Words, Morphemes, and Characters

Aaron Mueller
CAS CS 505: Introduction to Natural Language Processing
Spring 2026
Boston University

Overview of Concepts

Words are text units separated by whitespaces.

Morphemes are units of meaning that compose into words.

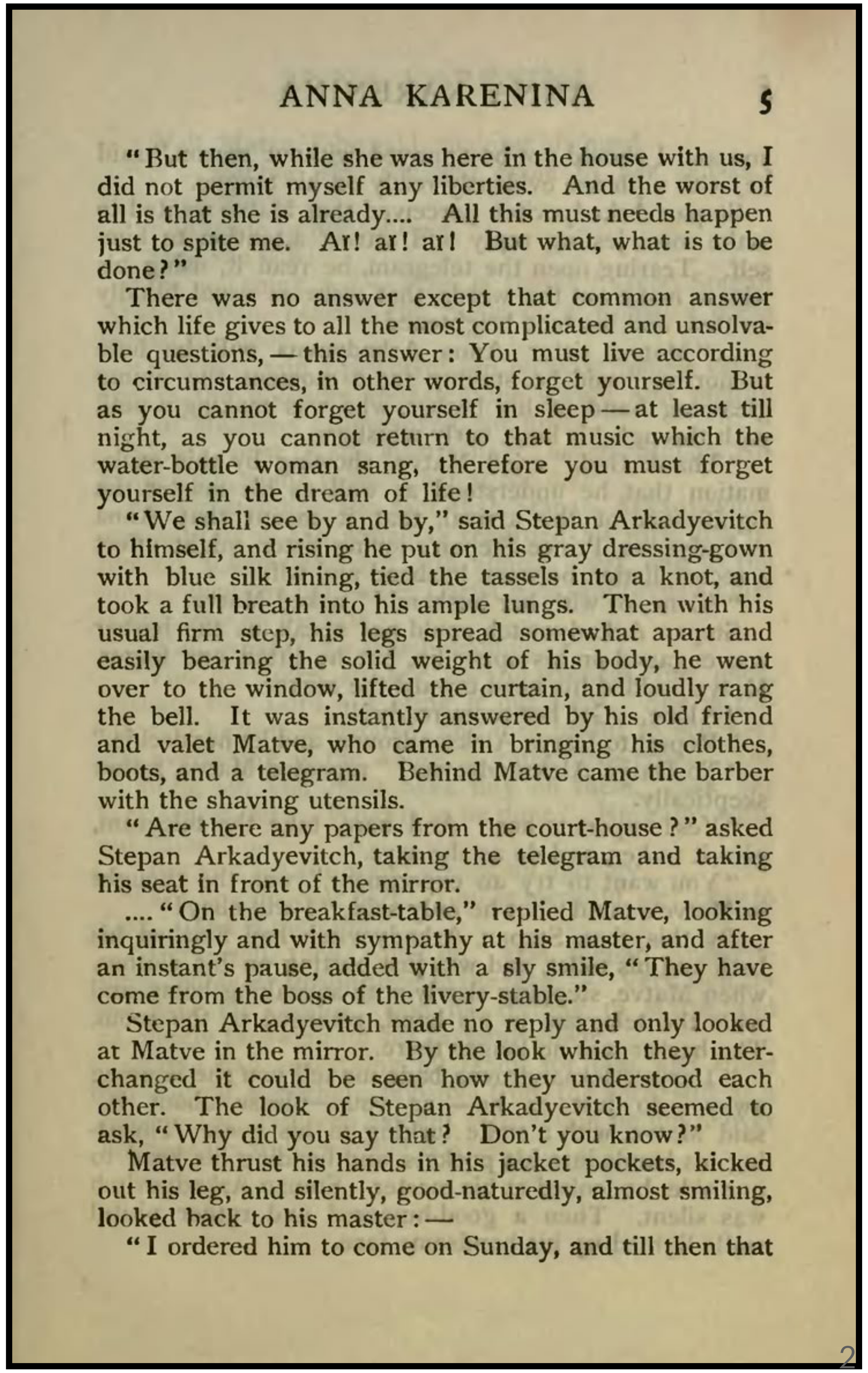
Tokens are units of input to a language model, usually (but not always) composed of units smaller than words.

Documents are collections of sentences.

Corpora are collections of documents. (Singular: *corpus*.)

Language models are systems that assign probabilities to arbitrary sequences of tokens.

Regular expressions are important tools for string matching and preprocessing.



ANNA KARENINA 5

"But then, while she was here in the house with us, I did not permit myself any liberties. And the worst of all is that she is already.... All this must needs happen just to spite me. Ai! ai! ai! But what, what is to be done?"

There was no answer except that common answer which life gives to all the most complicated and unsolvable questions, — this answer: You must live according to circumstances, in other words, forget yourself. But as you cannot forget yourself in sleep — at least till night, as you cannot return to that music which the water-bottle woman sang, therefore you must forget yourself in the dream of life!

"We shall see by and by," said Stepan Arkadyevitch to himself, and rising he put on his gray dressing-gown with blue silk lining, tied the tassels into a knot, and took a full breath into his ample lungs. Then with his usual firm step, his legs spread somewhat apart and easily bearing the solid weight of his body, he went over to the window, lifted the curtain, and loudly rang the bell. It was instantly answered by his old friend and valet Matve, who came in bringing his clothes, boots, and a telegram. Behind Matve came the barber with the shaving utensils.

"Are there any papers from the court-house?" asked Stepan Arkadyevitch, taking the telegram and taking his seat in front of the mirror.

.... "On the breakfast-table," replied Matve, looking inquiringly and with sympathy at his master, and after an instant's pause, added with a sly smile, "They have come from the boss of the livery-stable."

Stepan Arkadyevitch made no reply and only looked at Matve in the mirror. By the look which they interchanged it could be seen how they understood each other. The look of Stepan Arkadyevitch seemed to ask, "Why did you say that? Don't you know?"

Matve thrust his hands in his jacket pockets, kicked out his leg, and silently, good-naturedly, almost smiling, looked back to his master: —

"I ordered him to come on Sunday, and till then that

What can you learn from context?

Boston University is in ________. **[Factual knowledge]**

Cats like to eat ___________. **[Factual knowledge]**

Where are _________ napkins? **[Parts of speech, sentence structure]**

15 x 5 = ________ **[Arithmetic]**

The keys to the cabinet ______ on the table. **[Subject-verb agreement]**

You could easily fill in these blanks with plausible values. How could we get computers to do this?

How might we compute the similarity of two sentences?

How would we represent the meaning of a word or sentence in a computer?

