**Title**
Verkettete Textualität [Concatenative Textuality]



**ISBN**
978-3-7518-9020-5

**Author**
Shoemaker, Tyler

**Publication Date**
2024-10-01

**Concantenative Textuality**[1]

Tyler Shoemaker, UC Davis

In recent years, discourse about artificial intelligence (AI) has seen one claim come and go with near orbital periodicity. *AI models*, it runs, *invent languages of their own*. The claim usually pertains to a class of neural networks trained to predict sequences of text on the basis of gigantic volumes of data. Researchers and developers use these networks, or 'large language models' (LLMs), to perform tasks spanning text generation and summarization, machine translation, classification, and more. But training LLMs on heaps of data appears to have gone beyond producing general models of communication. Language modeling at the scale of millions of documents and billions, if not trillions of parameters now invents 'new' natural languages—ones native to AI models themselves.

At least, this is the claim. An early version of it began circulating in 2016 after tech reporters picked up a blog post from Google AI (Schuster, Johnson, and Thorat 2016). The post summarized how the latest Google Translate model, newly rebuilt with a neural network, could render translations between two languages without explicit training. If the model was trained to translate English ↔ Korean and English ↔ Japanese, Google researchers showed that it could also generalize to Korean ↔ Japanese, even though its training did not include this third pairing. Researchers attributed this ability to an "interlingua" in their network; tech reporting was characteristically uncritical in its adoption of that term. "Google's AI translation tool seems to have invented its own secret internal language," read one TechCrunch headline (Coldewey 2016). *Wired* quickly followed suit the next day, announcing, "Google's AI just created its own universal 'language' " (Burgess 2016).

Exactly what comprised this interlingua—its vocabulary and syntax, for instance—was unclear. Researchers pointed to a scatter plot, its clusters shaded to mark semantic equivalence across languages; this, apparently, is what Google

---

1 English translation of "Verkettete Textualität," *Quellcodekritik: Zur Philologie von Algorithmen*, eds. Hannes Bajohr and Markus Krajewski (Berlin: August Verlag 2024).

Translate's interlingua looks like. But later versions of the AI language claim have fixated on more human-readable examples. When, in 2017, Facebook supposedly shut down its negotiation chatbots after their conversations had deviated from English into terse, nonsensical utterances, numerous outlets recorded these exchanges verbatim. "What an AI's Non-Human Language Actually Looks Like," promised *The Atlantic*: "you i i i everything else . . . . . . . . . . . . . ." (LaFrance 2017). Most recently, computer scientists at the University of Texas at Austin announced in a viral Twitter thread that DALL-E 2, OpenAI's popular text-to-image generator, also has a "secret language" (Daras 2022; Daras and Dimakis 2022). They discovered as much by sending transcriptions of the model's own text-like images back to DALL-E 2 as new prompts. " 'Apoploe vesrreaitais' means birds. 'Contarra ccetnxniams luryca tanniounons' means bugs or pests." Images generated by DALL-E 2 in response to these phrases, showing birds on twigs or food in stoneware (for "Vicootes."), were meant to back up the scientists' claims.

In one sense, the relevance of such claims to reading source code is thematic. A cryptanalytic leitmotif runs throughout. AI language, "secret" or "hidden," has been cracked like a code—and with it, perhaps, the strange terrain of non-human intelligence. Outside popular discourse about AI there is a modicum of truth to this. Research in systems security and model interpretability has described the effects of adversarial "triggers" on LLMs, sequences of nonsense text like "zoning tapping fiennes" and "b 617 matrices dhabi ein wm" (Wallace et al. 2019; Singla et al. 2022; see also Millière 2022). When added to model input, such triggers force LLMs into errancy. Some sequences flip sentiment appraisals into polar inversion; others coerce reading comprehension systems into returning the same answer repeatedly; a third group disrupts entailment models; and a fourth causes text generators to regurgitate hate speech. Even here, outside the hype about AI language, there is a temptation to slip into talk of codebreaking with triggers. It is as if these sequences are evidence of some obscure source code in LLMs. They seem to suggest a deeper programmatic logic that drives models, one that might be hacked with only a few characters.

Beyond thematics, the actual practice of source code criticism, as exemplified in the work of Mark Marino as well as Winnie Soon and Geoff Cox, can do much to assess the linguistic status of both triggers and sequences like "Vicootes" (see Marino 2020; Soon and Cox 2021). For LLMs do not learn from words. They learn from *tokens*. And the 'tokenization' algorithms that transform words into tokens are as integral to model performance as are system architecture and training data. This is especially the case with so-called 'subword' tokens. Quasi-morphemic units like "##riated," "wa," and "##γ" are the very stuff of LLMs; but instead of language, a better way to think about these sequences would be textuality.[2] This is my counter to the AI language claim. While linguists and AI ethicists have made vital critiques of AI language using the idea of "communicative intent" (Bender and Koller 2020; Bender et al. 2021), the presence or lack of this intent among LLMs cannot account for the full reach of their material-semiotic operations. As the newest tokenization algorithms make particularly clear, LLMs are media systems, capable of signification beyond intent (see Liu 2010; Lazzarato 2014); source code criticism equips us with a framework to approach them as such.

My suggestion that we take this approach follows Michael Gavin's work on the unique textual objects of language modeling (2019). For Gavin, even the simplest procedures of mathematizing text, like compiling word frequency lists or padding corpus documents with zeros, catalyze a shift in registers to a new form of textuality, albeit one that does not quite converge with the meaning of the term in the way literature scholars might understand it. Subword tokenization is another such procedure of the kind Gavin describes, and the process itself offers a highly effective vantage from which to survey what I call the "concatenative" textuality of LLMs. The work of this paper is to show as such through an account of one method for producing subwords: the very algorithm Google researchers employed to build their highly generalizable translation model.

