From Words To Tokens: The WordPiece Algorithm

Damien Benveniste

The WordPiece algorithm is a subword tokenization algorithm that was initially developed by Google for use in neural machine translation in 2012[2, 3] and later popularized by models like BERT[1] in 2018. Tokenization is the feature engineering of natural language processing, and the goal of WordPiece is to find the "best" features!

We are going to create a toy example to illustrate the mechanisms of this algorithm. Let's assume we have very simple training data composed of four text sequences {`'low'`, `'lower'`, `'newest'`, `'widest'`}. We want to find consecutive groups of letters that can be used as tokens. At first, we are going to assume that the tokens are just the single characters themselves:

- `[_, l, o, w]`
- `[_, l, o, w, e, r]`
- `[_, n, e, w, e, s, t]`
- `[_, w, i, d, e, s, t]`

The character '_' is just a special character used as a marker to indicate the beginning of a word[1]. The current vocabulary capturing that character-level tokenization is just:

`{_, l, o, w, e, r, n, s, t, d}`

A typical strategy to guide the quality of a modeling choice is the likelihood maximization principle computed on the training data. The likelihood, in this context, is just the product of the text sequence probabilities:

$$\begin{aligned}\mathcal{L} =& P(\texttt{\_,l,o,w})\times \\ & P(\texttt{\_,l,o,w,e,r})\times \\ & P(\texttt{\_,n,e,w,e,s,t})\times \\ & P(\texttt{\_,w,i,d,e,s,t})\end{aligned} \tag{1}$$

We would like to group together characters that tend to co-occur often in the training data or, probabilistically speaking, characters that depend on each other. If A and B are two probabilistic events, we say that A and B are independent if:

$$P(A, B) = P(A)P(B)$$

and conversely, they are dependent if

$$P(A, B) \neq P(A)P(B).$$

If we have $P(A, B) > P(A)P(B)$, it means that A and B are more likely to co-occur than random chance would predict. Currently, with our simple character-level tokens, we are implying independence of the different characters:

[1] Note that there are few strategies to capture the word boundaries, and the character '_' was the one described in [3].

- $P(\texttt{\_,l,o,w}) = P(\texttt{\_})P(\texttt{l})P(\texttt{o})P(\texttt{w})$
- $P(\texttt{\_,l,o,w,e,r}) = P(\texttt{\_})P(\texttt{l})P(\texttt{o})P(\texttt{w})P(\texttt{e})P(\texttt{r})$
- $P(\texttt{\_,n,e,w,e,s,t}) = P(\texttt{\_})P(\texttt{n})P(\texttt{e})P(\texttt{w})P(\texttt{e})P(\texttt{s})P(\texttt{t})$
- $P(\texttt{\_,w,i,d,e,s,t}) = P(\texttt{\_})P(\texttt{w})P(\texttt{i})P(\texttt{d})P(\texttt{e})P(\texttt{s})P(\texttt{t})$

To find better tokens, we need to be able to estimate those probabilities. The most basic approach is to count the tokens. For example, for the character `e`:

$$P(\texttt{e}) \simeq \frac{\text{Freq}(\texttt{e})}{N} = \frac{4}{24} \tag{2}$$

Where Freq(`e`) is the number of occurrences of the character `e`, and N is the total number of tokens. Let's now consider all the pairs of consecutive characters in the training data:

`{_l, lo, ow, we, er, _n ,ne, ew, es, st, _w, wi, id, de}`

Again, to estimate the probabilities associated with those pairs, we count the occurrences. For example, for the pair `lo`:

$$P(\texttt{lo}) \simeq \frac{\text{Freq}(\texttt{lo})}{N_{\text{pairs}}} = \frac{2}{20} \tag{3}$$

Here, $N_{\text{pairs}} = N - 4$ represents the number of continuous pairs of characters in the training data, where 4 is the number of text sequences of our toy example[2]. How does that compare to $P(\texttt{l})P(\texttt{o})$?

$$P(\texttt{l})P(\texttt{o}) \simeq \frac{\text{Freq}(\texttt{l})}{N}\frac{\text{Freq}(\texttt{o})}{N} = \frac{2}{24} \times \frac{2}{24} = \frac{1}{144} \tag{4}$$

So $P(\texttt{lo}) > P(\texttt{l})P(\texttt{o})$ and the pair `lo` is a good token candidate to increase the likelihood because

$$P(\texttt{\_,lo,w})P(\texttt{\_,lo,w,e,r}) > P(\texttt{\_,l,o,w})P(\texttt{\_,l,o,w,e,r}) \tag{5}$$

To maximize the likelihood as much as possible, we actually want to find the pair of characters a and b that maximize the ratio $\frac{P(ab)}{P(a)P(b)}$ as long as it is greater than 1.

