
- “ huge”?
  - Yes - [18, 4]
- “ hugest”?
  - No, there is no ’t’ in the vocabulary
- “unassumingness”?
  - Yes - [19, 2, 12, 12, 13, 9, 7, 10, 5, 10, 3, 12, 12]

$$\mathcal{V} = \{1: \text{` '}, 2: \text{`a'}, 3: \text{`e'}, 4: \text{`f'}, 5: \text{`g'}, 6: \text{`h'}, 7: \text{`i'}, 8: \text{`k'}, 9: \text{`m'}, 10: \text{`n'}, 11: \text{`p'}, 12: \text{`s'}, 13: \text{`u'}, 14: \text{`ug'}, 15: \text{` p'}, 16: \text{`hug'}, 17: \text{` pug'}, 18: \text{` pugs'}, 19: \text{`un'}, 20: \text{` hug'}\}$$

Byte-pair encoding - Tokenization/Encoding

$$\mathcal{V} = \{1 : \text{` '}, 2 : \text{`a'}, 3 : \text{`e'}, 4 : \text{`f'}, 5 : \text{`g'}, 6 : \text{`h'}, 7 : \text{`i'},$$
$$8 : \text{`k'}, 9 : \text{`m'}, 10 : \text{`n'}, 11 : \text{`p'}, 12 : \text{`s'}, 13 : \text{`u'},$$
$$14 : \text{`ug'}, 15 : \text{` p'}, 16 : \text{`hug'}, 17 : \text{` pug'}, 18 : \text{` pugs'},$$
$$19 : \text{`un'}, 20 : \text{` hug'}\}$$

- Sometimes, there may be more than one way to represent a word with the vocabulary…
  - E.g., “ hugs” = [20, 12] = [1, 16, 12] = [1, 6, 14, 12] = [1, 6, 13, 5, 13]
    - Which is the best representation? Why?

Byte-pair encoding - Tokenization/Encoding

$$\mathcal{V} = \{1: \text{` '}, 2: \text{`a'}, 3: \text{`e'}, 4: \text{`f'}, 5: \text{`g'}, 6: \text{`h'}, 7: \text{`i'}, \\ 8: \text{`k'}, 9: \text{`m'}, 10: \text{`n'}, 11: \text{`p'}, 12: \text{`s'}, 13: \text{`u'}, \\ 14: \text{`ug'}, 15: \text{` p'}, 16: \text{`hug'}, 17: \text{` pug'}, 18: \text{` pugs'}, \\ 19: \text{`un'}, 20: \text{` hug'}\}$$

$$\text{Encode}(\text{`` hugs''}) = [20, 12]$$

$$\text{Encode}(\text{``misshapenness''}) = [9, 7, 12, 12, 6, 2, \\ 11, 3, 10, 10, 3, 12, 12]$$

Encoding Algorithm

Given string and (ordered) vocab ,

- Pretokenize in same way as before
- Tokenize into characters
- Perform merge rules in same order as in training until no more merges may be done

Byte-pair encoding - Decoding

$$\mathcal{V} = \{1: \text{` '}, 2: \text{`a'}, 3: \text{`e'}, 4: \text{`f'}, 5: \text{`g'}, 6: \text{`h'}, 7: \text{`i'}, \\ 8: \text{`k'}, 9: \text{`m'}, 10: \text{`n'}, 11: \text{`p'}, 12: \text{`s'}, 13: \text{`u'}, \\ 14: \text{`ug'}, 15: \text{` p'}, 16: \text{`hug'}, 17: \text{` pug'}, 18: \text{` pugs'}, \\ 19: \text{`un'}, 20: \text{` hug'}\}$$

Decoding Algorithm

Given list of tokens:

- Initialize string
- Keep popping off tokens from the front and appending to the string to

$$\text{Decode}([20, 12]) = \text{`` hugs''}$$

$$\text{Decode}([9, 7, 12, 12, 6, 2, 11, 3, 10, 10, 3, 12, 12]) \\ = \text{``misshapenness''}$$

Byte-pair encoding - Properties

- Efficient to run (greedy vs. global optimization)
- Lossless compression
- Potentially some shared representations - e.g., the token “hug” could be used both in “hug” and “hugging”