*

Tokenization is the process of chunking running text into countable units. These units, or tokens, are central to natural language processing (NLP), for they are

---

2 Unless indicated otherwise, I use tokens from Hugging Face's base uncased BERT (Bidirectional Encoder Representations from Transformers) model, from Devlin et al. (2019); see https://huggingface.co/bert-base-uncased.

what quite literally counts in the arithmetic of stochastic semantics. Their use has been all but constant since the earliest language models, running back to the work of Andrei Markov as well as Claude Shannon's "Mathematical Theory of Communication." To "approach a language" with statistical sampling, Shannon mostly used character-based tokens of varying lengths (Shannon and Weaver 1998, 43). Tokens comprised of two characters, or bigrams, served as the basis for his initial models; but models he built from trigrams, tokens with three characters each, more closely "approach[ed] a language": "IN NO IST LAT WHEY," reads output from the latter. "CRATICT FROURE BIRS GROCID PONDENOME." If this is an improvement, it is not because Shannon refined his models with linguistic principles in mind. Importantly, he, like many other computationalists, developed his tokenization methods ad hoc, adapting them on the fly to fit a changing problem space (Golumbia 2009). Most often a token is just a word—if by 'word' one means any sequence of characters set between two white spaces.

From the 1950s on, this definition of token was predominant, though the late 1980s and early 90s saw occasional departures from white space tokenization. Researchers in information retrieval experimented with "sublexical" tokens, piecing together words from smaller, combinatoric units in an attempt to reduce the vocabulary size of their search systems (Kimbrell 1988; Schütze 1992). These scattered experiments appear not to have had a wide influence at the time. However, the idea resurfaced as a key reference in the 2010s, when research efforts at both Google and Facebook turned toward the problem of unseen data, or so-called 'out-of-vocabulary' (OOV) items (Mikolov et al. 2011; Schuster and Nakajima 2012; Bojanowski et al. 2017). As engineers in the first wave of mechanical translation were to learn, models are only capable of processing tokens that were included in their original training vocabularies, and this means rare or nonce words are particularly troublesome during real-world deployment. Techniques for handling OOV items are legion, but at Google and Facebook, the approach centered on rethinking the static nature of training vocabularies. Researchers theorized that models could better handle unseen data if their training equipped them with the atomic building blocks of words, to be reassembled or newly combined as needed. Models, in other words, needed to 'learn' how to spell.

Training them to do so involved more closely binding tokenization to the

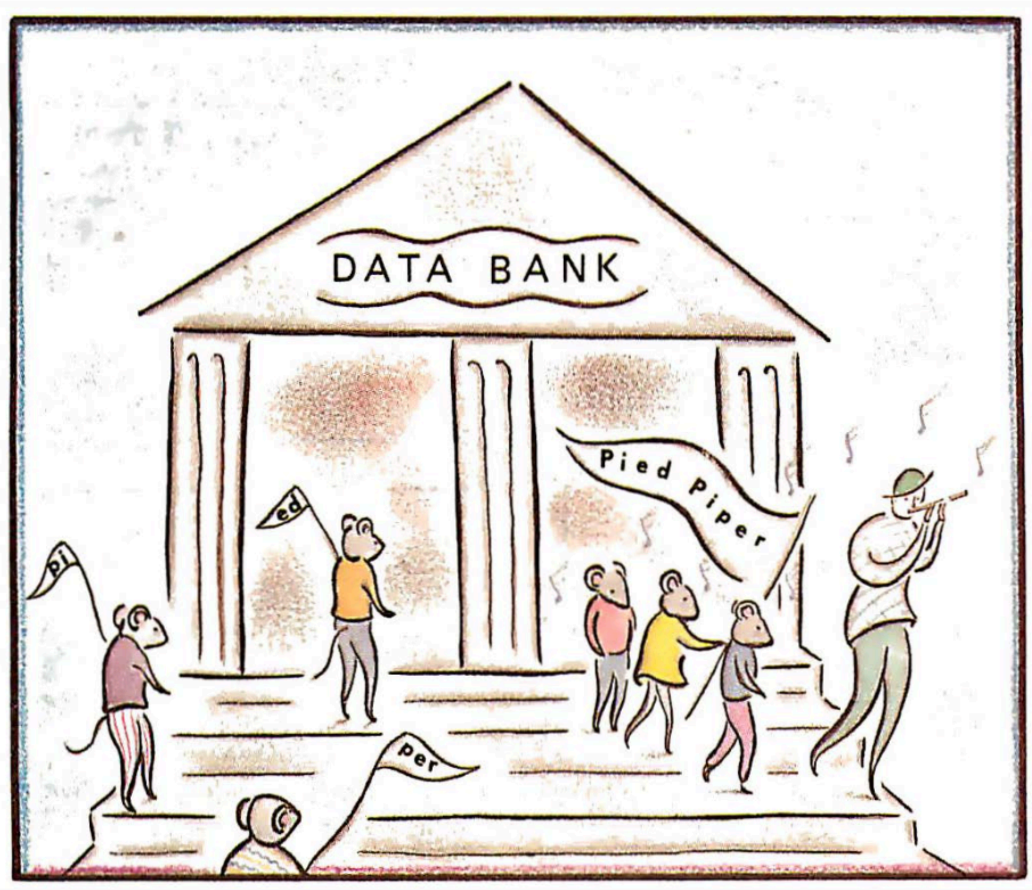


Figure 1: Searching with sublexical tokens, according to *Byte* magazine (Kimbrell 1988).

workings of LLMs. In a departure from other methods of preparing text for language modeling, models like BERT or GPT-3 come with built-in tokenizers, which automatically process raw data during training and later downstream tasks; model builders need give little direction for how this process unfolds, beyond specifying (if desired) a target size for the resultant vocabularies. The rest remains in a black box. The result: models that can explain jokes, or give extended answers to writing prompts, and subwords that read, “pre,” “##ters,” “##þ,” “##!.”