A **language model** is a system that produces probabilities over sequences of **tokens**:

$$p(w_1, w_2, \ldots, w_n)$$

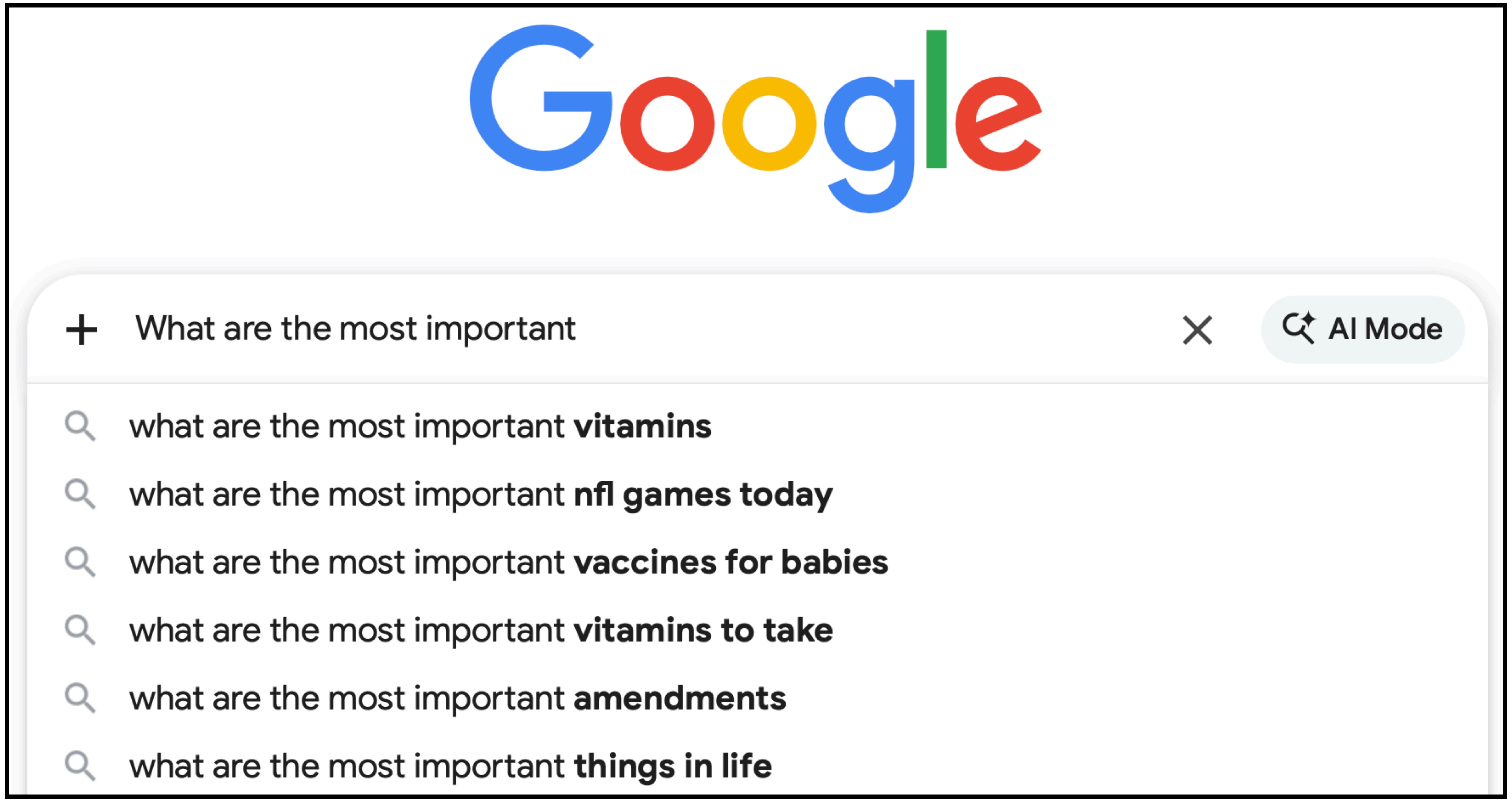
The number of possible token sequences is *infinite*. How could we model this?

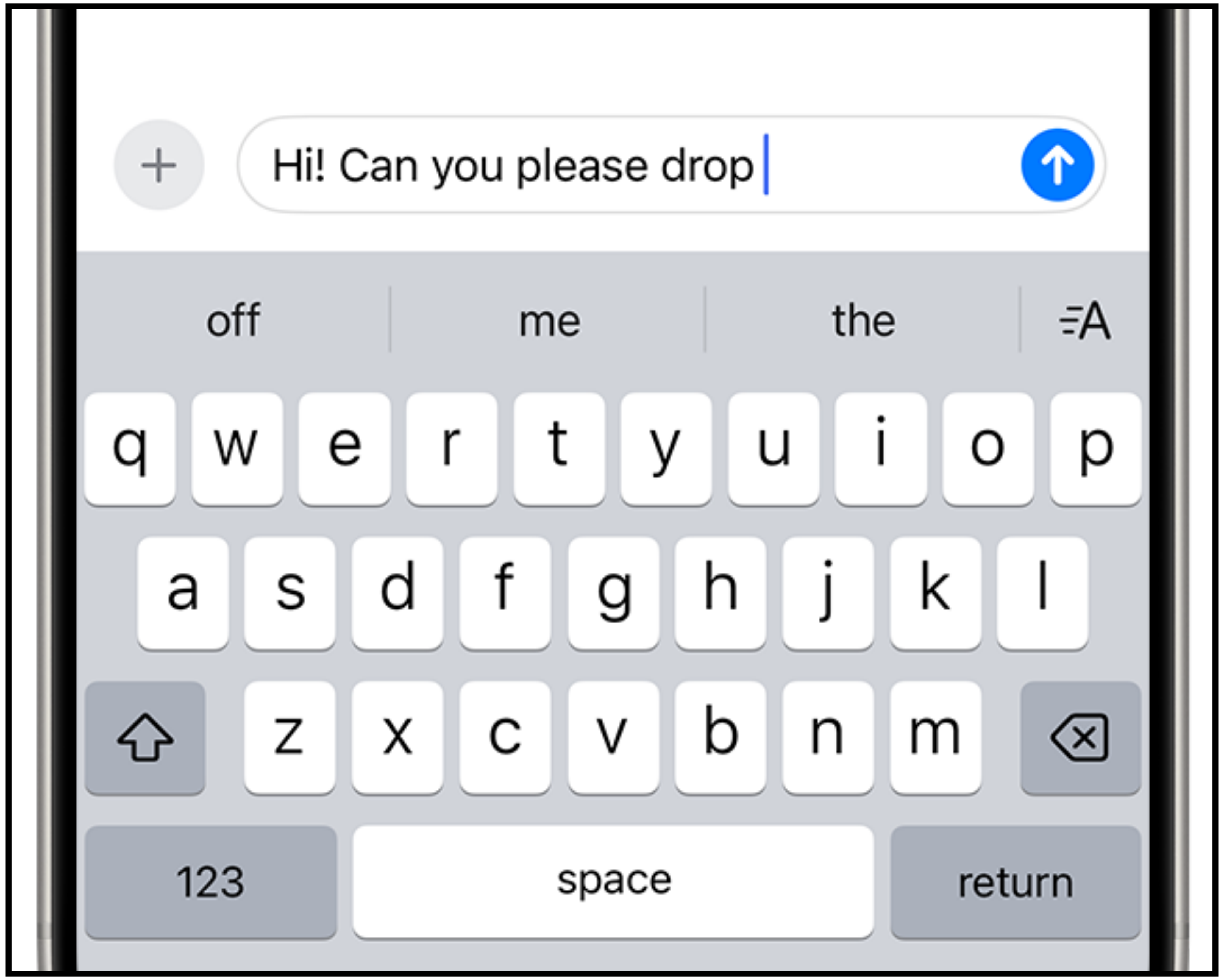
We'll use the **chain rule** to break this down:

$$p(w_1, w_2, \ldots, w_n) = \prod_{i=1}^{n} p(w_i \mid w_{<i})$$

This makes the definition more tractable:

A **language model** is a system that takes sequences of **tokens** as inputs, and produces a probability distribution over the next token.


