Words and Tokens

Words

How many words in a sentence?

`They picnicked by the pool, then lay back on the grass and looked at the stars.`

16 words

- if we don't count punctuation marks as words

18 if we count punctuation

How many words in an utterance?

"I do uh main- mainly business data processing"

Disfluencies

- Fragments *main-*
- *Filled pauses: uh* and *um*

- Should we consider these to be words?

How many words in a sentence?

`They picnicked by the pool, then lay back on the grass and looked at the stars.`

Type: an element of the vocabulary V

- The number of types is the vocabulary size |V|

Instance: an instance of that type in running text.

- 14 types and 16 instances (if we ignore punctuation).

More questions: Are `They` and `they` the same word?

How many words in a sentence?

`I'm`

Orthographically one word (in the English writing system)

But **grammatically** two words:

1. the subject pronoun I
2. the verb 'm, short for am

How many words in a sentence?

Not every written language uses spaces!!

Chinese, Japanese and Thai don't!

How to choose tokens in Chinese

Chinese words are composed of characters called "**hanzi" (汉字, hànzì)** (or sometimes just "**zi**")

Each one represents a meaning unit called a morpheme.

Each word has on average 2.4 of them.

But deciding what counts as a word is complex and not agreed upon.

How to choose tokens in Chinese?

姚明进入总决赛 "Yao Ming reaches the finals"

- yáo míng jìn rù zǒng jué sài

3 words?

姚明	进入	总决赛	
YaoMing	reaches	finals	Chinese Treebank

5 words?

姚	明	进入	总	决赛	
Yao	Ming	reaches	overall	finals	Peking University

7 words?

姚	明	进	入	总	决	赛	
Yao	Ming	enter	enter	overall	decision	game	Just use characters

Tokenization across languages

- So in Chinese we use characters (zi) as tokens
  - But that doesn't work for, e.g., Thai and Japanese
  - These differences make it hard to use words as tokens
- And there's another reason why we don't use words as tokens!

There are simply too many words!

	Types = |V|	Instances = N
Shakespeare	31 thousand	884,000
Brown Corpus	38 thousand	1 million
Switchboard conversations	20 thousand	2.4 million
COCA	2 million	440 million
Google N-grams	13+ million	1 trillion

The bigger the corpus, the more word types!

There are simply too many words!