Many NLP libraries base their tokenizers on a “byte pair encoding” (BPE) algorithm devised by Google researchers for their 2016 Google Translate model, the very one that attracted media attention for its supposed interlingua (Sennrich, Haddow, and Birch 2016). The algorithm, which I render below in Python, is remarkably simple. It contains only two steps. First, it counts character bigrams in a corpus. Then, on the basis of those counts, it gradually concatenates those bigrams into longer sequences to approximate the original data.

**Step One**

```
# Tokenization begins with a pre-generated dictionary of word
# counts. The counts are generated from a pass through a white
# space tokenizer, which also inserts white space between each
# character in every word. Another preparatory step will have
# appended corpus documents with special tags to mark sequence
# boundaries; the dictionary accounts for these tags as well.
# BERT uses [CLS] and [SEP] to annotate, respectively, the start
# and end of longer sequences, like sentences; another tokenizer
# uses tildes or underscores to represent leading/trailing
# spaces; the 2016 version only marks the ends of words, with
# </w>.
vocab = {
    "l o w </w>": 5,
    "l o w e r </w>" 2,
    "n e w e s t </w>": 6,
    "w i d e s t </w>": 3
}

import re
import collections

# When initialized, the tokenizer calls a function that loads
# the count data...
def get_stats(vocab):
    # ...and creates a new, empty dictionary.
    pairs = collections.defaultdict(int)
```

```
    # It then iterates through all words in its input,
    # splitting each into a list of characters.
    for word, freq in vocab.items():
        symbols = word.split()

        # For every one of these split words the function
        # generates a set of sequential bigrams and adds them
        # to its new dictionary. It gives each the corresponding
        # count of the word from which those bigrams are
        # derived. If a bigram is already in the dictionary
        # because of a previous word, the function updates that
        # bigram's count by adding the new count to the old one.
        for i in range(len(symbols) - 1):
            pairs[symbols[i], symbols[i + 1]] += 1

    # When it has finished iterating through all words in its
    # input, the function then returns the new dictionary.
    return pairs
```

---

**Step Two**

---

```
# With the new counts generated, the tokenizer selects a bigram.
# This bigram is either the most frequent one in the counts, or
# it is the one that most increases the overall likelihood of a
# simple, probabilistic language model fitted to the corpus.
# Whatever the metric, once the tokenizer selects a bigram it
# calls a second function, sending in the selected bigram and
# the original dictionary of split words.
def merge_vocab(pair, v_in):
    # The function creates a new dictionary...
```

```
    v_out = {}

    # ...and turns the bigram into a searchable string.
    bigram = re.escape(' '.join(pair))
    p = re.compile(r'(?<!\S)' + bigram + r'(?!\S)')

    # It then searches for this bigram in the input dictionary.
    for word in v_in:
        # When it finds a match in the input, the function
        # concatenates the left and right components of this
        # bigram into a single token. For example, the input
        # word "l o w e r </w>," with bigram "er," becomes
        # "l o w er </w>." If the concatenated bigram is in
        # the middle of a word, some versions of the
        # tokenization algorithm, like the one BERT uses, will
        # add a new tag to mark this ("l o w ##er </w>").
        w_out = p.sub(''.join(pair), word)

        # With the bigram made, it is placed in the new
        # dictionary, and the frequency count of the input word
        # is associated with it.
        v_out[w_out] = v_in[word]

    # Once the function finds all possible bigram matches in its
    # input, it returns the new dictionary. Step two is then
    # complete.
    return v_out

# A single run through these two steps does not accomplish much,
# but the tokenizer will repeat this process multiple (often
# very many) times.
num_merges = 5
```

```
# After it finds and concatenates all instances of a selected
# bigram, it returns to the first step and counts every bigram
# from the new, modified version of the word counts.
for _ in range(num_merges):
    # This time, the resultant counts will reflect the sequences
    # that include the newly made tokens, rather than each of
    # their separate characters.
    pairs = get_stats(vocab)

    # These new counts will then influence which bigram
    # components should be selected and concatenated next...,
    best = max(pairs, key = pairs.get)

    # ...which will in turn influence subsequent counts, further
    # bigram selections, and so on. Gradually, recognizable words
    # will begin to re-form out of the character sequences. A few
    # hundred iterations after merging "er," the tokenizer might
    # select "ow," to build "l ow er </w>"---as well as
    # "p i l l ow </w>," "ow </w>," "ow n er </w>," and more. The
    # process continues, merging sequences within words, until
    # one of two conditions is met: either the tokenizer runs out
    # of bigrams to create (in which case its output mirrors its
    # input), or, more commonly, it reaches a predetermined
    # number of subwords; any remaining bigrams are left as is.
    # With either condition met, the process finishes and the
    # tokenized corpus data is ready for modeling. Some words
    # are reassembled, others are not, but in both cases a model
    # will be able to represent them during training and
    # downstream tasks.
    vocab = merge_vocab(best, vocab)
```

| **Original Tokens** | **Merged Output** |
|---|---|
| l o w </w> | low </w> |
| l o w e r </w> | low e r </w> |
| n e w e s t </w> | n e w est</w> |
| w i d e s t </w> | w i d est</w> |