Google
What are the most important
AI Mode
what are the most important vitamins
what are the most important nfl games today
what are the most important vaccines for babies
what are the most important vitamins to take
what are the most important amendments
what are the most important things in life


Hi! Can you please drop
off
me
the
q w e r t y u i o p
a s d f g h j k l
z x c v b n m
123
space
return

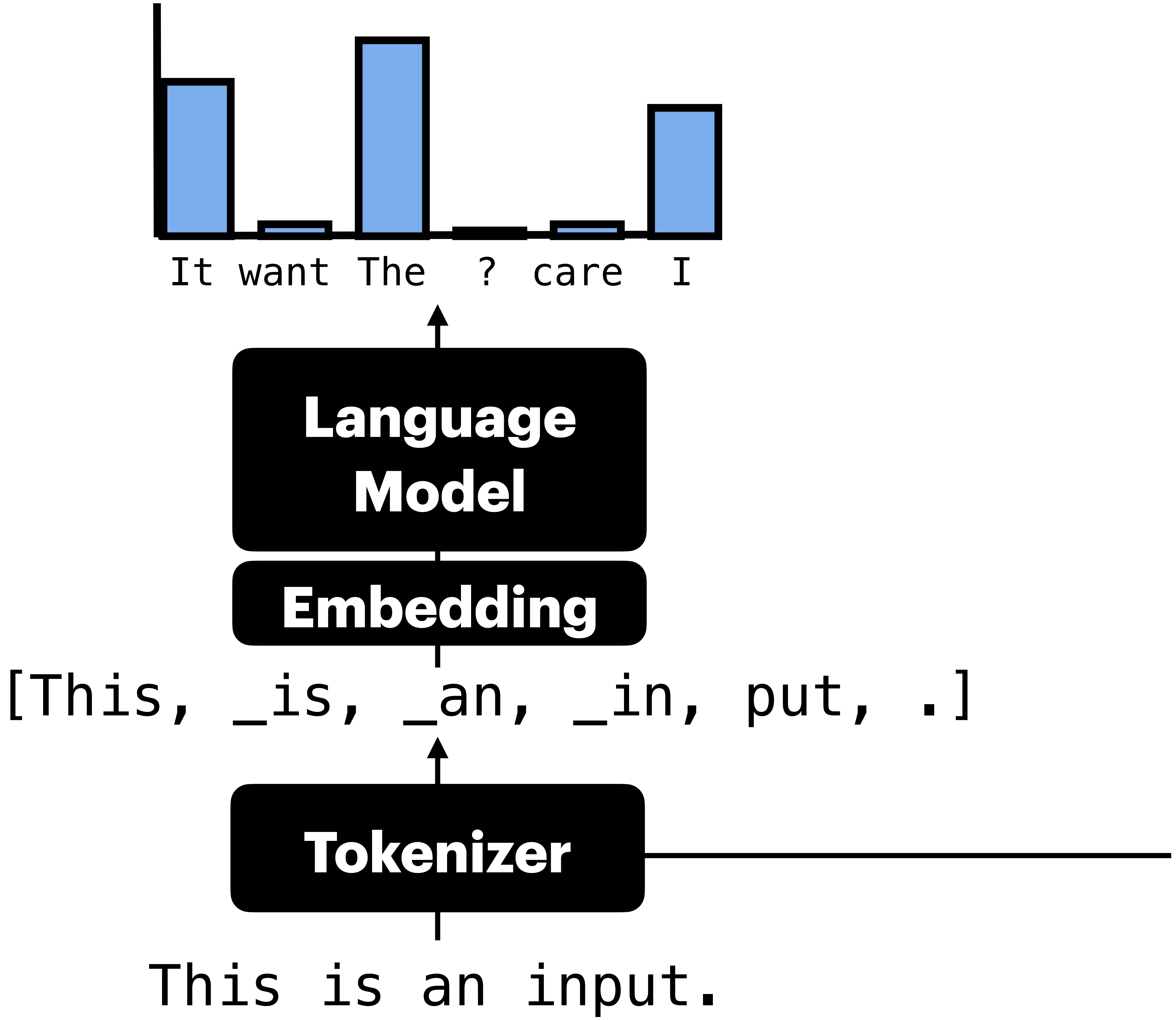
A **language model** is a system that takes sequences of **tokens** as inputs, and produces a probability distribution over next tokens:

$$p(w_1, w_2, \ldots, w_n) = \prod_{i=1}^{n} p(w_i \mid w_{<i})$$

Where w_i is part of a vocabulary V.

Over what units should we define our vocabulary? Words, characters, something else?

These questions relate to notions of **tokenization**—the mapping of a string to (lists of) tokens.



Tokens are the atomic unit of input to an NLP system.

In this example input, there are 4 words, but 6 tokens.

A **tokenizer** splits a string into **tokens**.

Words

For any NLP system, we need to define a *finite* **vocabulary.**

First idea: let's use the top-*k* most common words as our vocabulary.

Text:	The	man	saw	the	cat	.
Tokens:	**11**	**387**	**720**	**5**	**407**	**3**

Tokens are represented as *indices* in a vocabulary

What can you learn from context?

Boston University is in ________. **[Factual knowledge]**

Cats like to eat ___________. **[Factual knowledge]**

Where are _________ napkins? **[Parts of speech, sentence structure]**

15 x 5 = ________ **[Arithmetic]**

The keys to the cabinet ______ on the table. **[Subject-verb agreement]**

What can you learn from context?

1102 582 59 80 ________ 10 **[Factual knowledge]**

608 762 91 203 ___________ 10 **[Factual knowledge]**

1509 108 _________ 4092 11 **[Parts of speech, sentence structure]**

2091 102 2082 1011 ________ **[Arithmetic]**

81 2529 91 61 ______ 75 61 3520 10 **[Subject-verb agreement]**

Word Tokenization

- Not as simple as splitting based on whitespace!

*Mr. Johnson thinks the boys' stories about San Francisco **aren't** amusing.*

- There are lots of specialized rules about splitting things like contractions, punctuation, etc.