Byte-pair encoding - Usage

- Basically state of the art in tokenization
- Used in all modern left-to-right large language models (LLMs), including ChatGPT

Model/Tokenizer	Vocabulary Size
GPT-3.5/GPT-4/ChatGPT	100k
GPT-2/GPT-3	50k
Llama2	32k
Falcon	65k

Byte-pair encoding - ChatGPT Example

Moby Dick as tokenized by ChatGPT

Call me Ishmael. Some years ago—never mind how long precisely—having
little or no money in my purse, and nothing particular to interest me on
shore, I thought I would sail about a little and see the watery part of
the world. It is a way I have of driving off the spleen and regulating
the circulation. Whenever I find myself growing grim about the mouth;
whenever it is a damp, drizzly November in my soul; whenever I find
myself involuntarily pausing before coffin warehouses, and bringing up
the rear of every funeral I meet; and especially whenever my hypos get
such an upper hand of me, that it requires a strong moral principle to
prevent me from deliberately stepping into the street, and methodically
knocking people's hats off—then, I account it high time tozz get to sea
as soon as I can. This is my substitute for pistol and ball. With a
philosophical flourish Cato throws himself upon his sword; I quietly
take to the ship. There is nothing surprising in this. If they but knew
it, almost all men in their degree, some time or other, cherish very
nearly the same feelings towards the ocean with me.

