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

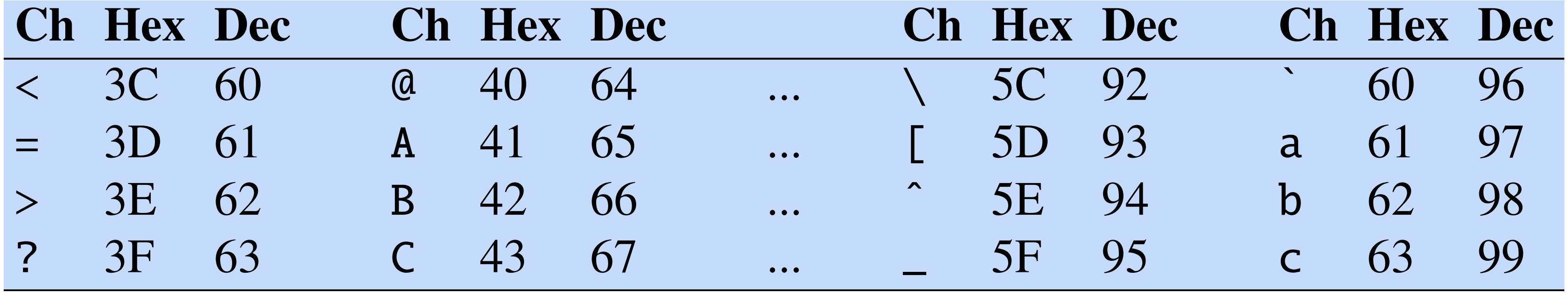
1960s American Standard Code for Information Interchange

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

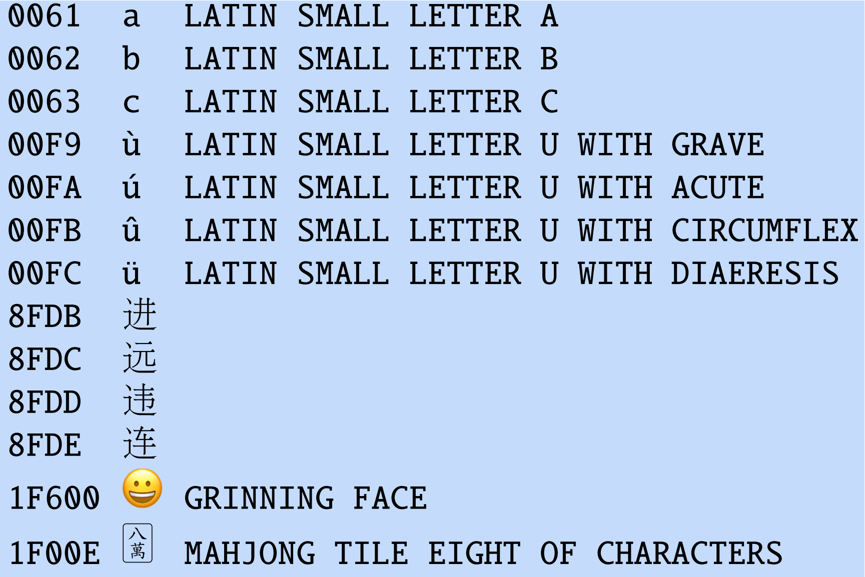
Written in hex, with prefix "U+"

- 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

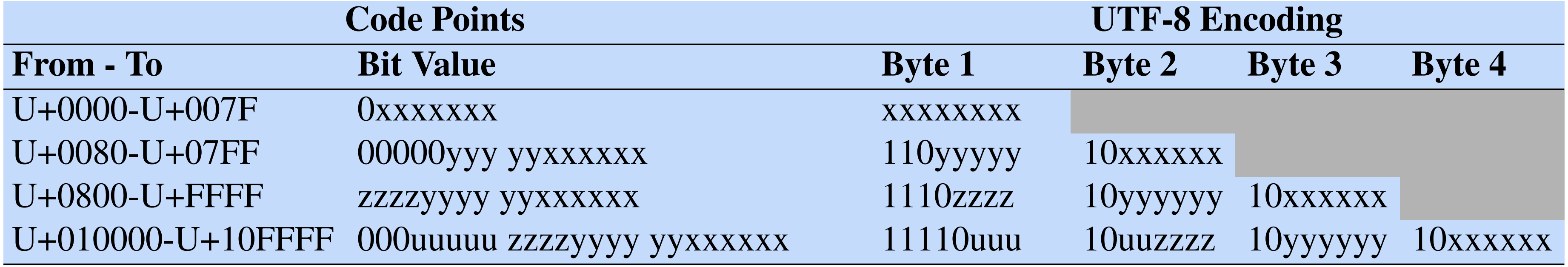
UTF-8 encoding of `hello` is :

- 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

Words and Tokens

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]