The algorithm starts to emerge! We can iteratively find the pair of tokens that maximizes the likelihood the most and add that pair of tokens as a new token into the vocabulary until the vocabulary is large enough. Let's go through a few iterations of this process to help us visualize better.

First iteration

Step 1. We start with the character-level tokenization:

- `[_, l, o, w]`
- `[_, l, o, w, e, r]`
- `[_, n, e, w, e, s, t]`
- `[_, w, i, d, e, s, t]`

with initial token dictionary:

`{_, l, o, w, e, r, n, s, t, d}`

[2]It is, in general, more accurate to estimate $P(ab)$ as the ratio $\text{Freq}(ab)/N_{\text{pairs}}$, but in practice, due to the number of tokens within each text sequence of typical training data, $N_{\text{pairs}} \simeq N$ is a valid approximation. It does make a non-negligible difference in the toy example we chose in this chapter.

Step 2. We compute the ratio $\mathcal{L}_{\text{new}}/\mathcal{L}_{\text{old}}$ for all the possible merges of pairs of consecutive tokens. Because the likelihood is multiplicative, the ratio simplifies to:

$$\left.\frac{\mathcal{L}_{\text{new}}}{\mathcal{L}_{\text{old}}}\right|_{\texttt{lo}} = \left[\frac{P(\texttt{lo})}{P(\texttt{l})P(\texttt{o})}\right]^2 \tag{6}$$

when we merge the characters `lo`. For all the possible merges:

Merges	$\mathcal{L}_{\text{new}}/\mathcal{L}_{\text{old}}$	Merges	$\mathcal{L}_{\text{new}}/\mathcal{L}_{\text{old}}$
`_l`	$\left[\frac{P(\texttt{\_l})}{P(\texttt{\_})P(\texttt{l})}\right]^2 \simeq 51.84$	`ew`	$\frac{P(\texttt{ew})}{P(\texttt{e})P(\texttt{w})} \simeq 1.8$
`lo`	$\left[\frac{P(\texttt{lo})}{P(\texttt{l})P(\texttt{o})}\right]^2 \simeq 207.36$	`es`	$\left[\frac{P(\texttt{es})}{P(\texttt{e})P(\texttt{s})}\right]^2 \simeq 51.84$
`ow`	$\left[\frac{P(\texttt{ow})}{P(\texttt{o})P(\texttt{w})}\right]^2 \simeq 51.84$	`st`	$\left[\frac{P(\texttt{st})}{P(\texttt{s})P(\texttt{t})}\right]^2 \simeq 207.36$
`we`	$\left[\frac{P(\texttt{we})}{P(\texttt{w})P(\texttt{e})}\right]^2 \simeq 12.96$	`_w`	$\frac{P(\texttt{\_w})}{P(\texttt{\_})P(\texttt{w})} \simeq 1.8$
`er`	$\frac{P(\texttt{er})}{P(\texttt{e})P(\texttt{r})} \simeq 7.2$	`wi`	$\frac{P(\texttt{wi})}{P(\texttt{w})P(\texttt{i})} \simeq 7.2$
`_n`	$\frac{P(\texttt{\_n})}{P(\texttt{\_})P(\texttt{n})} \simeq 7.2$	`id`	$\frac{P(\texttt{id})}{P(\texttt{i})P(\texttt{d})} \simeq 28.8$
`ne`	$\frac{P(\texttt{ne})}{P(\texttt{n})P(\texttt{e})} \simeq 7.2$	`de`	$\frac{P(\texttt{de})}{P(\texttt{d})P(\texttt{e})} \simeq 7.2$

We see that the best merge candidates for likelihood increase are `lo` and `st`. In the case where we have two candidates with the same likelihood increase, we can always have a heuristic rule to make a deterministic choice. Let's choose the first one in alphabetical order: `lo`.

Step 3. We update the token dictionary with the new merged token:

`{_, l, o, w, e, r, n, s, t, d, lo}`

and we adapt the tokenization to account for the new token:

- `[_, lo, w]`
- `[_, lo, w, e, r]`
- `[_, n, e, w, e, s, t]`
- `[_, w, i, d, e, s, t]`

Second iteration

Step 1. We start from the current tokenization and dictionary.

Step 2. We compute the ratio $\mathcal{L}_{\text{new}}/\mathcal{L}_{\text{old}}$ for all the possible merges of pairs of consecutive tokens.