TEXT TOKEN IDS

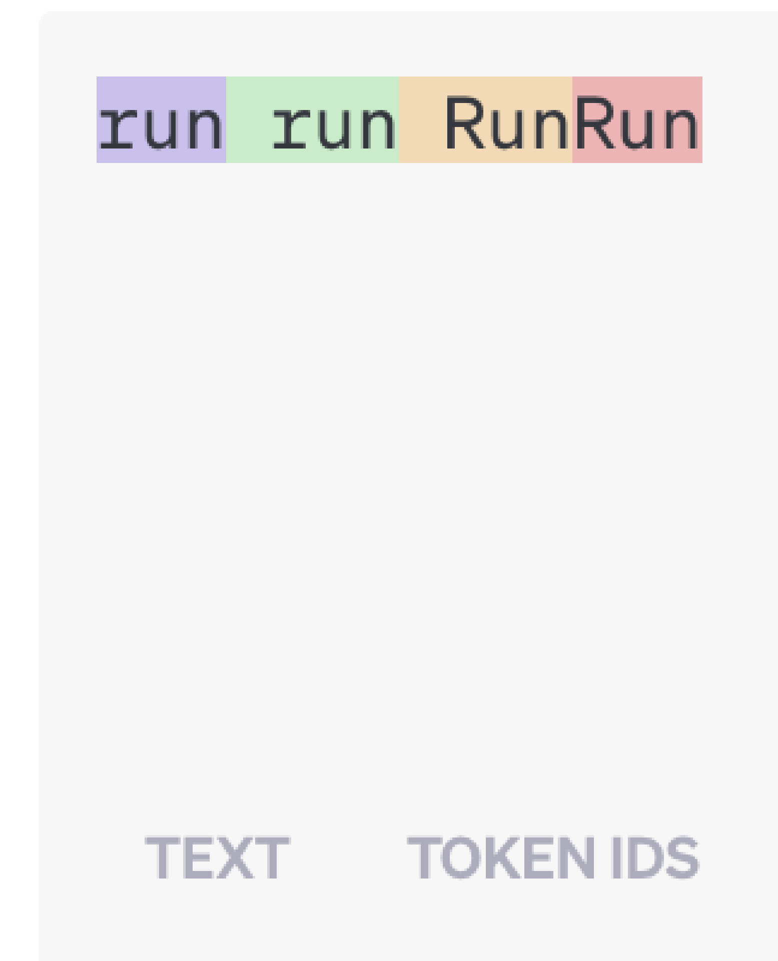
Tokens	Characters
239	1109

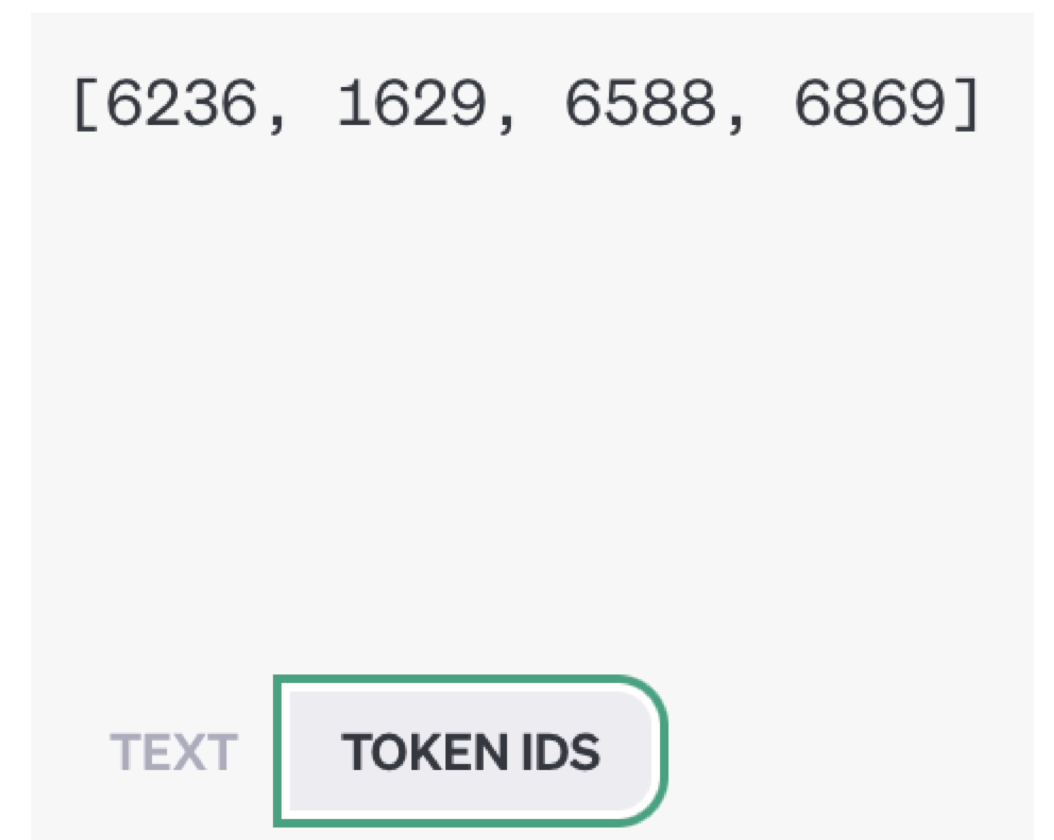
[7368, 757, 57704, 1764, 301, 13, 4427, 1667, 4227, 2345, 37593, 4059,
1268, 1317, 24559, 2345, 69666, 2697, 477, 912, 3300, 304, 856, 53101,
11, 323, 4400, 4040, 311, 2802, 757, 389, 31284, 11, 358, 3463, 358,
1053, 30503, 922, 264, 2697, 323, 1518, 279, 30125, 727, 961, 315, 279,
1917, 13, 1102, 374, 264, 1648, 358, 617, 315, 10043, 1022, 279, 87450,
268, 323, 58499, 279, 35855, 13, 43633, 358, 1505, 7182, 7982, 44517,
922, 279, 11013, 26, 15716, 433, 374, 264, 41369, 11, 1377, 73825, 6841,
304, 856, 13836, 26, 15716, 358, 1505, 7182, 4457, 3935, 6751, 7251, 985,
1603, 78766, 83273, 11, 323, 12967, 709, 279, 14981, 315, 1475, 32079,
358, 3449, 26, 323, 5423, 15716, 856, 6409, 981, 636, 1778, 459, 8582,
1450, 315, 757, 11, 430, 433, 7612, 264, 3831, 16033, 17966, 311, 5471,
757, 505, 36192, 36567, 1139, 279, 8761, 11, 323, 1749, 2740, 50244,
1274, 753, 45526, 1022, 2345, 3473, 11, 358, 2759, 433, 1579, 892, 311,
10616, 636, 311, 9581, 439, 5246, 439, 358, 649, 13, 1115, 374, 856,
28779, 369, 40536, 323, 5041, 13, 3161, 264, 41903, 67784, 356, 4428,
3872, 5678, 5304, 813, 20827, 26, 358, 30666, 1935, 311, 279, 8448, 13,
2684, 374, 4400, 15206, 304, 420, 13, 1442, 814, 719, 7020, 433, 11,
4661, 682, 3026, 304, 872, 8547, 11, 1063, 892, 477, 1023, 11, 87785,
[illegible], [illegible], [illegible], 1890, 16024, 7119, 279, 18435, 449, 757, 13]