Table 1: Output from putting the sample vocabulary through five iterations of the 2016 BPE tokenizer

Taken together, the above code blocks comprise a rather pedestrian counter to the AI language claim. That models like DALL-E 2 respond to sequences like "Apoploe vesrreaitais" has far less to do with invention than *concatenation*, a process by which LLMs string together fuzzy, bootstrapped representations of their training data "without focusing on semantics" (Schuster and Nakajima 2012, 5150). Again: how do you learn to use a word you've never seen before? You start by spelling it out: "A pop loe v es r re ait ais."[3] But fixed-length sequences of the kind Shannon modeled do not provide adequate material to do so. Key to the changeover subwords bring is variability. They are the result of a pervading indifference about what, at base, constitutes the atomic elements of text data. Subwords may be single characters from several script systems, prefix-like bigrams, chunks that isomorphically resemble morphemes, or even—most strangely of all—entire words. Whatever the form, for every such sequence a model will produce an embedding. Subwords in this sense enact a profound flattening effect on text. With them, a word is no different in nature than a single character.

*

While such a flattening may trouble the extent to which subwords count as language, it remains an open question whether, and how, they are to be read. Again, my interests here are in the unique textuality of LLMs and their subwords. If there

[3] These tokens are from the OpenAI tokenizer tool, which uses a version of the GPT-3 tokenizer implemented by Walton (2020); see https://beta.openai.com/tokenizer.

is indeed a change or challenge to the way one reads the latter, the limit case for this difference must be those instances where a tokenized character string appears no different from a word. What, in other words, is the difference between "lower," the word, and "lower," the subword? In one sense: none—and yet. The difference between these strings is, as Marcel Duchamp might say, infra-thin [*inframince*], but a difference, perhaps, it is.

I will approach this difference from two, somewhat abbreviated routes. The first is orthography. A reader's intuition that subwords might be analogized in terms of spelling or punctuation would be well placed; the various conventions typographers have developed to divide words across a text block are further points of comparison.[4] But there is also the special orthography of what Pip Thornton calls "language in the age of algorithmic reproduction" (Thornton 2018). From "program, ##matic, token, ##ization" there is a direct connection with the broader textual condition of linguistic capitalism, where services like autocomplete nudge writing into predictable (read: more economically exploitable) expressions (see Cayley 2013; Kaplan 2014). Among the buyers of Google Ads, writes Frédéric Kaplan, there is "not much bidding on misspelled words," a point about which Jeff Dean, the lead at Google AI, would have to concede (2014, 59). Orthographic normalization has been the company's business model from its earliest days. Recounts Dean, "In 2001, some colleagues sitting just a few feet away from me at Google realized they could use an obscure technique called machine learning to correct misspelled Search [sic] queries" (Dean 2021). Many of today's LLMs—and the subwords they require—descend from this very moment.

Put another way, in the orthography of linguistic capitalism, a token (textual unit) is always also a token (unit of currency). Thornton's "{poem}.py" project, which tallies up the cost of canonical poems according to their contents' Google AdWords prices, makes this particularly clear (Thornton 2016). But, turn over the thermal-printed receipt for Wordsworth's "Daffodils," and you will

---

4 In the English language context, one of the best-known style guides, *Hart's Rules*, formulates word divisions in predictive terms: "The principle is that the part of the word left at the end of a line should suggest the part commencing the next line" (1921, 54).

also find a slate of brand new modeling infrastructures, which have appeared during the widespread adoption of LLMs. Consider OpenAI, which fixes its prices directly on subword tokens. With the embeddings for the latest models in its GPT series kept locked away, the company's services are instead accessible through a paid API. To use its base models, prices range from $0.0008–0.06 per batch of 1,000 tokens; custom models, trained on a dataset of one's choosing, will cost as much as $0.12 per batch. A pricing guide tells us, "You can think of tokens as pieces of words, where 1,000 tokens is about 750 words"—a convenient mismatch, for customers will end up paying more for OpenAI's modeling services than their word counts would otherwise indicate. OpenAI spins this catch with wry, Wittgensteinian literalness: "Many| words| map| to| one| token|,| but| some| don|'t|:| ind|iv|isible|".

