**Word tokenization as compression**

[[Matthew Kwan](https://medium.com/@matti.kwan?source=post_page-----2540260f6eda--------------------------------)](https://medium.com/@matti.kwan?source=post_page-----2540260f6eda--------------------------------)

[Matthew Kwan](https://medium.com/@matti.kwan?source=post_page-----2540260f6eda--------------------------------)

·

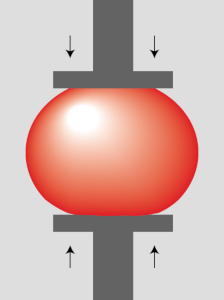
[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2F4f67732626e9&operation=register&redirect=https%3A%2F%2Fmedium.com%2F%40matti.kwan%2Fword-tokenization-as-compression-2540260f6eda&user=Matthew+Kwan&userId=4f67732626e9&source=post_page-4f67732626e9----2540260f6eda---------------------post_header-----------)

6 min read

·

Oct 25

5



In a [previous post](https://medium.com/@matti.kwan/finding-the-optimal-number-of-dimensions-for-word-embeddings-f19f71666723) I talked about finding the optimal number of dimensions for text embeddings. As part of that research I first had to tokenize the text corpus, and my quick-and-dirty approach left a lot to be desired.

First I split the text corpus into word fragments using the NLTK [treebank tokenizer](https://www.nltk.org/_modules/nltk/tokenize/treebank.html), converted the fragments to lower-case, and counted their usage. The 20,000 most common fragments got their own token.

Word fragments that didn’t match a token were first matched to the longest existing token, then *suffix tokens* were created to make up the rest of the fragment. For example, the word **duckling** might not occur frequently enough to get its own token, so it’s represented by the token for **duck**, followed by the suffix tokens -**l**, -**i**, -**n**, and -**g**.

This ran into a few problems:

1. The treebank parser, written in Python, is *slow*. Processing the text corpus took a couple of days.
2. The corpus contained a lot of non-Latin characters. Although I selected the English subset of the Gutenberg corpus, it contained things like an English-Japanese dictionary, and all of those Japanese UTF-8 characters got their own token, which blew out the token count. After ingesting Gutenberg, the token count was around 25,000, which was *just* manageable. The Wikipedia corpus blew that out to 30,000, which *wasn’t*.
3. Despite building tokens from a massive corpus, there’s was no guarantee the tokens could represent the *next* document.

So I needed to make a few changes:

1. Rewrite it in C++. Although I ended up writing a sanitizer in Python to get rid of embedded HTML and Wikipedia markup, using C++ for everything else was a lot faster and more memory-efficient.
2. Treat word fragments as byte sequences rather than UTF-8 character sequences. This way, by initializing with 256 one-byte tokens and 256 one-byte suffix tokens, you are guaranteed to parse any document. It does mean the algorithm is case-sensitive, but that’s probably a good thing. For example, the words **trump** and **Trump** usually don’t mean the same thing! Switching to C++ also made byte sequences easier to process.
3. Design a corpus-independent way to turn byte sequences into tokens.

I ended up treating a sequence of bytes as a *word fragment* if it

* formed a sequence of ISO 8859–1 “letters”;
* formed a complete multi-byte UTF-8 sequence; or
* was a single non-alphabetic, non-whitespace ASCII character

I realize these rules favour English and Western European languages, but that’s what I’ve got in my training corpus. I could add rules for letter sequences in Russian and Arabic, but I’m not familiar with those languages, and I’d end up playing whack-a-mole with every new alphabet that appeared in the corpus. So it’s not perfect, but it does the job.

**Optimizing for fewer tokens**

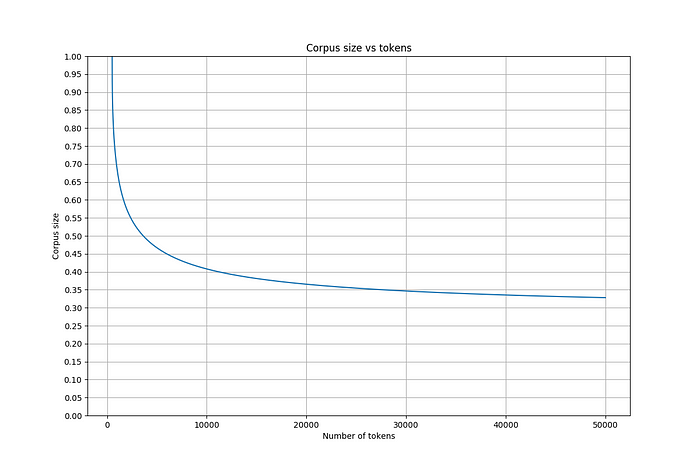
The approach I chose for assigning tokens was to minimize the number of tokens needed to represent the corpus.