TEXT TOKEN IDS

Weird properties of tokenizers

- Token != word
- Spaces are part of token
  - “run” is a different token than “ run”
- Not invariant to case changes
  - “Run” is a different token than “run”





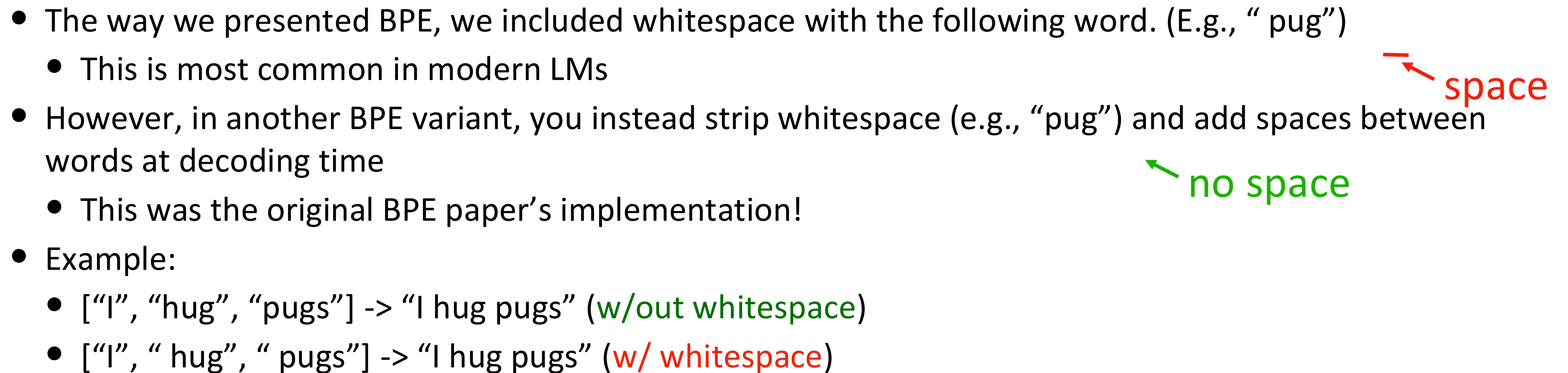
Weird properties of tokenizers

- Token != word
- Spaces are part of token
  - “run” is a different token than “ run”
- Not invariant to case changes
  - “Run” is a different token than “run”
- Tokenization fits statistics of your data
  - e.g., while these words are multiple tokens…
  - These words are all 1 token in GPT-3’s tokenizer!
  - *Why?*
    - Reddit usernames and certain code attributes appeared enough in the corpus to surface as its own token!

Example from https://www.lesswrong.com/posts/aPeJE8bSo6rAFoLqg/solidgoldmagikarp-plus-prompt-generation

Other Tokenization Variants

Variants - No spaces in tokens



- The way we presented BPE, we included whitespace with the following word. (E.g., “ pug”)
  - This is most common in modern LMs
- However, in another BPE variant, you instead strip whitespace (e.g., “pug”) and add spaces between words at decoding time
  - This was the original BPE paper’s implementation!
- Example:
  - [“I”, “hug”, “pugs”] -> “I hug pugs” (w/out whitespace)
  - [“I”, “ hug”, “ pugs”] -> “I hug pugs” (w/ whitespace)

Original (w/ whitespace)

Required:
- Documents
- Desired vocabulary size (greater than chars in)

Algorithm:

- Pre-tokenize by splitting into words (**split before** whitespace/punctuation)
- Initialize as the set of characters in

Updated (w/out whitespace)

Required:
- Documents
- Desired vocabulary size (greater than chars in)

Algorithm:

+ Pre-tokenize by splitting into words (**removing** whitespace)
- Initialize as the set of characters in