N = number of instances

$|V|$ = number of types in vocabulary V

Heaps Law = Herdan's Law

$$|V| = kN^{\beta}$$

← Roughly 0.5

Vocab size for a text goes up with the square root of its length in words

Two kinds of words

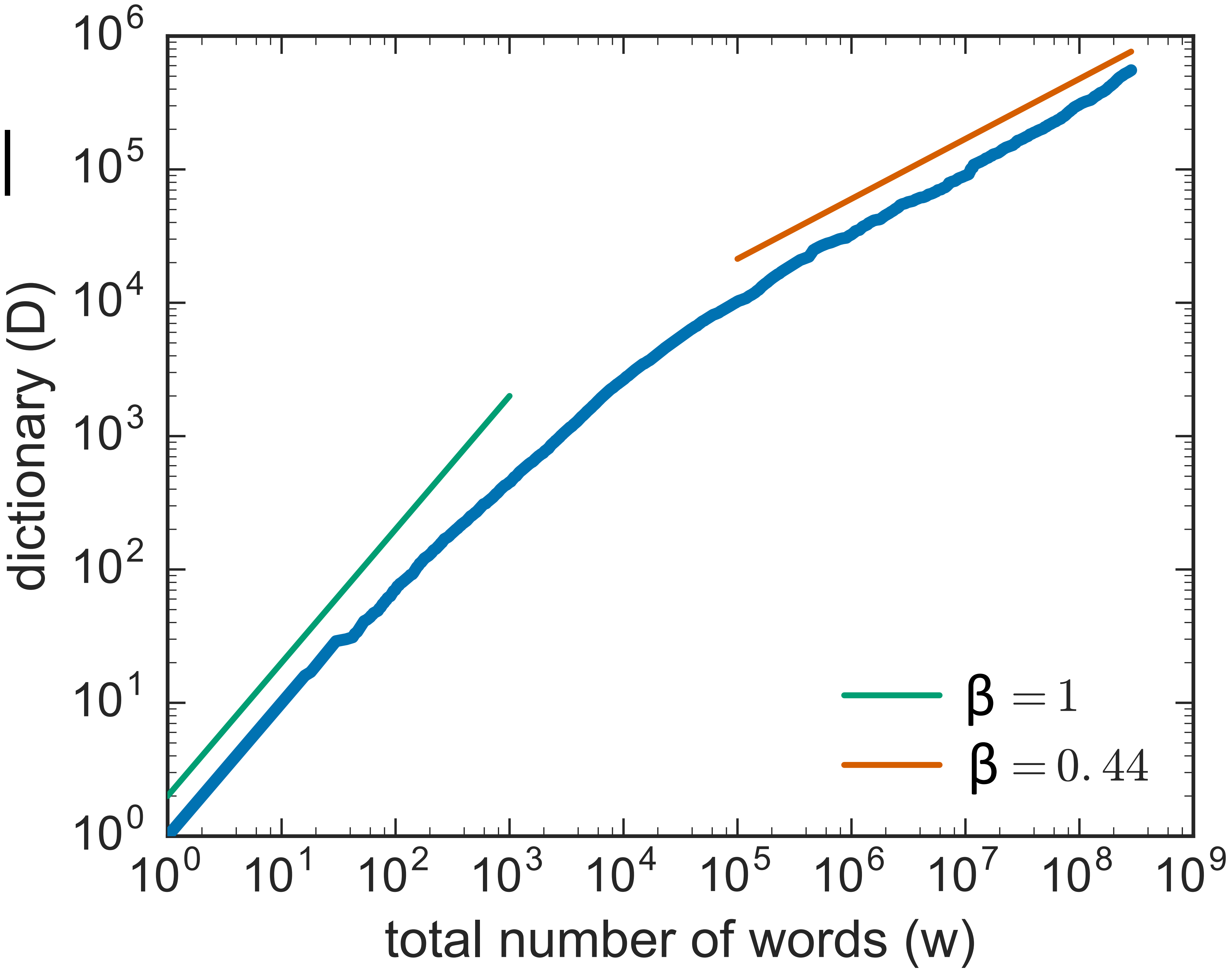
Function words

- of, the, is, and, una, 是,

Content words

- mango, braise, snowy, feliz, 北京

|V|



Tria, Loreto, Servedio, 2018

Why is too many words a problem?

No matter how big our vocabulary
There will always be words we missed!
We will always have unknown words!

Words and Subwords

Because of these three problems:

1. Many languages don't have orthographic words
2. Defining words post-hoc is challenging
3. The number of words grows without bound

NLP systems don't use words, but smaller **subwords**

Words and Tokens

Words

Words and Tokens

Morphemes

Words have parts

Morpheme: a minimal meaning-bearing unit in a language.

`fox`: one morpheme

`cats`: two morphemes `cat` and `-s`

Morphology: the study of morphemes

Morphemes in English and Chinese

`Doc work-ed care-ful-ly wash-ing the glass-es`

梅 干 菜 用 清 水 泡 软 ，捞 出 后 ，沥 干
plum dry vegetable use clear water soak soft , remove out after , drip dry
切 碎
chop fragment

Soak the preserved vegetable in water until soft, remove, drain, and chop

Types of morphemes

root: central morpheme of the word
- supplying the main meaning

affix: adding additional meanings

`worked`

root `work`

affix `-ed`

`glasses`

root `glass`

affix `-es`

Types of affixes

Inflectional morphemes

- grammatical morphemes
- often syntactic role like agreement

  `-ed` past tense on verbs

  `-s/-es` plural on nouns

Derivational morphemes

- more idiosyncratic in application and meaning
- often change grammatical class

  care (noun)

  \+ -*full* → *careful* (*adjective*)

  \+ -*ly* → *carefully* (*adverb*)

Clitics

A morpheme that acts syntactically like a word but:

- is reduced in form
- and attached to another word

English: `'ve` in `I've` (`'ve` can't appear alone)

English: `’s` in `the teacher’s book`

French: `l’` in `l’opera`

Arabic: `b` ‘by/with’, `w` ‘and’.

Morphological Typology

Dimensions along which languages vary

Two are salient for tokenization:

1. number of morphemes per word
2. how easy it is to segment the morphemes

Number of morphemes per word

Few. *Cantonese, spoken in Guangdong, Guangxi, Hong Kong*

keoi5 waa6 cyun4 gwok3 zeoi3 daai6 gaan1 uk1 hai6 ni1 gaan1

he say entire country most big building house is this building

"He said the biggest house in the country was this one"