Check out spaCy's tokenizers for examples: https://spacy.io/api/tokenizer

Types vs. Tokens

This document is about cats. This document explains cats.

```
[This, document, is, about, cats, ., This, document, explains, cats, .]
```

Types are the unique items in a vocabulary.

If we use a word-level tokenizer, how many tokens do we have? **11**

How many types? **7**

The type–token distinction can be tricky:

- In a word-level tokenizer, are “the” and “The” distinct types?
- How about “the” and “_the”?

Corpus	Types = $\lvert V \rvert$	Instances = N
Shakespeare	31 thousand	884 thousand
Brown corpus	38 thousand	1 million
Switchboard telephone conversations	20 thousand	2.4 million
COCA	2 million	440 million
Google n-grams	13 million	1 trillion

Some of these datasets have a *huge* number of types!

Content Words vs. Function Words

- **Function words** are a *closed class:* they are finite, and you cannot (usually) add more
  - *Articles:* the, a
  - *Prepositions:* of, by, near
  - *Conjunctions:* and, but, yet

- **Content words** are an *open class:* they are, in theory, infinite
  - *Nouns:* cats, generosity, giants, apricity, grub
  - *Verbs:* fly, abvolate, yeet

The Finite Vocabulary Problem

- There are an *infinite* number of words. Thus, any finite vocabulary based in words will not fully cover natural language.
- What if we see tokens in the test set that weren’t in the training set? What if the vocabulary is too small for how big the dataset is?
- If we encounter a word we haven’t seen before, we replace it with a special `<UNK>` token.
    - `<UNK>` has its own representation and probability.
    - This token will kill our language model’s quality fast. We want to minimize how often this token appears as much as possible.

The Finite Vocabulary Problem

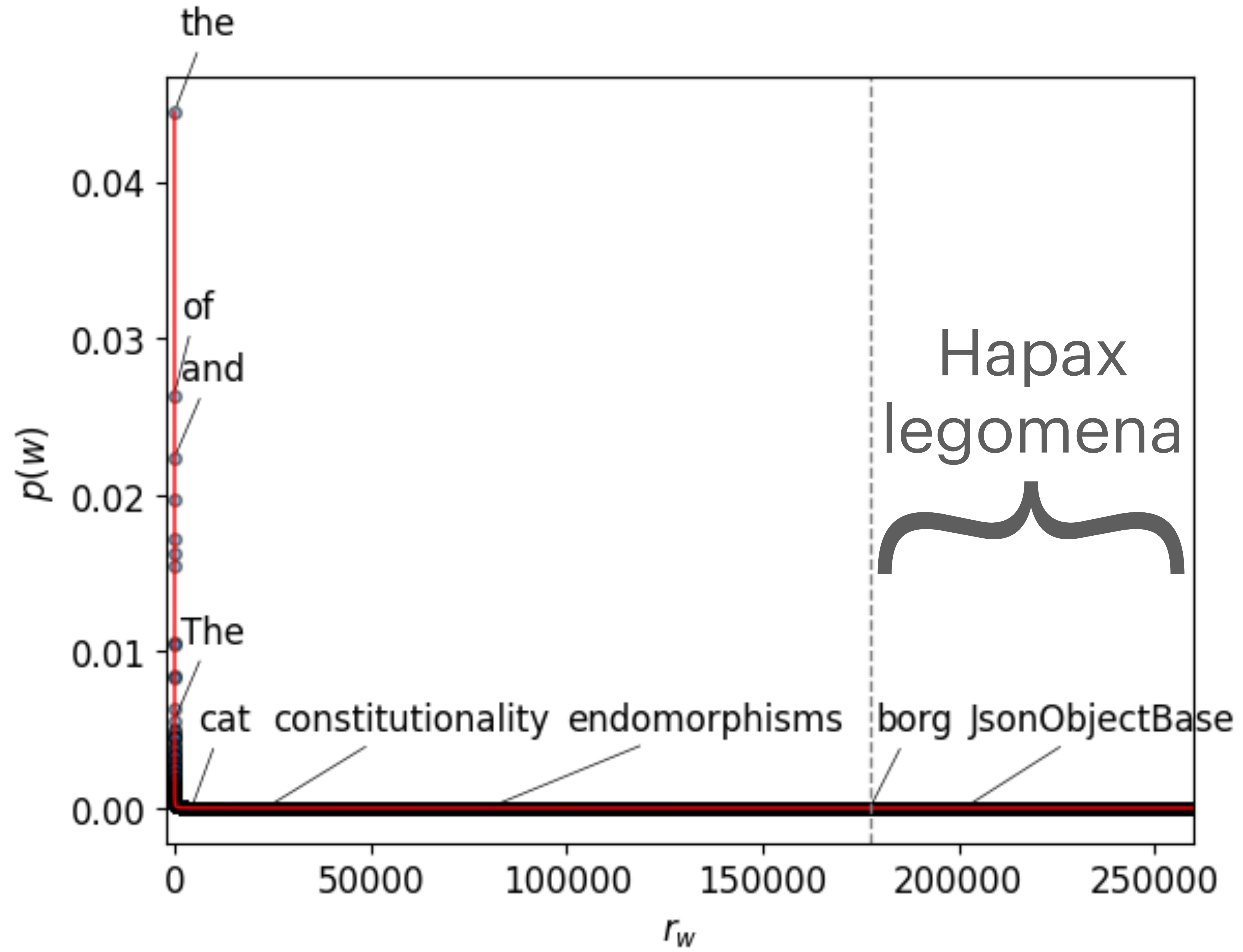
The subject was Argus-eyed; he perceived the glint of a feline's eyes.

If a word is outside our vocabulary, we'll replace it with <UNK>, a token for unknown or out-of-vocabulary tokens.

The subject was `<UNK>`; he `<UNK>` the `<UNK>` of a `<UNK>` eyes.

Zipf's Law

Zipf's Law: If we sort words by frequency, the probability of a word is inversely proportional to its rank:



$$p(w) \propto \frac{H}{r_w}$$

This means that *most* words are very rare!

Normalization and Lemmatization

Normalization: Standardizing text into a particular format

Examples:

Lowercasing: convert all capitals to lowercase

James left for Scotland → james left for scotland

Spelling correction:

James left for Scoltand

Abbreviation reformatting:

Ph.D., U.S.A. → PhD, USA

Normalization and Lemmatization

Lemmatization: Replacing inflected forms of a word with their uninflected roots:

ran, runs, running → run

cars, car → car

John worked late on projects. → John work late on project.

(Note: lemmatization is not always as easy as removing suffixes!
Consider “ran”, “stories”, “went”).

For real NLP systems, normalization is essential, but lemmatization is rare.