Variants - No spaces in tokens

- For sub-word tokens, need to add “continue word” special character
  - E.g., for the word “Tokenization”, if the subword tokens are “Token” and “ization”,
    - W/out special character: [“Token”, “ization”] -> “Token ization”
    - W/ special character #: [“Token”, “#ization”] -> Tokenization”
  - When decoding, if does not have special character add a space
- Example:
  - [“I”, “li”, “#ke”, “to”, “hug”, “pug”, “#s”] -> “I like to hug pugs”

Variants - No spaces in tokens

- Loses some whitespace information (lossy compression!)
  - E.g., Tokenize(“I eat cake.”) == Tokenize(“ I   eat cake   .”)
  - Especially problematic for code (e.g., Python) - why?

```
tokenizer = AutoTokenizer.from_pretrained("openai-gpt")
tokens = tokenizer.encode("i eat cake.")
print(tokens)
print(tokenizer.decode(tokens))

tokens = tokenizer.encode(" i      eat   cake    .")
print(tokens)
print(tokenizer.decode(tokens))
```

✓ 0.4s

```
[249, 2425, 5409, 239]
i eat cake.
[249, 2425, 5409, 239]
i eat cake.
```

(Example using GPT’s tokenizer, which does not include spaces in the token)

Variants - No Pre-tokenization

- In the variant we proposed, we start by splitting into words
  - This guarantees that each token will be no longer than one word
  - However, this does not work so well for character-based languages. *Why?*

Variants - No Pre-tokenization

- Instead, we could *not* pre-tokenize, and treat the entire document or sentence as a single list of tokens
  - Allows for tokens to span multiple words/characters
- Sometimes called SentencePiece tokenization* (Kudo, 2018)

* (not to be confused with the SentencePiece library, which is an implementation of *many* kinds of tokenization)

Paper: https://arxiv.org/abs/1808.06226
Library: https://github.com/google/sentencepiece

Original (w/ pre-tokenization)

Required:

- Documents
- Desired vocabulary size (greater than chars in)

Algorithm:

- Pre-tokenize by splitting into words (split before whitespace/punctuation)

- Initialize as the set of characters in

Updated (w/out pre-tokenization)

Required:

- Documents
- Desired vocabulary size (greater than chars in)

Algorithm:

+ Do not pre-tokenize

- Initialize as the set of characters in

Variants - No Pre-tokenization

- Allows sequences of words/characters to become tokens

SentencePiece paper example in Japanese:

https://arxiv.org/pdf/1808.06226.pdf

- **Raw text:** [こんにちは世界。] (*Hello world.*)
- **Tokenized:** [こんにちは] [世界] [。]

Jurassic-1 model example in English:

https://uploads-ssl.webflow.com/60fd4503684b466578c0d307/61138924626a6981ee09caf6_jurassic_tech_paper.pdf

Q: What is the most successful film to date?

A: The most successful film to date is "**The Lord of the Rings: The Fellowship of the Ring**".

Lord of the Rings	%8.47
Matrix	%7.65
Avengers	%5.86

Variants - Byte-based

- Originally, we presented BPE as dealing with characters as the smallest unit
  - However, there are *many* characters - especially if you want to support:
    - character-based languages (e.g., Chinese has >100k characters!)
    - non-alphanumeric characters like emojis (Unicode 15 has ~150k characters!)
- Instead, can initialize tokens as set of bytes! (e.g., with UTF-8*)

*Only 256 bytes! Each Unicode char is 1-4 bytes

Original (w/ characters)

Required:

- Documents
- Desired vocabulary size (greater than chars in)

Algorithm:

- Pre-tokenize by splitting into words (split before whitespace/punctuation)

- Initialize as the set of **characters** in

- Convert into a list of tokens (**characters**)

- While < :

Modified (w/ bytes)

Required:

- Documents
- Desired vocabulary size (greater than chars in)

Algorithm:

- Pre-tokenize by splitting into words (split before whitespace/punctuation)

+ Initialize as the set of **bytes** in

+ Convert into a list of tokens (**bytes**)