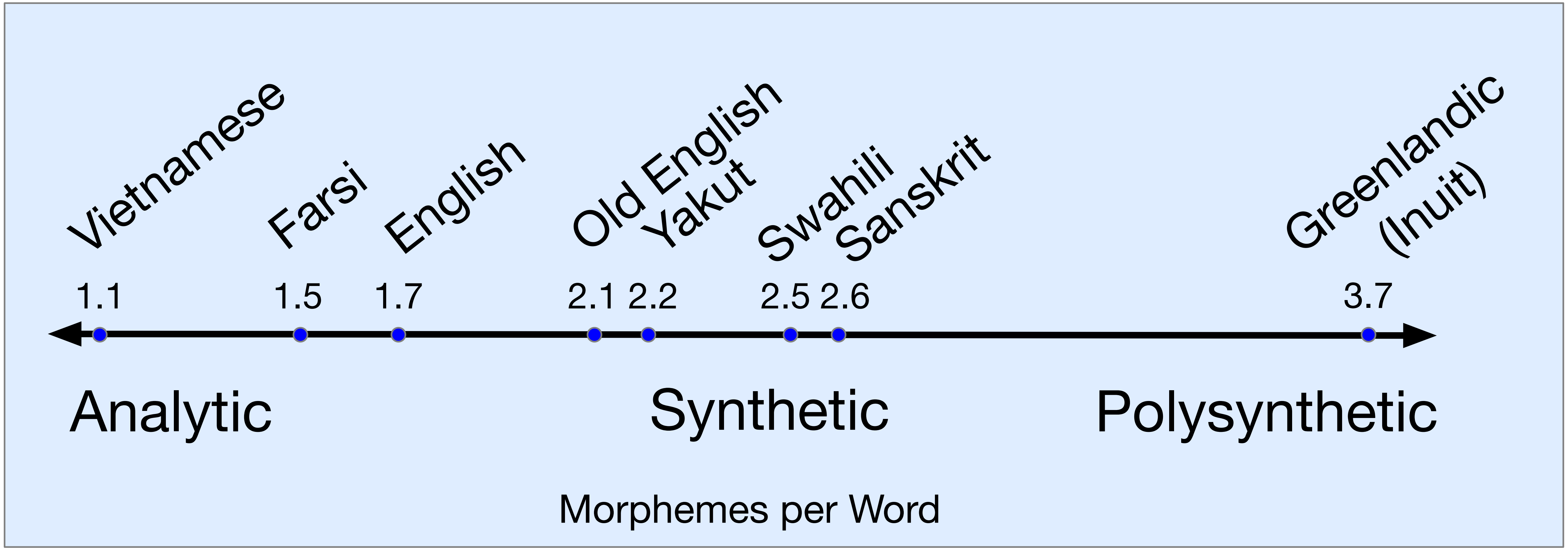
Many. Koryak, Kamchatka peninsula in Russia,

t-ə-nk'e-mejŋ-ə-jetemə-nni-k

1SG.S-E-midnight-big-E-yurt.cover-E-sew-1SG.S[PFV]

"I sewed a lot of yurt covers in the middle of a night."

Joseph Greenberg (1960) scale



How easily segmentable

Agglutinative languages like Turkish

- Very clean boundaries between morphemes

Fusion languages

- a single affix may conflate multiple morphemes,

- Russian `-om` in `stolom` (table-SG-INSTR- DECL1)
  - instrumental, singular, and first declension.
- English `-s` in "`She reads the article`"
  - Means both "third person" and "present tense"

These are tendencies rather than absolutes

Words and Tokens

Morphemes

Words and Tokens

Unicode

Unicode

a method for representing text written using

- any character (more than 150,000!)
- in any script (168 to date!)
- of the languages of the world
  - Chinese, Arabic, Hindi, Cherokee, Ethiopic, Khmer, N'Ko,…
  - dead ones like Sumerian cuneiform
  - invented ones like Klingon
  - plus emojis, currency symbols, etc.

ASCII: Some history for English

1960s American Standard Code for Information Interchange

1 byte per character

- In principle 256 characters
- But high bit set to 0
- So 7 bits = 128
- However only 95 used

The rest were for teletypes

ASCII: Some history for English

Ch	Hex	Dec	Ch	Hex	Dec		Ch	Hex	Dec	Ch	Hex	Dec
<	3C	60	@	40	64	…	\	5C	92	`	60	96
=	3D	61	A	41	65	…	[	5D	93	a	61	97
>	3E	62	B	42	66	…	^	5E	94	b	62	98
?	3F	63	C	43	67	…	_	5F	95	c	63	99

```
h  e  l  l  o
68 65 6C 6C 6F
```

ASCII wasn't enough!

Spanish: Señor- respondió Sancho

This sentence has non-ASCII ñ and ó

About 100,000 **Chinese/CJKV** characters (Chinese, Japanese, Korean, or Vietnamese)

Devanagari script for 120 languages like **Hindi**, Marathi, Nepali, Sindhi, Sanskrit, etc.