Other Limitations of Word Tokenizers

- Word-level tokenizers will consider different forms of the same word as different tokens:

run, runs, ran, running

apple, apples

- This means these forms will all have separate representations
- Also an issue in languages that have very complex **morphology.**

Morphemes

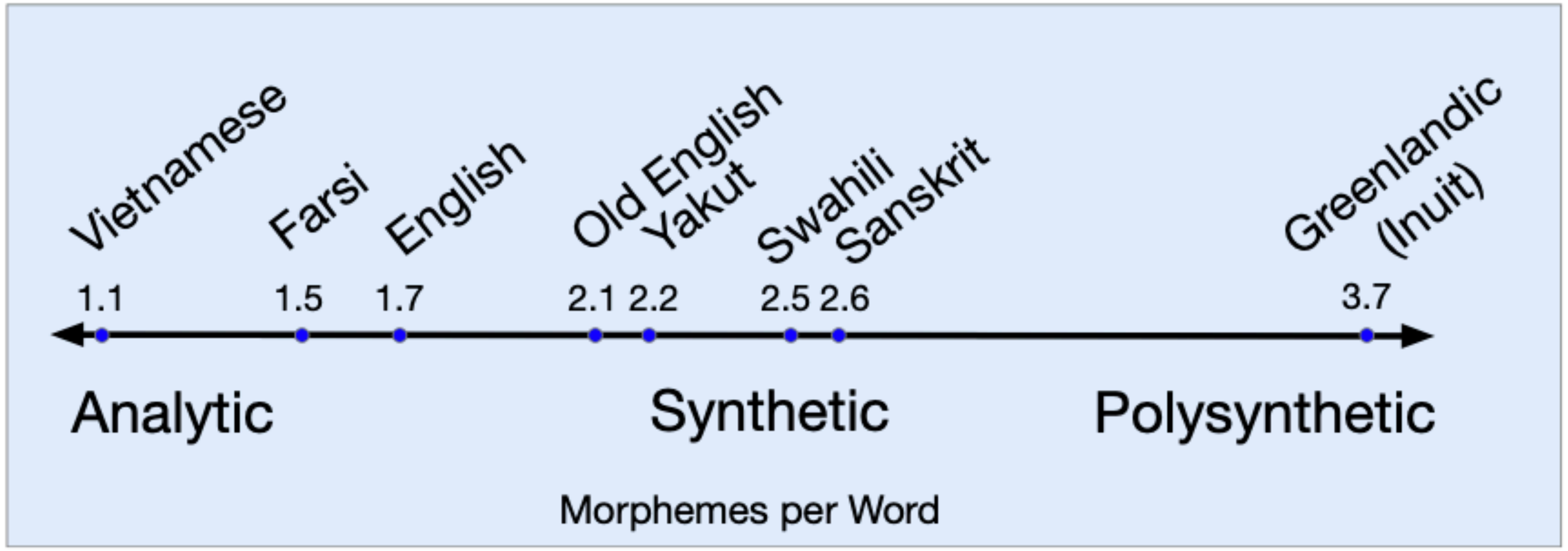
Çekoslovakyalılaştıramadıklarımızdanmısınız?

Çekoslovaky	alı	laş	tır	a	ma	dık	lar	ımız	dan	mı	sınız?
Czechoslovakia	OF	BECOME	CAUS	NEG	NEG	PST. PTCP	PL	1PL. POSS	ABL	Q	2PL. COP

"Are you one of those that we could not make into a Czechoslovakian?"

Each of these units of meaning is a **morpheme.**

Different languages have very different numbers of morphemes per word:

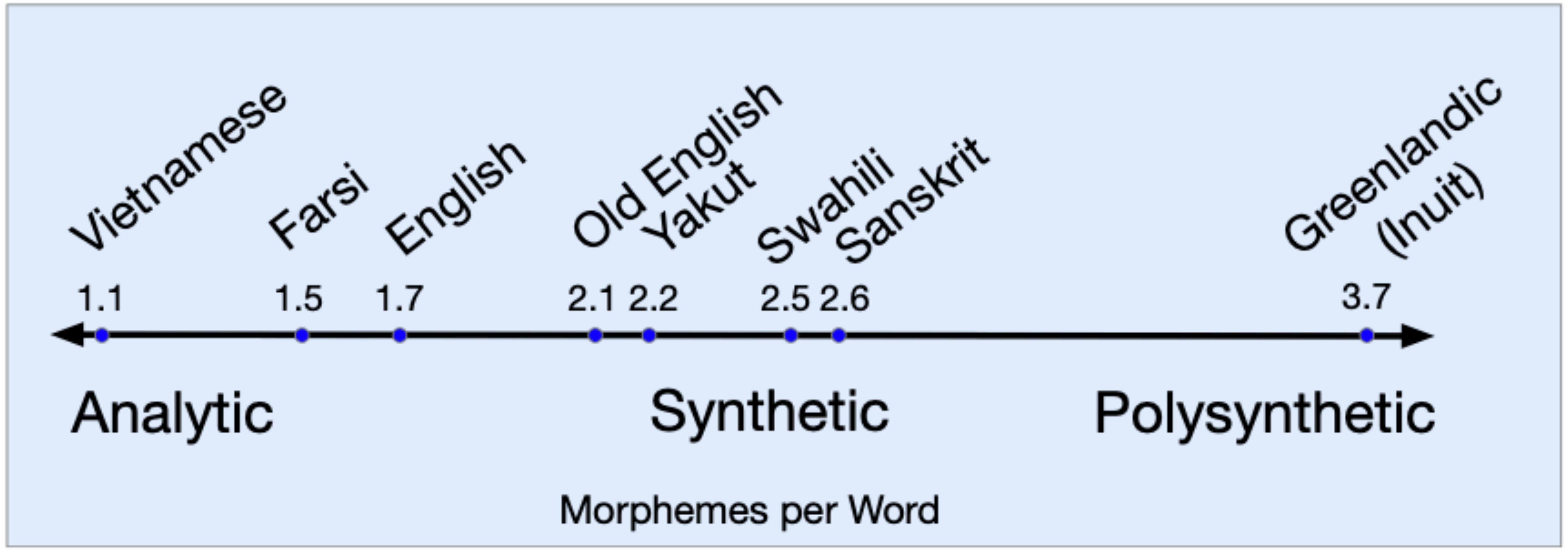


Morphemes

John	work - ed	late	on	project - s.
John	work - PAST	late	on	project - PL

Each of these units of meaning is a **morpheme.**

Different languages have very different numbers of morphemes per word:



Maybe we could split words into morphemes! Unfortunately, this is slow and hard… but inspired by this, let's pursue the idea of *splitting words into subwords*.

Characters

The man saw the cat.

↓ *Character-level tokenizer*

T,h,e,_,m,a,n,_,s,a,w,_,t,h,e,_,c,a,t,.

Pros:

- Solves the finite-vocabulary problem—to a degree.
  (But may not work as well for Chinese, which has >100,000 characters.)
- Easy to implement.

Cons:

- Can be hard to train a good language model. *Long* contexts,
  and the same character can appear in many different contexts.

2016: Subword Tokenization

- Developed for machine translation by **Sennrich et al. [2016]**

> "The main motivation behind this paper is that the translation of some words is transparent in that they are translatable by a competent translator even if they are novel to him or her, based on a translation of known subword units such as morphemes or phonemes."