- While < :
  - Let

Variants - Byte-based

Instead, can initialize tokens as set of bytes! (e.g., with UTF-8)

```
print('apple'.encode('utf-8'))
print('😂'.encode('utf-8'))
print('こんにちは'.encode('utf-8'))
✓ 0.0s

b'apple'
b'\xf0\x9f\x98\x82'
b'\xe3\x81\x93\xe3\x82\x93\xe3\x81\xab\xe3\x81\xa1\xe3\x81\xaf'
```

UTF-8 Byte Encoding in Python

Variants - Byte-based

While character-based GPT tokenizer fails on emojis and Japanese…

```python
gpt_tokenizer = AutoTokenizer.from_pret
tokens = gpt_tokenizer.encode('😂')
print(tokens)
print(gpt_tokenizer.decode(tokens))
tokens = gpt_tokenizer.encode('こんにちは
print(tokens)
print(gpt_tokenizer.decode(tokens))
```

✓ 0.7s

```
[0]
<unk>
[0, 0, 0, 0, 0]
<unk><unk><unk><unk><unk>
```

The Byte-based GPT-2 tokenizer succeeds!

```python
gpt2_tokenizer = AutoTokenizer.from_pretrained("gpt2")
tokens = gpt2_tokenizer.encode('😂')
print(tokens)
print(gpt2_tokenizer.decode(tokens))
tokens = gpt2_tokenizer.encode('こんにちは')
print(tokens)
print(gpt2_tokenizer.decode(tokens))
```

✓ 0.5s

```
[47249, 224]
😂
[46036, 22174, 28618, 2515, 94, 31676]
こんにちは
```

Variants - WordPiece Objective

- BPE: the bigram with highest frequency/highest probability $p(v_i, v_j)$
- WordPiece: bigram which maximizes the likelihood of the data after the merge is made $\frac{p(v_i, v_j)}{p(v_i)p(v_j)}$
  - Maximizes the probability of the bigram, normalized by the probability of the unigrams
  - *What does it mean if* $\frac{p(v_i, v_j)}{p(v_i)p(v_j)}$ *is close to 1?*
    - Whenever the individual tokens appear, the bigram almost always appears
  - *What does it mean if* $p(v_i, v_j)$ *is high but* $\frac{p(v_i, v_j)}{p(v_i)p(v_j)}$ *is low?*
    - The tokens appear many other times (not in the bigram) in the corpus

Variants - Unigram Objective

- BPE starts with a small vocabulary (characters) and builds up until the desired vocabulary size
- The Unigram tokenization algorithm starts with a large vocabulary (all sub-word substrings) and throws away tokens until we reach size

Variants - Unigram Objective (High-level Algorithm)

- Initialize vocabulary V with all sub-word substrings of D
- Repeat until vocabulary is of size N
  - For each token v,
    1. Estimate a Unigram model based on vocab V with v removed.
    2. Calculate the probability of each word in D on the best possible tokenization (tokenization with highest probability under unigram model)
       - Can calculate this efficiently with Viterbi algorithm/Dynamic Programming
    3. Calculate the likelihood of D under the unigram model. (Likelihood after removing the token v)
  - Remove p% (where p is hyper parameter) of the tokens for which the likelihood of the data is highest after removal (e.g., the tokens which least impact loss)

For more details and a worked example, see:

https://huggingface.co/learn/nlp-course/chapter6/7?fw=pt

Examples of Models and their Tokenizers

Model/Tokenizer	Objective	Spaces part of token?	Pre-tokenization	Smallest unit
GPT	BPE	No	Yes	Character-level
GPT-2/3/4, ChatGPT, Llama(2), Falcon, …	BPE	Yes	Yes	Byte-level
Jurassic	BPE	Yes	No. “SentencePiece” - treat whitespace like char	Byte-level
Bert, DistilBert, Electra	WordPiece	No	Yes	Character-level
T5, ALBERT, XLNet, Marian	Unigram	Yes	No. “SentencePiece” - treat whitespace like char*	Character-level

*For non-English languages

Tokenizer-free modeling

- ByT5 (Xue, 2021) converts text to bytes (e.g., UTF-8 encoding) and directly predicts bytes, treating each byte as a "token"
  - Performs fairly well, especially at small model sizes! But, byte sequences are longer than BPE-based tokenized sequences https://arxiv.org/pdf/2105.13626.pdf
- Many other variants, small but active area of research

Thank you!