अनुच्छेद १(एक): सभी मनुष्य जन्म से स्वतन्त्र तथा मर्यादा और अधिकारों में समान होते हैं। वे तर्क और विवेक से सम्पन्न हैं तथा उन्हें भ्रातृत्व की भावना से परस्पर के प्रति कार्य करना चाहिए।

Code Points

Unicode assigns a unique ID, a **code point,** to each of its 150,000 characters

1.1 million possible code points

- 0 – 0x10FFFF

Written in hex, with prefix "U+"

- a is U+0061 which = 0x0061

First 127 code points = ASCII

- For backwards compatibility

Some code points

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

A code point has no visuals; it is **not** a glyph!

Glyphs are stored in **fonts**: **a** or *a* or a or *a*

But one code point (U+0061, abstract "LATIN SMALL A") represents all those different a's!

Encodings and UTF-8

We don't stick code points directly in files

We store **encodings** of chars.

The most popular encoding is UTF-8

Most of the web is stored in UTF-8

Encodings

`hello` has these 5 code points:

U+0068 U+0065 U+006C U+006C U+006F

How to write in a file?

There are more than 1 million code points

So would need 4 bytes (or 3 but 3 is inconvenient):

00 00 00 68 00 00 00 65 00 00 00 6C 00 00 00 6C 00 00 00 6F

But that would make files very long!

- Also zeros are bad (since mean "end of string" in ASCII)

Instead: Variable Length Encoding

UTF-8 (Unicode Transformation Format 8)

For the first 127 code points, same as ASCII

UTF-8 encoding of `hello` is :

- 68 65 6C 6C 6F

Code points ≥128 are encoded as a sequence of 2, 3, or 4 bytes

- In range 128 - 255, so won't be confused with ASCII
- First few bits say if its 2-byte, 3-byte, or 4-byte

UTF-8 Encoding

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

yyy yyxxxxxx

ñ, code point U+00F1, = 00000**000 11110001**

- Gets encoded with pattern 110yyyyy 10xxxxxx
- So is mapped to a two-byte bit sequence
- 110**00011** 10**110001** = 0xC3B1.

UTF-8 encoding

The first 127 characters (ASCII) map to 1 byte

Most remaining characters in European, Middle Eastern, and African scripts map to 2 bytes

Most Chinese, Japanese, and Korean characters map to 3 bytes

Rarer CJKV characters, emojis/symbols map to 4 bytes.

UTF-8 encoding

Efficient: fewer bytes for common characters,

Doesn't use **zero bytes** (except for NULL character U+0000),

Backwards compatible with **ASCII**,

Self-synchronizing,

- If a file is corrupted, the nearest character boundary is always findable by moving only up to 3 bytes

UTF-8 and Python 3

Python 3 strings stored internally as Unicode

- each string a sequence of Unicode code points
- string functions, regex apply natively to code points.
  - **len() returns string length in code points, not bytes**

Files need to be encoded/decoded when written or read

- Every file is stored in some encoding
- ***No such thing as a text file without an encoding***
  - If it's not UTF-8 it's something older like ASCII or iso_8859_1

Words and Tokens

Unicode

Byte Pair Encoding

The NLP standard for tokenization

Instead of

- white-space / orthographic words
  - Lots of languages don't have them
  - The number of words grows without bound
- Unicode characters
  - Too small as tokens for many purposes
- morphemes
  - Very hard to define

We use the data to tell us how to tokenize.

Why tokenize?

Using a deterministic series of tokens means systems can be compared equally

- Systems agree on the length of a string

Algorithms like perplexity assume all texts have a fixed tokenization

Eliminates the problem of unknown words

If some word occurs in test set but not training set, we still know how to segment it into known tokens.

Subword tokenization

Two most common algorithms:

- **Byte-Pair Encoding (BPE)** (Sennrich et al., 2016)
- **Unigram language modeling tokenization** (Kudo, 2018) (sometimes confusingly called "SentencePiece" after the library it's in)

All have 2 parts:

- A token **learner** that takes a raw training corpus and induces a vocabulary (a set of tokens).
- A token **encoder/segmenter** that takes a raw test sentence and tokenizes it according to that vocabulary

Byte Pair Encoding (BPE) token learner

Iteratively merge frequent neighboring tokens to create longer tokens.

Repeat:

- Choose most frequent neighboring pair ('A', 'B')
- Add a new merged symbol ('AB') to the vocabulary
- Replace every 'A' 'B' in the corpus with 'AB'.

Until *k* merges

Vocabulary

[A, B, C, D, E]

[A, B, C, D, E, AB]

[A, B, C, D, E, AB, CAB]

Corpus

A B D C A B E C A B

AB D C AB E C AB

AB D CAB E CAB

BPE token learner algorithm

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 til 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

Byte Pair Encoding (BPE) Addendum

Generally run **within** space-separated words

Don't merge across word boundaries

- First separate corpus by whitespace or similar, using specialized regular expressions
- This gives a set of starting strings, with whitespace attached to start of each strong
- Counts come from the corpus, but can only merge within strings.