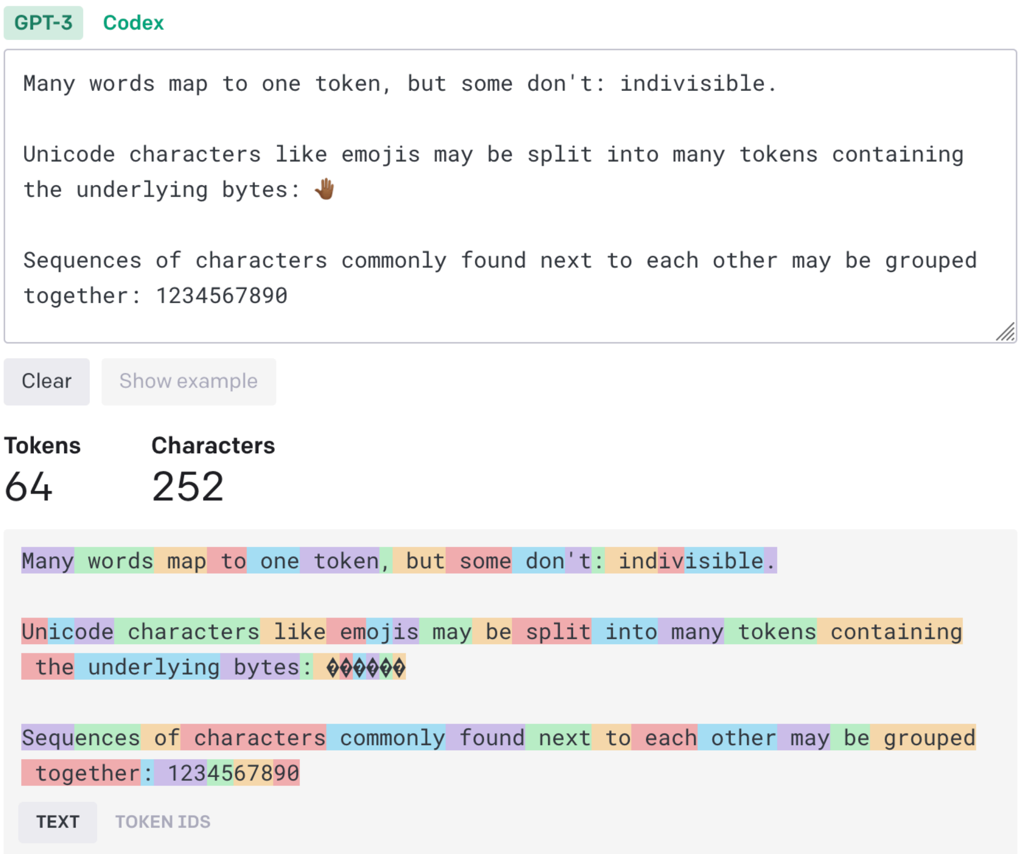


Figure 2: OpenAI's tokenizer tool.

To OpenAI's pricing schemes one might add thousands of Jupyter Notebook tutorials, many of the apps hosted by Hugging Face Spaces, prompt engineering services, "chain authoring" (Wu et al. 2022), and startups solely dedicated to optimizing models for low-cost deployment. The ordinary practice of language modeling is now stationed within a whole "semiotic infrastructure" dedicated to managing and manipulating textual data (Weatherby and Justie 2022, 382). The work of mapping the many concatenations that support this new infrastructure will be, I expect, a major task of future source code critics. But so too may the effects of these concatenations be found on the very surfaces of LLM outputs. They abide, for instance, in the difference between "indivisible" and "ind|iv|isible," and even in moments when, as with "lower" and "lower," that difference becomes infra-thin.

*

My second route is semiotic, and it requires digging up the origins of subword tokenization. In its current form, the BPE algorithm is very much a product of contemporary machine learning. Curiously, it does not originate from earlier text preparation methods in NLP, like stemming or lemmatization. Its developers instead followed what is now a common pattern among machine learning practitioners, sourcing the technique from a distant research domain and patching it ad hoc into their own system at Google (see Mackenzie 2017; Roberge and Castelle 2021). BPE was originally intended as a data compression technique, and it was first published in 1994 by Philip Gage, a software engineer. Gage's version works on the same bigram logic as today's tokenizers, but instead of matching and merging chunks of text, it selects pairs of adjacent bytes in a block of data and encodes them into a single byte. It then iteratively builds on the units it creates. Bytes that represent encoded pairs are encoded into new bytes, the latter into another set, and so on, until the compressor reduces its input data into the smallest possible footprint.

Gage provides an example: a sequence like ABABCABCD would become XXCXCD, where X stands for AB; the new sequence would in turn become XYYD, where Y is XC (1994, 3). Likewise, "l o w e r </w>" becomes "l o w er </w>", and then eventually "l ow er </w>," "l ow er</w>," "low er</w>," and finally "lower</w>." Significantly, in the latter case the sequences a tokenizer uses to represent subwords are not external data; instead, they are fragments of the original character sequences in the corpus. In something of a reversal of the original BPE logic, subword tokenization encodes texts by decompressing them into longer sequences, before decoding them back into more recognizable strings.

Alexandra Schneider's definition of compression, that it is "the reduction of data to the threshold of comprehensibility," would be one way to state my limit case challenge between word and subword in the context of the original BPE algorithm (2019, 140). But Gage himself will also move in the direction of textual matters in 1996, when he returns to his compressor and rewrites it (Gage 1997). This second version only accepts text files. It works quite similarly to his 1994 program, though Gage adds an intriguing check, which runs before the program begins compressing data. I record a fragment of the relevant source code (in C) below:

---

```c
/* The compressor creates, among other macros, a maximum size
 * limit for the blocks of data it will read in. */
#define MAXSIZE 65535L

/* With this done, it defines a function, which accepts
 * arguments for a file to be compressed (input) and the
 * compressed version of that file (output). */
void compress (FILE *input, FILE *output)
{
    /* Several variables are initialized at the top of this
     * function. I will only list one of them: the variable
     * that handles buffer allocation, which corresponds to
     * the size of the macro set above. When the compressor
     * assigns this variable, it creates a pointer to the
     * location in memory where the input file will be
     * stored. */
    buffer = (unsigned char *)malloc(MAXSIZE);