- Later used in BERT, RoBERTa, GPT, among other models
- Relies on a simple algorithm called *byte-pair encoding*

Byte-pair Encoding

1. *Split corpus into characters.*

the man saw the cat.

t,h,e,_,m,a,n,_,s,a,w,_,t,h,e,_,c,a,t,.

2. *Count each pair of characters:*

(t,h): 2

(h,e): 2

(m,a): 1

(a,n): 1

...

3. Merge the highest-frequency pair into one token:

(t,h) -> th

th,e,_,m,a,n,_,s,a,w,_,th,e,_,c,a,t.

4. Repeat *m* times, where *m* is the number of merges (a hyperparameter).

Byte-pair Encoding

the man saw the cat.

↓

th,e,_,m,a,n,_,s,a,w,_,th,e,_,c,a,t.

2. *Count each pair of **tokens**:*

(th,e): 2

(m,a): 1

(a,n): 1

...

3. Merge the highest-frequency pair into one token:

(th,e) -> the

the,_,m,a,n,_,s,a,w,_,the,_,c,a,t.

4. Repeat *m* times, where *m* is the number of merges (a hyperparameter).

Byte-pair Encoding

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

1. *Split inputs into characters.*
2. *Count each pair of tokens.*
3. Merge the highest-frequency pair into a new token. <u>Do not merge across word boundaries.</u>
4. Repeat *k* times, where *k* is the number of merges (a hyperparameter).

Byte-pair Encoding

- To avoid <UNK>, all possible characters or symbols need to be in the base vocab. This can be a *lot*!
    - Unicode has hundreds of thousands, and growing!
- GPT-2 uses *bytes* as the base vocabulary (only 256 of them), and applies BPE on top of byte sequences (with some special rules to prevent certain kinds of merges).
- Usually our vocab is somewhere between 32K to 100K

Unicode

- We used an algorithm called *byte*-pair encoding, but over *characters*. What's the difference? What is a "character"?

- This almost always refers to **Unicode** characters.

- Unicode assigns a **code point** to each character.

- There are *a lot* of Unicode characters, so this doesn't solve the finite vocabulary problem.

U+0061	a	LATIN SMALL LETTER A
U+0062	b	LATIN SMALL LETTER B
U+0063	c	LATIN SMALL LETTER C
U+00F9	ù	LATIN SMALL LETTER U WITH GRAVE
U+00FA	ú	LATIN SMALL LETTER U WITH ACUTE
U+00FB	û	LATIN SMALL LETTER U WITH CIRCUMFLEX
U+00FC	ü	LATIN SMALL LETTER U WITH DIAERESIS
U+8FDB	进	
U+8FDC	远	
U+8FDD	违	
U+8FDE	连	
U+1F600	😀	GRINNING FACE
U+1F00E	🀎	MAHJONG TILE EIGHT OF CHARACTERS

UTF-8 and Bytes

Code Points		UTF-8 Encoding			
From - To	Bit Value	Byte 1	Byte 2	Byte 3	Byte 4
U+0000-U+007F	0xxxxxxx	xxxxxxxx			
U+0080-U+07FF	00000yyy yyxxxxxx	110yyyyy	10xxxxxx		
U+0800-U+FFFF	zzzzyyyy yyxxxxxx	1110zzzz	10yyyyyy	10xxxxxx	
U+010000-U+10FFFF	000uuuuu zzzzyyyy yyxxxxxx	11110uuu	10uuzzzz	10yyyyyy	10xxxxxx

- A **byte** is 8 bits, so it can take values in [0, 255].
- In UTF-8, a character contains a variable number of bytes. E.g., ‘ñ' has Unicode code point `U+00F1`, and bytes `C3 B1 (195, 177)`
- There are only 256 possible bytes, so a tokenizer based on bytes would have full coverage!
- A byte-based LM could generate invalid Unicode, however, which would yield a meaningless sequence

Implementation Details

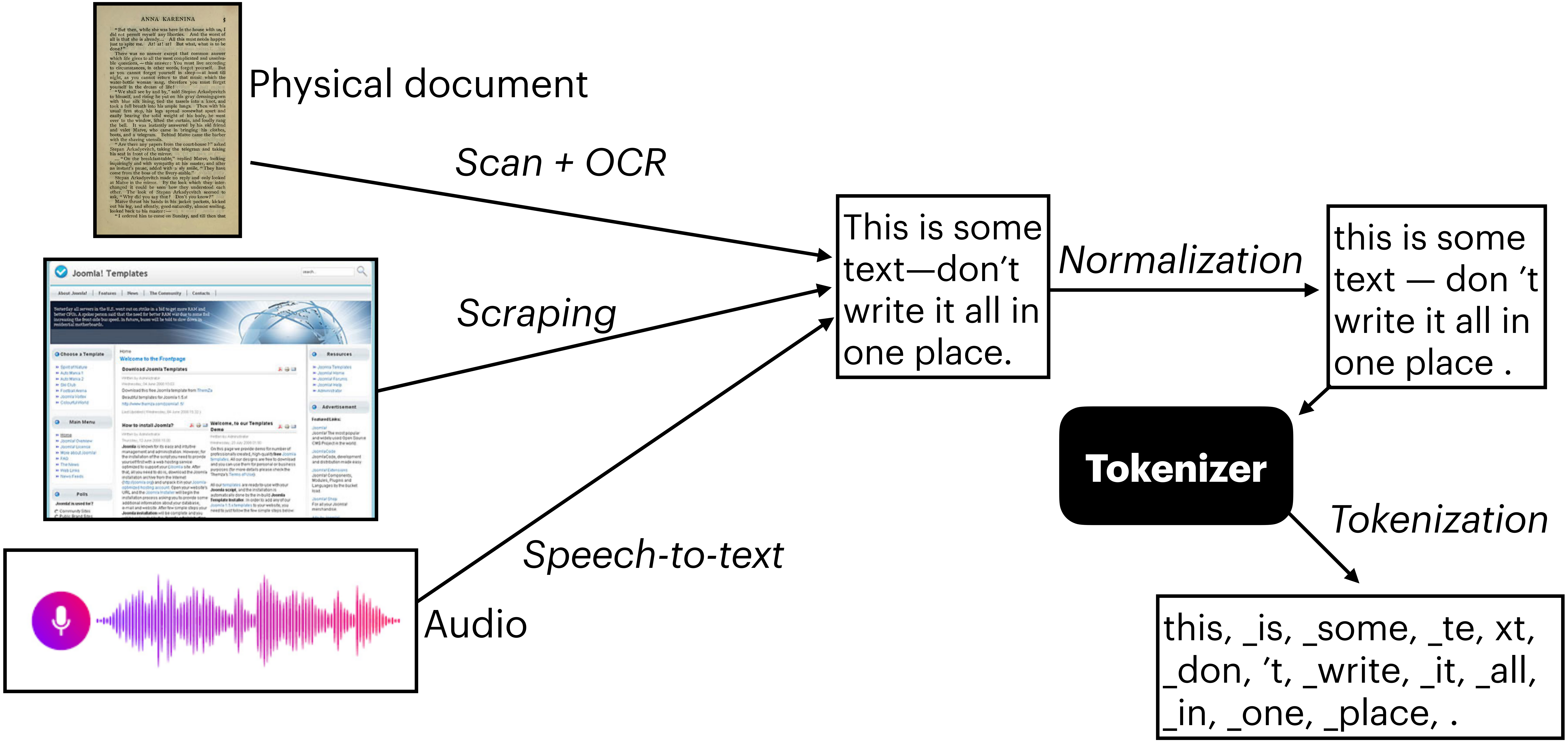
- In practice, common tokenizers tend to use subword vocabularies with tens of thousands to hundreds of thousands of entries.
  - BERT (2018): 30,522
  - GPT-2 (2019): 50,257
  - Llama 3.1 (2024): 128,256
  - GPT-4o (2024): ≈200,000
  - Gemma 3 (2025): 256,000