BPE token learner

Original (very fascinating🙄) corpus:

`set␣new␣new␣renew␣reset␣renew`

Put space token at start of words

corpus		vocabulary
2	␣ n e w	␣, e, n, r, s, t, w
2	␣ r e n e w	
1	s e t	
1	␣ r e s e t	

BPE token learner

corpus

2	␣ n e w
2	␣ r e n e w
1	s e t
1	␣ r e s e t

vocabulary

␣, e, n, r, s, t, w

Merge n e to ne (count 4 = 2 new + 2 renew)

corpus

2	␣ ne w
2	␣ r e ne w
1	s e t
1	␣ r e s e t

vocabulary

␣, e, n, r, s, t, w, ne

BPE token learner

corpus		vocabulary
2	`␣ ne w`	`␣, e, n, r, s, t, w, ne`
2	`␣ r e ne w`	
1	`s e t`	
1	`␣ r e s e t`	

Merge `ne w` to `new` (count 4)

corpus		vocabulary
2	`␣ new`	`␣, e, n, r, s, t, w, ne, new`
2	`␣ r e new`	
1	`s e t`	
1	`␣ r e s e t`	

BPE token learner

corpus		vocabulary
2	␣ n e w	␣, e, n, r, s, t, w, ne, new
2	␣ r e new	
1	s e t	
1	␣ r e s e t	

Merge ␣ r to ␣r (count 3) and ␣r e to ␣re (count 3)

corpus		vocabulary
2	␣ new	␣, e, n, r, s, t, w, ne, new, ␣r, ␣re
2	␣re new	
1	s e t	
1	␣re s e t	

System has learned prefix re- !

BPE

The next merges are:

merge	**current vocabulary**
(␣, new)	␣, e, n, r, s, t, w, ne, new, ␣r, ␣re, ␣new
(␣re, new)	␣, e, n, r, s, t, w, ne, new, ␣r, ␣re, ␣new, ␣renew
(s, e)	␣, e, n, r, s, t, w, ne, new, ␣r, ␣re, ␣new, ␣renew, se
(se, t)	␣, e, n, r, s, t, w, ne, new, ␣r, ␣re, ␣new, ␣renew, se, set

BPE **encoder** algorithm

Tokenize a test sentence: run each merge learned from the training data:

- Greedily, in the order we learned them
- (test frequencies don't play a role)

First: segment each test word into characters

Then run rules: (1) merge every n e to ne, (2) merge ne w to new, (3) ␣r, (4) ␣re etc.

Result:

- Recreates training set words
- But also learns subwords like ␣re that might appear in new words like rearrange

BPE and Unicode

We run BPE on large Unicode corpora, with vocabulary sizes of 50,000 to 200,000

On individual bytes of UTF-8-encoded text

- Not on Unicode characters
- BPE rediscovers 2-byte and common 3-byte UTF-8 sequences
- Only 256 possible values of a byte, so no unknown tokens
- (BPE might learn a few illegal UTF-8 sequences across character boundaries, but these can be filtered)

Visualizing GPT4o tokens

Tat Dat Duong's Tiktokenizer visualizer

`Anyhow,·she's·seen·Jane's·224123·flowers·anyhow!`

Tokens: 11865, 8923, 11, 31211, 6177, 23919, 885, 220, 19427, 7633, 18887, 147065, 0

Most words are tokens, w/initial space

Clitics like 's

- Are segmented off Jane
- But part of frequent words like she's

Numbers segmented into chunks of 3 digits

Anyhow and ·anyhow are segmented differently

Some of this is from preprocessing

- regular expressions for chunking digits, stripping clitics

Tokenizing across languages

- Even though BPE tokenizers are multilingual
- LLM training data is still vastly dominated by English
  - Most BPE tokens used for English, leaving less for other languages
  - Words in other languages are often split up

Tokenization is better in English

Tat Dat Duong's [Tiktokenizer]() visualizer on GPT4o

A recipe sentence in two languages

English: 18 tokens; no words are split into multiple tokens):

In·a·deep·bowl,·mix·the·orange·juice·with·the·sugar,·g
inger,·and·nutmeg.

Spanish: 33 tokens; 6/16 words are split

En·un·recipiente·hondo,·mezclar·el·jugo·de·naranja·con
·el·azúcar,·jengibre,·y·nuez·moscada.

Byte Pair Encoding

Words and Tokens

Corpora

Corpora

Words don't appear out of nowhere!