```

```
    /* With all variables initialized, the function reads the
     * input file into the memory buffer and performs the
     * first of two error checks. */
    size = fread(buffer,1,MAXSIZE,input);

    /* First, it determines whether the file is too large to
     * compress. If so, the compressor outputs an error
     * message to the user and quits. */
    if (size == MAXSIZE) {
        printf("File too big\n");
        exit(1);
    }

    /* If the file passes the size check, the compressor
     * performs a second check. It iterates through each
     * byte in the buffer and, ... */
    for (i=0; i<size; i++)
        /* ...for every byte, it checks whether the number
         * this byte represents exceeds 127. */
        if (buffer[i] > 127) {
            /* If the byte is larger than 127, the
             * compressor will also quit, again returning
             * an error message before doing so. The reason
             # it cannot accept anything larger than this
             * number has to do with a decision Gage makes:
             * unlike the original compressor, which looks
             * for any unused byte in the memory block, the
             * text version reserves all high-order bytes
             * (128-256) to encode pairs. */
            printf("This program works only on text files\n");
            exit(1);
        }
}
```

In text, the high-order bytes that Gage's compressor checks for represent the extended set of ASCII characters. The practical consequence of this check is that any character within this set cannot be compressed by the algorithm. Accented characters (Á, ñ), certain mathematical and currency symbols (±, ¼, ¥), characters for rendering basic text graphics (░, ┤), and others will all raise an error: "This program only works on text files." Put another way, anything beyond what the Unicode Consortium calls "Basic Latin" is, in this source code, *not text*, or ~~text~~.

There is much to elaborate here about how this erasure re-inscribes the linguistic hegemony of Latin alphabetics in digital media; among others, Lydia Liu's work on Shannon and "Printed English" (mentioned above) would be a major point of reference. For the time being, however, I will only acknowledge this as a future site of critique—and add, as a preliminary step toward one, a reminder that, for contemporary LLMs, BPE tokenization often serves to handle the very kind of text that Gage puts under erasure. This adds an ironic dimension to BPE's proliferation, to be sure. But in working toward a close, I want to pursue another dimension of the message, "This program only works on text files." For this message also suggests that the output of BPE compression is itself ~~text~~. That is, if high-order ASCII is ~~text~~, then any data encoded into this space will also fall under that same category, subwords included. To read the message in this second sense is to therefore return to the considerations I have already laid out: what is this ~~text~~, and in what way might it shape the unique textuality of subwords, these tokens that are neither language nor words?

Among the source code files of current BPE tokenizers, there is a naming convention that suggests one answer. The code blocks for the 2016 Google Translate model have made use of it already:

```
for word, freq in vocab.items():
    symbols = word.split()
```

Here it is (in Rust) in the BERT tokenizer hosted at Hugging Face ("Tokenizers" 2022):

```
#[derive(Debug, Clone, Copy)]
struct Symbol {
    c: u32,
    prev: isize,
    next: isize,
    len: usize
}
impl Symbol {
    /// Merges the current Symbol with the other one.
    /// In order to update prev/next, we consider Self to be the
    /// symbol on the left, and other to be the one on the right.
    pub fn merge_with($mut self, other: $Self, new_c: u32) {
        self.c = new_c;
        self.len += other.len;
        self.next = other.next;
    }
}
```

Elsewhere, it appears in explanatory comments, as in the Python function below, which is part of OpenAI's GPT-2 encoder (Radford et al. 2019):

```
def get_pairs(word):
    """Return a set of symbol pairs in a word.

    Word is represented as a tuple of symbols (symbols being
    variable-length strings).
    """
    pairs = set()
    prev_char = word[0]
    for char in word[1:]:
```

```
        pairs.add((prev_char, char))
        prev_char = char
    return pairs
```

---

For these and other tokenizers, the concatenated strings generated by subword tokenization are *symbols*. Despite the many recent claims that say deep learning has brought about a paradigm shift in the practice and epistemology of AI, contemporary language modeling remains in an important sense tied to the realm of the symbolic. Within this realm, a symbol betokens a linkage, which may in turn betoken another. Whole chains of linkages are thus condensed down to the threshold of comprehensibility, into flattened, variable-length sequences of characters. One such chain links "lower" to its infra-thin variant, "lower." Other concatenations abound. All hold place, notating.

In doing so, they challenge, I think, the very borders of what constitutes modeling. For with subwords, LLMs do not model language, and neither do they model tokens nor text. They model ~~text~~, which is itself a bootstrapped model of the corpus data from which it was derived. Large language models model models. And we are left to read the results.

**Appendix**

Below I record outputs from several hundred iterations of the 2016 BPE tokenizer. The corpus is the same one Google researchers included in their original release of the algorithm. It contains ads for construction companies and travel agencies, snippets from the PHP manual, licensing boilerplate, terms and conditions, and bible verses—quintessential internet text, in other words.

1 iteration

```
i r o n c e m e n t i s a r e a d y f o r u s e p a s t e w h i c h
i s l a i d a s a f i l l e t b y p u t t y k n i f e o r f i n g
e r i n t h e m o u l d e d g e s ( c o r n e r s ) o f t h e s t
e e l i n g o t m o u l d . i r o n c e m e n t p r o t e c t s t
h e i n g o t a g a i n s t t h e h o t , a b r a s i v e s t e e l
c a s t i n g p r o c e s s . a f i r e r e s t a n t r e p a i r c
e m e n t f o r f i r e p l a c e s , o v e n s , o p e n f i r e
p l a c e s e t c . c o n s t r u c t i o n a n d r e p a i r o f
h i g h w a y s a n d . . . a n a n n o u n c e m e n t m u s t b e
c o m m e r c i a l c h a r a c t e r . g o o d s a n d s e r v i c
e s a d v a n c e m e n t t h r o u g h t h e P . O . B o x s y s t e
m i s N O T A L L O W E D . d e l i v e r i e s ( s p a m ) a n
d o t h e r i m p r o p e r i n f o r m a t i o n d e l e t e d . t
r a n s l a t o r I n t e r n e t i s a T o o l b a r f o r M S I
n t e r n e t E x p l o r e r . i t a l l o w s y o u t o t r a n
s l a t e i n r e a l t i m e a n y w e b p a s g e f r o m o n e l
a n g u a g e t o a n o t h e r .
```