Corpora

- We usually train our tokenizers and language models on **corpora**—collections of documents.
- No corpus is fully representative of all natural language. Documents are written:
    - By specific people
    - From a specific time and place
    - In a specific language variety
    - For some specific purpose(s).

- These days, language models are trained primarily on internet-based corpora.
    - The internet has tons of useful information and knowledge!
    - …But also a great deal of negativity and hatred.

Collecting a Corpus



Problems in Tokenization

- A tokenizer trained well for one corpus may not generalize well because of:
    - **Language imbalance**: A great English tokenizer would not necessarily be a good Turkish tokenizer
    - **Domain shift**: A tokenizer that works well for scientific articles would not necessarily work well for social media
    - **Temporal shift**: A tokenizer trained on internet text from before the year 2000 may not effectively handle text from the 2025 internet.

Problems in Tokenization

Handling numbers is particularly tricky. Let's say you want to represent this sequence:

```
85,219 x 20 =
```

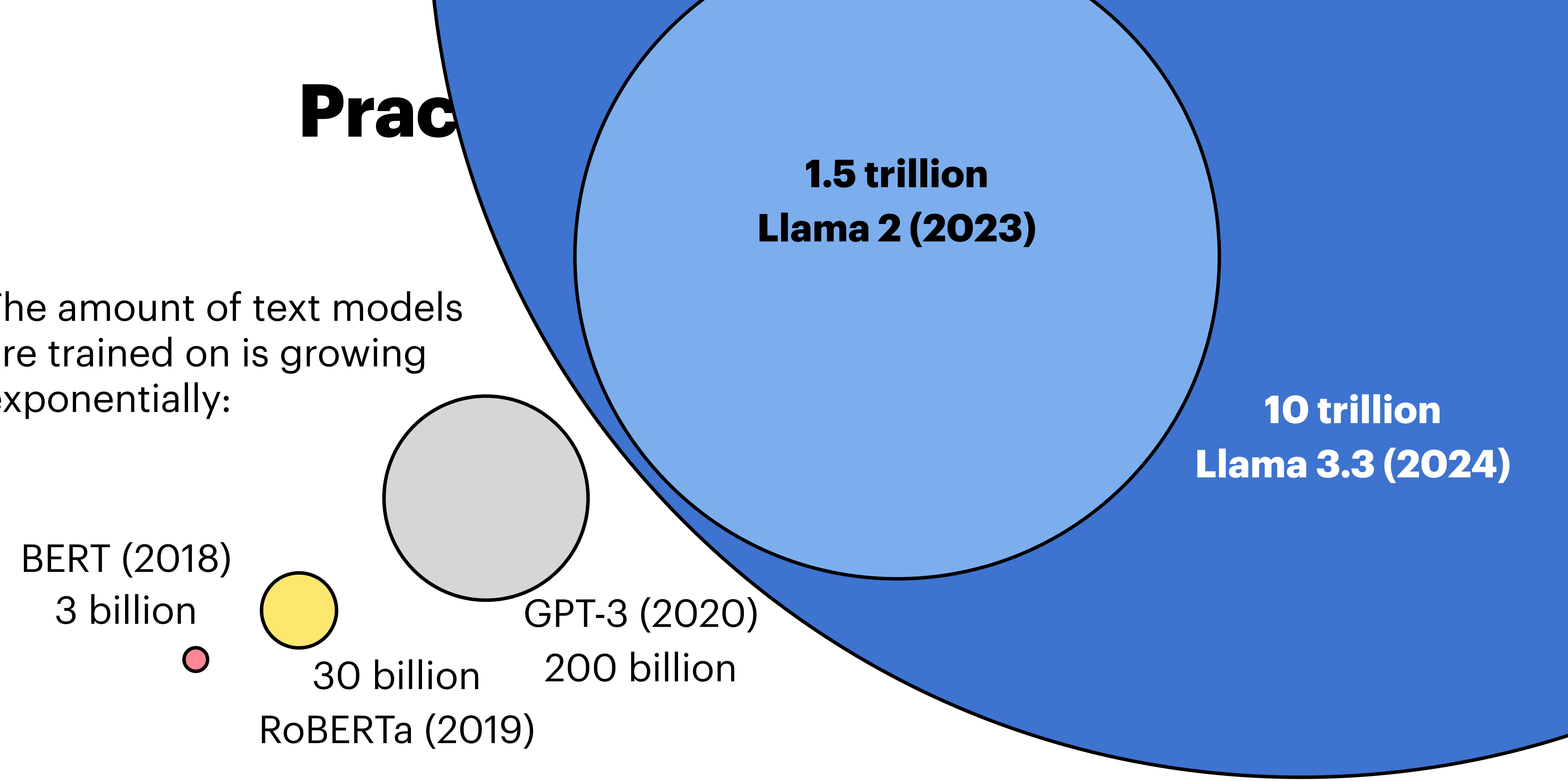
A BPE-based tokenizer might spit out something like:

```
[8, 5, ,, 21, 9, x, 20, =]
```

Clearly this isn't great. Some models (like Gemma 2) just split all digits into their own tokens; others (like Llama 3) preserve common multi-digit sequences. There are trade-offs to both approaches.

Prac

The amount of text models are trained on is growing exponentially:



It is *impossible* to process this much text by hand. This is an issue when most gains in NLP come from **data** these days, and not from algorithmic innovations.