A text is produced by

- a specific writer(s),
- at a specific time,
- in a specific variety,
- of a specific language,
- for a specific function.

Corpora vary along dimensions like

Language: 7097 languages in the world

It's important to test algorithms on multiple languages

What may work for one may not work for another

Corpora vary along dimensions like

Variety, like African American English varieties

- AAE Twitter posts might include forms like "*iont*" *(I don't)*

Genre: newswire, fiction, scientific articles, Wikipedia

Author Demographics: writer's age, gender, ethnicity, socio-economic status

Code Switching

Speakers use multiple languages in the same utterance

This is very common around the world

Especially in spoken language and related genres like texting and social media

Code Switching: Spanish/English

Por primera vez veo a @username actually being hateful! It was beautiful:)

[For the first time I get to see @username actually being hateful! it was beautiful:)]

Code Switching: Hindi/English

dost tha or ra- hega ... dont wory ... but dherya rakhe

[“he was and will remain a friend ... don’t worry ... but have faith”]

Corpus **datasheets**

Gebru et al (2020), Bender and Friedman (2018)

Motivation:

- Why was the corpus collected?
- By whom?
- Who funded it?

Situation: In what situation was the text written?

Collection process: How was it sampled? Was there consent? Pre-processing?

+Annotation process, variety, demographics, etc.

Words and Tokens

Corpora

Words and Tokens

Regular Expressions

Regular expressions are used everywhere

- A formal language for specifying text strings
- Part of every text processing task
  - Useful pre-processing or text formatting step, for example for BPE pre-tokenization
- Also necessary for data analysis of text
- A widely used tool in industry and academics

Regular expressions

We can use regular expressions to search for a pattern in a string

For example, the **Python** function

```
re.search(pattern,string)
```

scans through the string and returns the first match inside it for the pattern

Python syntax

We'll show regex on raw strings with double quotes:

```
r"regex"
```

Raw strings treat backslashes as literal characters

Many regex patterns use backslashes.

Why raw strings?

- Regex and Python both use backslash `"\"` for special characters. If you don't use raw notation you would have to type extra backslashes!
  - `"\\d+"` to search for 1 or more digits
  - `"\n"` in Python means the "newline" character, not a "slash" followed by an "n". Need `"\\n"` for two characters.
- Instead: use Python's **raw string notation** for regex:
  - `r"[tT]he"`
  - `r"\d+"` matches one or more digits
    - instead of `\\d+`

Regular expressions

The pattern

```
r"Buttercup"
```

matches the substring Buttercup in any string, like the string

```
I’m called little Buttercup
```

Regular Expressions: Disjunctions

Letters inside square brackets []

Pattern	Matches
`r"[mM]ary"`	`Mary` or `mary`
`r"[1234567890]"`	Any one digit

Ranges using the dash `[A-Z]`

Pattern	Matches	
`r"[A-Z]"`	An upper case letter	`Drenched Blossoms`
`r"[a-z]"`	A lower case letter	`my beans were impatient`
`r"[0-9]"`	A single digit	`Chapter 1: Down the Rabbit Hole`

Regular Expressions: Negation in Disjunction

Carat as first character in [] negates the list

- Note: Carat means negation only when it's first in []
- Special characters (., *, +, ?) lose their special meaning inside []

Pattern	Matches	Examples
`r"[^A-Z]"`	Not upper case	Oyfn pripetchik
`r"[^Ss]"`	Neither 'S' nor 's'	I have no exquisite reason"
`r"[^.]"`	Not a period	Our resident Djinn
`r"[e^]"`	Either e or ^	Look up ^ now

Kleene star and Kleene plus

```
baa
baaa
baaaa …
```

Kleene star * (0 or more of previous characters)

Kleene plus + (1 or more of previous character)

```
r"baaa*"     baa baaa baaaa …
r"baa+"
```

Stephen C Kleene

Wildcard

The period means "any character"

`r"."` matches anything

`r".*"` matches any sequence of 0 or more of anything

Regular Expressions: Anchors ^ $

Pattern	Matches
`r"^[A-Z]"`	<u>P</u>alo Alto
`r"\.$"`	The end<u>.</u>
`r".$"`	The end<u>?</u> The end<u>!</u>

Regular Expressions: More Disjunction

Groundhog is another name for woodchuck!