250 iterations

```
iron c ement is a re ad y for use p ast e which is l a i d as a fil l et
by p ut t y k n i f e or f ing er in the m ould e d g es ( c or ners ) of
the st e el ing ot m ould . iron c ement prot ec ts the ing ot ag a in st
the h ot , ab r asi ve st e el c asting pro c ess . a f ire restan t re p
a ir c ement for f ire pl ac es , o v en s , open f ire pl ac es et c .
```

con str u c tion and re p a ir of high w ay s and ... an an n oun c ement m ust be com mer c i al char ac ter . g o o ds and s er vic es ad v an c ement th rou gh the P . O . B o x s y st em is N O T A L L O W E D . deli veri es ( sp am) and other i m proper in for mation del et ed . tr an s l at or I n ter net is a T o ol b ar for M S I n ter net E x plorer . it allow s you to tr an s l ate in re al ti me an y w e b p as g e from one l an g u ag e to an other .

500 iterations

iron c ement is a read y for use p ast e which is lai d as a fil let by put t y k ni fe or fing er in the m ould e d g es ( cor ners ) of the st e el ingot m ould . iron c ement prot ec ts the ingot ag ain st the h ot , abrasi ve st e el c asting pro c ess . a fire restant rep air c ement for fire plac es , o v en s , open fireplac es etc . con str uc tion and rep air of high w ays and ... an an n oun c ement m ust be commer ci al charac ter . go o ds and servic es ad v an c ement th rou gh the P . O . B o x syst em is N O T A L L O W E D . deli veries ( sp am ) and other im proper information deleted . trans lator I n ter net is a T o ol b ar for M S I n ter net E x plorer . it allow s you to trans late in real time an y we b p as ge from one lan g u age to an other .

2,500 iterations

iron cement is a ready for use paste which is laid as a fillet by putty kni fe or finger in the m ould ed ges ( cor ners ) of the ste el ingot m ould . iron cement protects the ingot against the hot , abrasive ste el casting process . a fire restant rep air cement for fire places , ovens , open fireplaces etc . con struc tion and rep air of high ways and ... an announ cement must be commercial charac ter . go ods and services advancement through the P . O . Bo x system is N O T A L L O W E D . deli veries ( sp am ) and other im proper information deleted . translator Internet is a T o ol b ar for MS Internet Ex plorer . it allows you to translate in real time any web pas ge from one language to another .

**References**

Bender, Emily M., Timnit Gebru, Angelina McMillan-Major, and Shmargaret Shmitchell. 2021. "On the Dangers of Stochastic Parrots: Can Language Models Be Too Big?" In *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*, 610–23. Virtual Event Canada: ACM. doi:10.1145/3442188.3445922.

Bender, Emily M., and Alexander Koller. 2020. "Climbing Towards NLU: On Meaning, Form, and Understanding in the Age of Data." In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 5185–98. Online: Association for Computational Linguistics. doi:10.18653/v1/2020.acl-main.463.

Bojanowski, Piotr, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2017. "Enriching Word Vectors with Subword Information." *arXiv:1607.04606 [Cs]*, June. http://arxiv.org/abs/1607.04606.

Burgess, Matt. 2016. "Google's AI Just Created Its Own Universal 'Language'." *Wired*, November. https://www.wired.co.uk/article/google-ai-language-create.

Cayley, John. 2013. "Terms of Reference and Vectoralist Transgressions." *Amodern* 2 (October). https://amodern.net/article/terms-of-reference-vectoralist-transgressions/.

Coldewey, Devin. 2016. "Google's AI Translation Tool Seems to Have Invented Its Own Secret Internal Language." *TechCrunch*, November. https://techcrunch.com/2016/11/22/googles-ai-translation-tool-seems-to-have-invented-its-own-secret-internal-language/.

Daras, Giannis. 2022. "DALLE-2 Has a Secret Language." *Twitter*. https://twitter.com/giannis_daras/status/1531693093040230402.

Daras, Giannis, and Alexandros G. Dimakis. 2022. "Discovering the Hidden Vocabulary of DALLE-2." doi:10.48550/ARXIV.2206.00169.

Dean, Jeff. 2021. "Introducing Pathways: A Next-Generation AI Architecture." *The Keyword*. https://blog.google/technology/ai/introducing-pathways-next-generation-ai-architecture/.

Devlin, Jacob, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019.

"BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding." *arXiv:1810.04805 [Cs]*, May. http://arxiv.org/abs/1810.04805.

Gage, Philip. 1994. "A New Algorithm for Data Compression." *The C Users Journal* 12 (2): 1–14.

———. 1997. "Byte Pair Encoding." https://github.com/Algorithms-in-cpp/CUJ-1990-2000/tree/1fd9fc2ed887229c0c9774e408e2c0f17e010910/SOURCE/1997/SEP97.

Gavin, Michael. 2019. "Is There a Text in My Data? (Part 1): On Counting Words." *Journal of Cultural Analytics*, September. doi:10.22148/001c.11830.

Golumbia, David. 2009. *The Cultural Logic of Computation*. Cambridge, MA: Harvard University Press.