Regular Expressions

Regular Expressions

- A.k.a., **regex**
- Used in every computer language. Some regex tools you may have used:
  - Unix grep
  - Python re

- Can be used to:
  - Find strings of a certain type
  - Search large corora
  - *Preprocess text*

Regex Tokenizers

```
>>> text = 'That U.S.A. poster-print costs $12.40...'
>>> pattern = r'''(?x)    # set flag to allow verbose regexps
...     ([A-Z]\.)+        # abbreviations, e.g. U.S.A.
...   | \w+(-\w+)*        # words with optional internal hyphens
...   | \$?\d+(\.\d+)?%?  # currency and percentages, e.g. $12.40, 82%
...   | \.\.\.            # ellipsis
...   | [][.,;"'?():-_`]  # these are separate tokens; includes ], [
... '''
>>> nltk.regexp_tokenize(text, pattern)
['That', 'U.S.A.', 'poster-print', 'costs', '$12.40', '...']
```

The Natural Language Toolkit (nltk)'s word tokenizer is based on regular expressions.

Character Disjunctions

Square brackets indicate logical ORs (disjunctions) or ranges:

Pattern	Match	String
`r"[mM]ary"`	Mary or mary	"Mary Ann stopped by Mona's"
`r"[abc]"`	'a', 'b', *or* 'c'	"In uomini, in soldati"
`r"[1234567890]"`	any one digit	"plenty of 7 to 5"

You can tell the regex what *not* to find using a carat (^):

Regex	Match (single characters)	Example Patterns Matched
`r"[^A-Z]"`	not an upper case letter	"Oyfn pripetchik"
`r"[^Ss]"`	neither 'S' nor 's'	"I have no exquisite reason for't"
`r"[^.]"`	not a period	"our resident Djinn"
`r"[e^]"`	either 'e' or '^'	"look up ^ now"
`r"a^b"`	the pattern 'a^b'	"look up a^ b now"

Counting Characters

Regex	Match
`*`	zero or more occurrences of the previous char or expression
`+`	one or more occurrences of the previous char or expression
`?`	zero or one occurrence of the previous char or expression
`{n}`	exactly *n* occurrences of the previous char or expression
`.`	any single char
`.*`	any string of zero or more chars

ba*: matches b, b<u>a</u>, b<u>aaaaa</u>

ba+: matches b<u>a</u>, b<u>aaaaa</u>

ba?: matches b or b<u>a</u>

ba{3}: matches b<u>aaa</u>

b.: matches b<u>a</u>, b<u>b</u>, b<u>4</u>, ...

b.*: matches anything that starts with b

Anchors

Regex	Match
^	start of line
$	end of line
\b	word boundary
\B	non-word boundary

These allow you to specify *where* a regex should be matched.

Example: say you want to find sentences containing the word “the”.

r”the”: Doesn’t catch capitalized “The”!

r”[tT]he”: might match things like “ba<u>the</u>” or “<u>The</u>me”

r”\b[tT]he\b”: only matches words “The” or “the”!

Order of Operations

Parenthesis	`()`
Counters	`* + ? {}`
Sequences and anchors	`the ^my end$`
Disjunction	`|`

r"the*" matches "th<u>eeeee</u>" but not "thethe" because sequences are processed after counters.

r"the|any" matches "the" or "any" but not "thany" because disjunctions are processed after sequences.

Application: Word Tokenizer

```
>>> text = 'That U.S.A. poster-print costs $12.40...'
>>> pattern = r'''(?x)    # set flag to allow verbose regexps
...     ([A-Z]\.)+        # abbreviations, e.g. U.S.A.
...   | \w+(-\w+)*        # words with optional internal hyphens
...   | \$?\d+(\.\d+)?%?  # currency and percentages, e.g. $12.40, 82%
...   | \.\.\.            # ellipsis
...   | [][.,;"'?():-_`]  # these are separate tokens; includes ], [
... '''
>>> nltk.regexp_tokenize(text, pattern)
['That', 'U.S.A.', 'poster-print', 'costs', '$12.40', '...']
```

Application: BPE Pre-tokenizer

Before we apply the BPE algorithm, we usually do the following:

Split contractions off from their roots:

Split words from each other:

Split numbers from each other:

Split punctuation into separate tokens:

Handle remaining whitespace:

(This is the actual pre-tokenizer for GPT-2!)

```
>>> import regex as re
>>> pat = re.compile(
... # Contractions: 't and 'm are tokens
...     r"'s|'t|'re|'ve|'m|'ll|'d|"
... # Words:  sequence of Unicode letters (after optional space)
...     r" ?\p{L}+|"
... #Number: sequence of digits (after optional space)
...     r" ?\p{N}+|"
... # Punctuation: sequence of non-alphanumeric/non-space
...                             #(after optional space)
...     r" ?[^\s\p{L}\p{N}]+|"
... # whitespace
...     r"\s+(?!\S)|\s+"
... )
>>> text = "We're 350 dogs! Um, lunch?"
>>> print(pat.findall(text))
['We', "'re", ' 350', ' dogs', '!', ' Um', ',', ' lunch', '?']
>>>
```

Substitutions

Substitutions allow you to replace one string with another:

string = “The cherry grove was red.”

↓

re.sub(“cherry”, “apricot”, string)

↓

“The apricot grove was red.”

Substitutions

- Substitutions can be very powerful. Here's a regex for converting date formats:

```
re.sub(r"(\d{2})/(\d{2})/(\d{4})", r"\2-\1-\3", string)
```

A string of form "01/15/1985" A date of form "15-01-1985"

- You can use them to find and remove repeated words in a string:

```
re.sub(r"\b([A-Za-z]+)\s+\1\b", "", string)
```

Finds a sequence of letters of length at least 1

Captures this sequence as a group and looks back for it after a whitespace

Deletes it

Application: A Simple Chatbot

```
re.sub(r".* YOU ARE (DEPRESSED|SAD) .*",r"I AM SORRY TO HEAR YOU ARE \1",input)
re.sub(r".* YOU ARE (DEPRES
re.sub(r".* ALWAYS .*",r"CA
```

Locates instances of “You are depressed/s

Locates instances of “You are depressed/s

Locates instances of “ always ”, replies wi

```
  Welcome to
                    EEEEEE  LL      IIII   ZZZZZZ   AAAAA
                    EE      LL       II       ZZ   AA   AA
                    EEEEE   LL       II      ZZZ   AAAAAAA
                    EE      LL       II     ZZ     AA   AA
                    EEEEEE  LLLLLL  IIII ZZZZZZ    AA   AA

  Eliza is a mock Rogerian psychotherapist.
  The original program was described by Joseph Weizenbaum in 1966.
  This implementation by Norbert Landsteiner 2005.


ELIZA: Is something troubling you ?
YOU:   Men are all alike.
ELIZA: What is the connection, do you suppose ?
YOU:   They're always bugging us about something or other.
ELIZA: Can you think of a specific example ?
YOU:   Well, my boyfriend made me come here.
ELIZA: Is it important to you that your boyfriend made you come here ?
YOU:   He says I'm depressed much of the time.
ELIZA: I am sorry to hear that you are depressed.
YOU:   It's true. I am unhappy.
ELIZA: Can you explain what made you unhappy ?
YOU:
```

Next Time

- A brief review of probability theory
- Using tokens to build our first language models: **n-gram language models**
- What can you learn from a token frequeny? From *pairs of* tokens? From *triplets of* tokens?