The pipe symbol | for disjunction

Pattern	Matches
r"groundhog|woodchuck"	woodchuck
r"yours|mine"	yours
r"a|b|c"	= [abc]
r"[gG]roundhog|[Ww]oodchuck"	Woodchuck

Regular Expressions: Convenient aliases

Pattern	Expansion	Matches	Examples
`r"\d"`	`[0-9]`	Any digit	`Fahreneit 451`
`r"\D"`	`[^0-9]`	Any non-digit	`Blue Moon`
`r"\w"`	`[a-zA-Z0-9_]`	Any alphanumeric or _	`Daiyu`
`r"\W"`	`[^\w]`	Not alphanumeric or _	`Look!`
`r"\s"`	`[ \r\t\n\f]`	Whitespace (space, tab)	`Look up`
`r"\S"`	`[^\s]`	Not whitespace	`Look up`

The iterative process of writing regex's

Find me all instances of the word “the” in a text.

`the`

Misses capitalized examples

`[tT]he`

Incorrectly returns `other` or `Theology`

`\W[tT]he\W`

False positives and false negatives

The process we just went through was based on fixing two kinds of errors:

1. Not matching things that we should have matched (The)

False negatives

2. Matching strings that we should not have matched (there, then, other)

False positives

Characterizing work on NLP

In NLP we are always dealing with these kinds of errors.

Reducing the error rate for an application often involves two antagonistic efforts:

- Increasing coverage (or *recall*) (minimizing false negatives).
- Increasing accuracy (or *precision*) (minimizing false positives)

Regular expressions play a surprisingly large role

Widely used in both academics and industry

1. Part of most text processing tasks, even for big neural language model pipelines
   - including text formatting and pre-processing
2. Very useful for data analysis of any text data

Words and Tokens

Regular Expressions

Substitutions, Capture Groups, and Lookahead

Regex Substitutions in Python

To change every instance of cherry to apricot in string:

```
re.sub(r"cherry", r"apricot", string)
```

Upper case all examples of a name:

```
re.sub(r"janet", r"Janet", string)
```

Substitutions often need capture groups

Task: Change:

US format dates (mm/dd/yyyy)   to

EU format dates (dd-mm-yyyy)

Pattern to match US format:

```
r"\d{2}/\d{2}/\d{4}"
```

How to specify in the *replacement* that we want to swap the date and month values?

Capture group

Use parentheses to capture (*store*) the values that we matched in the search,

Groups have numbers

In the replacement ("repl"), we refer back to that group with a number command.

Capture group

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

Parens (and) around the two month digits, the two day digits, and the four year digits,

This stores

- the first 2 digits in group 1,
- the second 2 digits in group 2,
- final digits in group 3.

Then in the *repl* string,

- \1, \2, and \3, refer to the 1st, 2nd, and 3rd registers.

That regex will

map

The date is 10/15/2011

to

    The date is 15-10-2011

But suppose we don't want to capture?

Parentheses have a double function: grouping terms, and capturing

Non-capturing groups: add a ?: after paren:

`r"(?:some|a few) (people|cats) like some \1/"`

matches

- `some cats like some cats`

but not

- `some cats like some some`

Lookahead assertions

`(?= pattern)` is true if pattern matches, but is **zero-width; doesn't advance character pointer**

`(?! pattern)` true if a pattern does not match

How to capture the first word on the line, but only if it doesn't start with the letter T:

```
r"^(?![tT])(\w+)\b"
```

Simple Application: ELIZA

Early NLP system that imitated a Rogerian psychotherapist

- Joseph Weizenbaum, 1966.

Uses pattern matching to match, e.g.,:

- `"I need X"`

and translates them into, e.g.

- `"What would it mean to you if you got X?`

Simple Application: ELIZA

Men are all alike.
IN WHAT WAY

They're always bugging us about something or other. CAN YOU THINK OF A SPECIFIC EXAMPLE

Well, my boyfriend made me come here.
YOUR BOYFRIEND MADE YOU COME HERE

He says I'm depressed much of the time.
I AM SORRY TO HEAR YOU ARE DEPRESSED

How ELIZA works

```
re.sub(r".* I’M (depressed|sad) .*",r"I AM SORRY TO HEAR YOU ARE \1",input)

re.sub(r".* I AM (depressed|sad) .*",r"WHY DO YOU THINK YOU ARE \1",input)


re.sub(r".* all .*",r"IN WHAT WAY?",input)

re.sub(r".* always .*",r"CAN YOU THINK OF A SPECIFIC EXAMPLE?",input)
```

Substitutions, Capture Groups, and Lookahead

Words and Tokens

Pretokenization for BPE

Pretokenization for BPE

The GPE-2 rules for pretokenization:

- Count clitics like 't and 'm as tokens
- Pull off punctuation and numbers

We do this with one regex:

```
r"'s|'t|'re|'ve|'m|'ll|'d| ?\p{L}+| ?\p{N}+| ?[^\s\p{L}\p{N}]+|\s+(?
```

The Python regex (as opposed to re) library

Special \p and \P operators:

\p{L} matches any Unicode letter,

\P{L} matches any non-letter,

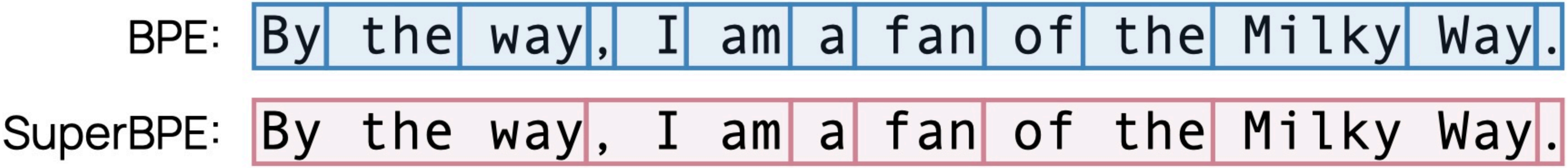
\p{N} matches any number,

\P{N} matches any non-number.

```
>>> 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', '?']
>>>
```

Pretokenization could also allow multiple words!

SUPERBPE runs a second stage of BPE allowing merges across spaces and punctuation.



Liu et al 2025

Words and Tokens

Pretokenization for BPE

Words and Tokens

Rule-based tokenization and Simple Unix tools

Rule-based tokenization

Although subword tokenization is the norm

Sometimes we need particular tokens

Like for parsing, where the parser needs grammatical words, or social science

Issues for rule-based tokenization

Mostly but not always remove punctuation:
- m.p.h., Ph.D., AT&T, cap'n
- prices ($45.55)
- dates (01/02/06)
- URLs (http://www.stanford.edu)
- hashtags (#nlproc)
- email addresses (someone@cs.colorado.edu)

Numbers are tokenized differently across languages
- English 555,500.50 = French 555 500,50

Multiword expressions (MWE)?
- New York, rock 'n' roll

Penn Treebank Tokenization Standard

Input: `"The San Francisco-based restaurant," they said,`
`"doesn’t charge $10".`

Output: `"␣The␣San␣Francisco-based␣restaurant␣,␣"␣they␣said␣,␣`
`"␣does␣n’t␣charge␣$␣10␣"␣.`

Tokenization in NLTK

Bird, Loper and Klein (2009), *Natural Language Processing with Python*. O'Reilly

```
>>> 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, 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', '...']
```

Sentence Segmentation

!, ? mostly unambiguous but **period** "." is very ambiguous

- Sentence boundary
- Abbreviations like Inc. or Dr.
- Numbers like .02% or 4.3

Common algorithm: Tokenize first: use rules or ML to classify a period as either (a) part of the word or (b) a sentence-boundary.

- An abbreviation dictionary can help

Sentence segmentation can then often be done by rules based on this tokenization.

Space-based tokenization

A very simple way to tokenize

- For languages that use space characters between words
  - Arabic, Cyrillic, Greek, Latin, etc., based writing systems
- Segment off a token between instances of spaces

Unix tools for space-based tokenization

- The "tr" command
- Inspired by Ken Church's UNIX for Poets
- Given a text file, output the word tokens and their frequencies

Simple Tokenization in UNIX

(Inspired by Ken Church's UNIX for Poets.)

Given a text file, output the word tokens and their frequencies

```
tr -sc 'A-Za-z' '\n' < shakes.txt
     | sort
     | uniq -c
```

Change all non-alpha to newlines

Sort in alphabetical order

Merge and count each type

```
1945 A
  72 AARON
  19 ABBESS
   5 ABBOT
 ... ...
```

```
25 Aaron
 6 Abate
 1 Abates
 5 Abbess
 6 Abbey
 3 Abbot
.... …
```

The first step: tokenizing

```
tr -sc 'A-Za-z' '\n' < shakes.txt | head
```

```
THE
SONNETS
by
William
Shakespeare
From
fairest
creatures
We
...
```

The second step: sorting

```
tr -sc 'A-Za-z' '\n' < shakes.txt | sort | head
```

```
A
A
A
A
A
A
A
A
A
...
```

More counting

Merging upper and lower case

```
tr 'A-Z' 'a-z' < shakes.txt | tr -sc 'A-Za-z' '\n' | sort | uniq -c
```

Sorting the counts

```
tr 'A-Z' 'a-z' < shakes.txt | tr -sc 'A-Za-z' '\n' | sort | uniq -c | sort -n -r
```

```
23243 the
22225 i
18618 and
16339 to
15687 of
12780 a
12163 you
10839 my
10005 in
8954  d
```

What happened here?

Words and Tokens

Rule-based tokenization and Simple Unix tools