Hart, Horace. 1921. *Hart's Rules for Compositors and Readers at the University Press, Oxford.* 26th ed. London: Oxford University Press. //catalog.hathitrust.org/Record/007012963.

Kaplan, Frédéric. 2014. "Linguistic Capitalism and Algorithmic Mediation." *Representations* 127 (1): 57–63. doi:10.1525/rep.2014.127.1.57.

Kimbrell, Roy. 1988. "Searching for Text? Send an N-Gram!" *Byte* 13 (5): 297–312. https://ia802908.us.archive.org/7/items/byte-magazine-1988-05/byte-magazine-1988-05.pdf.

LaFrance, Adrienne. 2017. "What An AI's Non-Human Language Actually Looks Like." *The Atlantic*, June. https://www.theatlantic.com/technology/archive/2017/06/what-an-ais-non-human-language-actually-looks-like/530934/.

Lazzarato, Maurizio. 2014. *Signs and Machines: Capitalism and the Production of Subjectivity*. Translated by Joshua David Jordan. Semiotext(e) Foreign Agents Series. Los Angeles, CA: Semiotext(e).

Liu, Lydia H. 2010. *The Freudian Robot: Digital Media and the Future of the Unconscious*. Chicago: University of Chicago Press.

Mackenzie, Adrian. 2017. *Machine Learners: Archaeology of a Data Practice*. Cambridge, MA: The MIT Press.

Marino, Mark C. 2020. *Critical Code Studies*. Software Studies. Cambridge, Massachusetts: The MIT Press.

Mikolov, Tomáš, Ilya Sutskever, Anoop Deoras, Hai-Son Le, Stefan Kombrink, and Jan Černocký. 2011. "Subword Language Modeling with Neural Networks," 4.

Millière, Raphaël. 2022. "Adversarial Attacks on Image Generation With Made-Up Words." arXiv. http://arxiv.org/abs/2208.04135.

Radford, Alec, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. "Language Models Are Unsupervised Multitask Learners." https://github.com/openai/gpt-2.

Roberge, Jonathan, and Michael Castelle, eds. 2021. *The Cultural Life of Machine Learning: An Incursion into Critical AI Studies*. Cham, Switzerland: Palgrave Macmillan. doi:10.1007/978-3-030-56286-1.

Schneider, Alexandra. 2019. "Viewer's Digest: Small Gauge and Reduction Prints as Liminal Compression Formats." In *Format Matters: Standards, Practices, and Politics in Media Cultures*, edited by Marek Jancovic, Axel Volmar, and Alexandra Schneider, 129–46. Lüneburg: meson press. https://doi.org/10.14619/1556.

Schuster, Mike, Melvin Johnson, and Nikhil Thorat. 2016. "Zero-Shot Translation with Google's Multilingual Neural Machine Translation System." *Google Research*. https://ai.googleblog.com/2016/11/zero-shot-translation-with-googles.html.

Schuster, Mike, and Kaisuke Nakajima. 2012. "Japanese and Korean Voice Search." In *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 5149–52. Kyoto, Japan: IEEE. doi:10.1109/ICASSP.2012.6289079.

Schütze, Hinrich. 1992. "Word Space." *Advances in Neural Information Processing Systems* 5.

Sennrich, Rico, Barry Haddow, and Alexandra Birch. 2016. "Neural Machine Translation of Rare Words with Subword Units." *arXiv:1508.07909 [Cs]*, June. http://arxiv.org/abs/1508.07909.

Shannon, Claude Elwood, and Warren Weaver. 1998. *The Mathematical Theory of Communication*. 21st ed. Urbana: University of Illinois Press.

Singla, Yaman Kumar, Swapnil Parekh, Somesh Singh, Changyou Chen, Balaji Krishnamurthy, and Rajiv Ratn Shah. 2022. "MINIMAL: Mining Models for Universal Adversarial Triggers." *Proceedings of the AAAI Conference on Artificial Intelligence* 36 (10): 11330–39. doi:10.1609/aaai.v36i10.21384.

Soon, Winnie, and Geoff Cox. 2021. *Aesthetic Programming: A Handbook of Software Studies*. London: Open Humanities Press.

Thornton, Pip. 2016. "{Poem}.py." https://culture.theodi.org/poem-py/.

———. 2018. "Language in the Age of Algorithmic Reproduction: A Critique of Linguistic Capitalism." PhD thesis, London: Royal Holloway, University of London.

"Tokenizers." 2022. Hugging Face. https://github.com/huggingface/tokenizers/releases/tag/v0.13.1.

Wallace, Eric, Shi Feng, Nikhil Kandpal, Matt Gardner, and Sameer Singh. 2019. "Universal Adversarial Triggers for Attacking and Analyzing NLP." In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, 10. Hong Kong, China: Association for Computational Linguistics. doi:10.18653/v1/D19-1221.

Walton, Nick. 2020. "Gpt-3-Encoder." https://www.npmjs.com/package/gpt-3-encoder.

Weatherby, Leif, and Brian Justie. 2022. "Indexical AI." *Critical Inquiry* 48 (2): 381–415. doi:10.1086/717312.

Wu, Tongshuang, Ellen Jiang, Aaron Donsbach, Jeff Gray, Alejandra Molina, Michael Terry, and Carrie J Cai. 2022. "PromptChainer: Chaining Large Language Model Prompts Through Visual Programming." In *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems*. CHI EA '22. New York, NY, USA: Association for Computing Machinery. doi:10.1145/3491101.3519729.