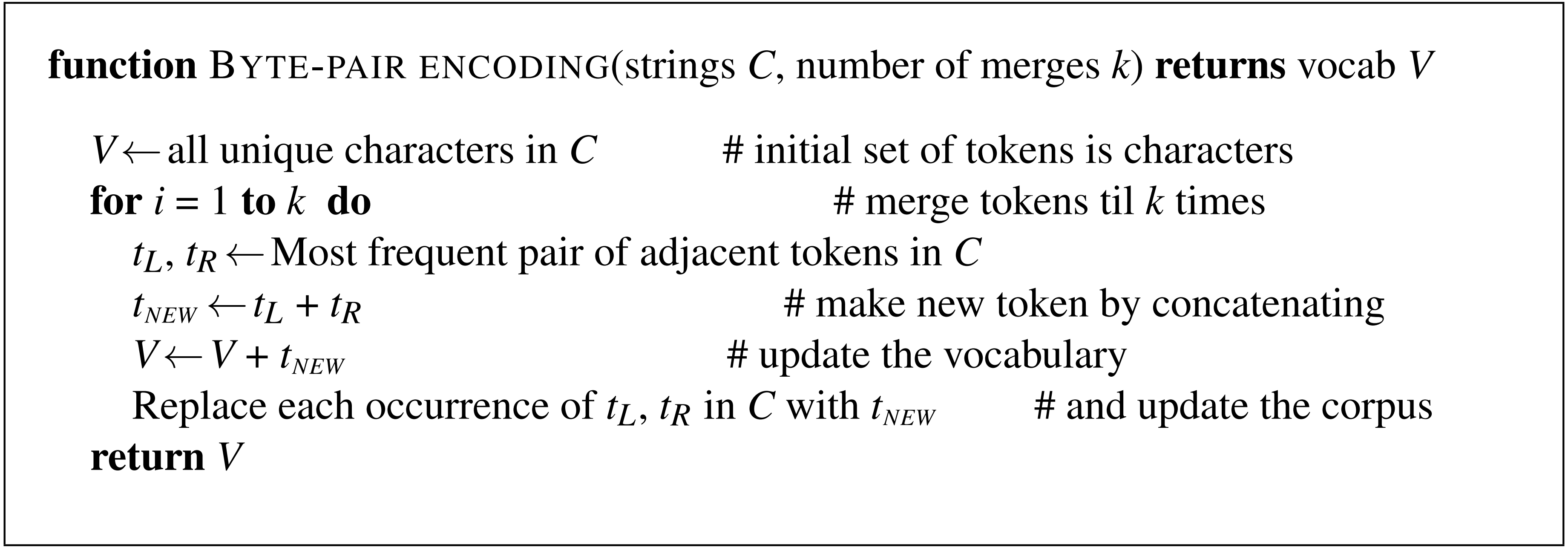
Corpus

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

2	␣ new
2	␣ r e new
1	s e t
1	␣ r e s e t

vocabulary

␣, e, n, r, s, t, w, ne, new

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

corpus

2	␣ new
2	␣re new
1	s e t
1	␣re s e t

vocabulary

␣, e, n, r, s, t, w, ne, new, ␣r, ␣re

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

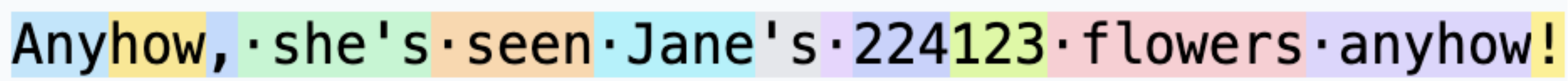
We run BPE on large Unicode corpora, with vocabulary sizes of 50,000 to 200,000

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

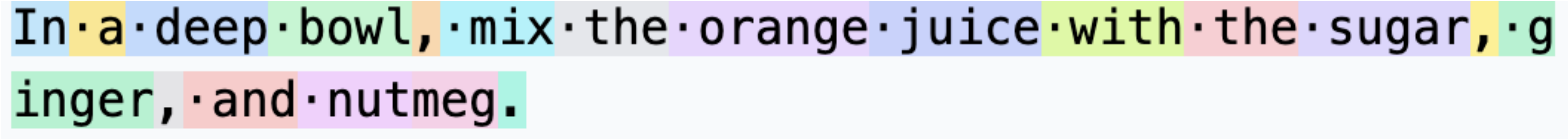
- Most BPE tokens used for English, leaving less for other languages
- Words in other languages are often split up

Tokenization is better in English

Tat Dat Duong's Tiktokenizer visualizer on GPT4o

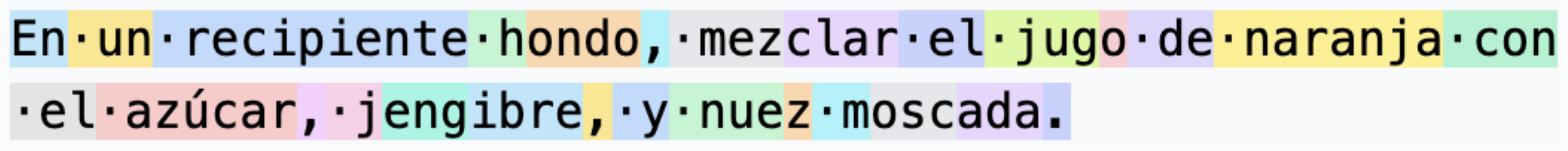
A recipe sentence in two languages

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.

Words and Tokens

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