Merges	$\mathcal{L}_{\text{new}}/\mathcal{L}_{\text{old}}$	Merges	$\mathcal{L}_{\text{new}}/\mathcal{L}_{\text{old}}$
`_lo`	$\left[\frac{P(\texttt{\_lo})}{P(\texttt{\_})P(\texttt{lo})}\right]^2 \simeq 45.19$	`ew`	$\frac{P(\texttt{ew})}{P(\texttt{e})P(\texttt{w})} \simeq 1.68$
`low`	$\left[\frac{P(\texttt{low})}{P(\texttt{lo})P(\texttt{w})}\right]^2 \simeq 45.19$	`es`	$\left[\frac{P(\texttt{es})}{P(\texttt{e})P(\texttt{s})}\right]^2 \simeq 45.19$
`we`	$\left[\frac{P(\texttt{we})}{P(\texttt{w})P(\texttt{e})}\right]^2 \simeq 11.30$	`st`	$\left[\frac{P(\texttt{st})}{P(\texttt{s})P(\texttt{t})}\right]^2 \simeq 180.75$
`er`	$\frac{P(\texttt{er})}{P(\texttt{e})P(\texttt{r})} \simeq 6.72$	`_w`	$\frac{P(\texttt{\_})}{P(\texttt{w})P(\texttt{i})} \simeq 1.68$
`_n`	$\frac{P(\texttt{\_n})}{P(\texttt{\_})P(\texttt{n})} \simeq 6.72$	`wi`	$\frac{P(\texttt{id})}{P(\texttt{i})P(\texttt{d})} \simeq 6.72$
`ne`	$\frac{P(\texttt{ne})}{P(\texttt{n})P(\texttt{e})} \simeq 6.72$	`id`	$\frac{P(\texttt{id})}{P(\texttt{i})P(\texttt{d})} \simeq 26.89$
`de`	$\frac{P(\texttt{de})}{P(\texttt{d})P(\texttt{e})} \simeq 6.72$		

`st` is the best merge candidate.

Step 3. We update the token dictionary with the new merged token:

`{_, w, e, r, n, s, t, d, lo, st}`

and we adapt the tokenization to account for the new token:

- `[_, lo, w]`
- `[_, lo, w, e, r]`
- `[_, n, e, w, e, st]`
- `[_, w, i, d, e, st]`

The process is straightforward, and we can iterate as many times as we wish. Let's summarize the algorithm:

1. We start with character-level tokenization
2. We compute $\mathcal{L}_{\text{new}}/\mathcal{L}_{\text{old}}$ for all the possible merges of pairs of consecutive tokens.
3. We merge the token that maximizes the likelihood.
4. Repeat the process until you reach the desired number of merges.

or in a more pseudo-code format:

Input: Text corpus, desired number of merges N
Output: Final WordPiece vocabulary
// Step 1: Start with character-level tokenization
tokens ← CharTokenize(corpus)
for $n = 1$ to N **do**
    // We'll perform N merges
    for all pairs of consecutive tokens (t_i, t_{i+1}) in tokens **do**
        // Step 2: Compute likelihood ratio for merging each pair
        Compute $\mathcal{L}_{\text{new}}$ for merging t_i and t_{i+1}
        Let $\mathcal{L}_{\text{old}}$ be the current likelihood
        $\Delta_i \leftarrow \mathcal{L}_{\text{new}} \,/\, \mathcal{L}_{\text{old}}$
    end for
    // Step 3: Merge the pair that maximizes the ratio

14: Find the pair (t_p, t_{p+1}) with the maximum Δ_p
15: Merge (t_p, t_{p+1}) into a single token
16: **end for**

By applying this algorithm, we could continue the merging process and progressively add more tokens to the dictionary:

- **3rd iteration:** We add `_lo` to the dictionary:

  `[_lo, w]`, `[_lo, w, e, r]`, `[_, n, e, w, e, st]`, `[_, w, i, d, e, st]`

- **4th iteration:** We add `id` to the dictionary:

  `[_lo, w]`, `[_lo, w, e, r]`, `[_, n, e, w, e, st]`, `[_, w, id, e, st]`

- **5th iteration:** We add `_low` to the dictionary:

  `[_low]`, `[_low, e, r]`, `[_, n, e, w, e, st]`, `[_, w, id, e, st]`

- **6th iteration:** ...

The original BERT-Base and BERT-Large had a final vocabulary of $\sim 30,000$ tokens.

References

[1] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. **BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding**. 2019. arXiv: 1810.04805 [cs.CL]. URL: https://arxiv.org/abs/1810.04805.

[2] Mike Schuster and Kaisuke Nakajima. "Japanese and Korean voice search". In: **2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)**. 2012, pp. 5149–5152. DOI: 10.1109/ICASSP.2012.6289079.

[3] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Łukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. **Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation**. 2016. arXiv: 1609.08144 [cs.CL]. URL: https://arxiv.org/abs/1609.08144.