Why? Well, in theory you could represent a corpus as a sequence of bits, with two tokens **0** and **1**. But the information content per token would be tiny, so you couldn’t generate useful embeddings. Same problem with tokenizing each character (at least for alphabetic languages — it might work for Chinese).

Thus you want to maximize the information content per token, which means minimizing the number of tokens needed to represent the corpus information. And, for English at least, that means tokenizing whole words where possible (it would be nice to also tokenize word *pairs* such as **ad** **hoc**, but the gains probably wouldn’t justify the complexity).

So how does the approach word? Let’s imagine the word **duck** appears a million times. Initially it will be represented by four single-byte tokens **d**, -**u**, -**c**, and -**k** — so four million tokens in total. Creating the token **duck** will thus reduce the token count by three million.

So a simple algorithm is to count the usage of each word fragment, sort them by descending (**usage** x (**length**-1)), and create tokens for the first *N* entries in the list. This works fairly well:



In terms of compression, we save about 25% by discarding whitespace, and reduce the token count by two-thirds using the algorithm. Each token requires two bytes, so the effective compression ratio is around 50% (but lossy, since whitespace information is lost).

Peak compression occurs with 65,536 unique tokens, making full use of a 2-byte encoding. However, if you look at it from an information-theoretical perspective, counting the number of token *bits* needed to encode the corpus, 52,081 unique tokens (15.7 bits per token) is optimal (although between 14 and 16 bits there isn’t much difference).

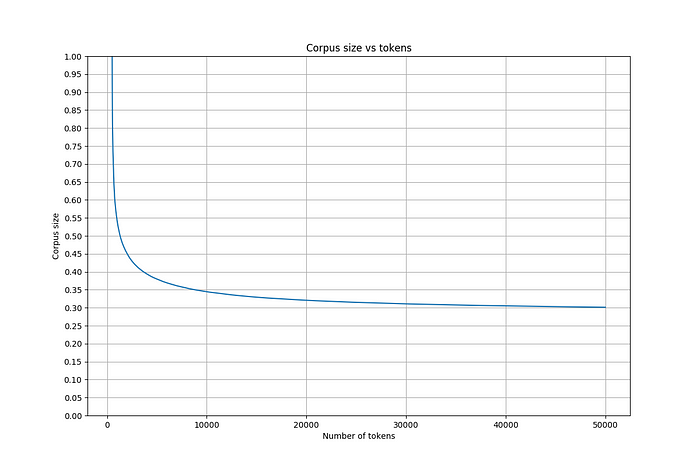
**Optimizing suffixes**

I figured I could get better results by also optimizing the suffix tokens. Imagine you have the words **duck**, **duckling**, **year**, and **yearling**. Depending on the usage counts, you might want to generate the tokens **duck**, **year**, and -**ling**. This is especially useful when the suffix token has a consistent meaning across words, for example -**smith**.

Implementing this optimization turned out to be complicated and slow, but it worked eventually. The approach was to

1. Represent each word in the corpus as a sequence of single-byte tokens.
2. For each word, generate a set of token sub-sequences that are candidates for tokenization.
3. Select the sub-sequence that will result in the biggest reduction in the corpus size, and replace it with a single token. Update all the words to use the new token.
4. Repeat until you hit the desired number of unique tokens.

This reduced the corpus size by about 8% relative to the simple algorithm. Not bad, and it’s nice being able to quantify the improvement. For reference, the optimal number of tokens from a compression-per-bit perspective was 14,474 (13.8 bits).



**Conclusion**

So I ended up with a decent tokenization algorithm. The results for English text are pretty good, and it should work well for any languages that use the Latin alphabet.

It will still work, but be less efficient, for other languages — although it could easily be improved for something like Russian and Arabic by identifying which UTF-8 characters are used to compose words. For Chinese and Japanese however, I don’t know if they even have a clear concept of a *word*, or whether word boundaries can be easily detected, so the algorithm would need fine-tuning for those languages.

But most important, by measuring compression rates, I now have an *objective* measure of the effectiveness of a tokenization strategy. For a given corpus I can try different ideas and pick a clear winner based on numbers, rather than gut instinct.

I can also make an informed decision about the number of tokens to use. Personally I’m reluctant to go beyond 20,000 tokens, because the resulting token-pair matrices (400 million cells) are at the limit of what my GPU can handle in-memory, so it’s helpful to quantify the trade-off relative to, say, 30,000 tokens (not worth it!).

So there we have it: some new tokenization algorithms, and a technique for measuring their effectiveness. I hope someone finds this useful.