Computational Approaches to Morphological Segmentation and Tokenization

David R. Mortensen

May 25, 2023

Introduction

In this lecture, we will look at five different algorithms for subword segmentation:

1. Byte-pair encoding
2. Wordpiece
3. Unigram tokenization
4. Sentencepiece
5. Morfessor

The first four are not primarily intended for morpheme segmentation (although this behavior should be a side effect, to some degree). The fifth is explicitly intended for morphological segmentation (but is very computationally intensive).

Tokenization Algorithms

The following algorithms have been used widely in practical NLP systems. The first two (BPE and Wordpiece) are very efficient to train and encoding and decoding are also efficient, so they are attractive for use with a large quantity of data. However, it is clear that neither of these algorithms approaches the theoretical limit in terms of tokenizing text into meaningful units.

In some respects, Unigram and Sentencepiece seem more promising on the theoretical front, though they are also less efficient than BPE and Wordpiece.

Byte-pair encoding

In byte-pair encoding[1], the model is initialized with character-level or byte level representations (in training). Subsequent pairs of bytes are then replaced with an as-yet unused symbol until a certain vocabulary size is reached. At each step, the byte-pair or character pair that is replaced by another token is the type for which there are the most tokens.

Take the following example. Start with a training corpus that contains five instances of *big*, three instances of *bigger*, one instance of *biggest*, and one instance of *Everest*. We can represent this as follows:

```
corpus = {"b i g </w>": 5,
          "b i g g e r </w>": 3,
          "b i g g e s t </w>": 1,
          "E v e r e s t </w>": 1}
```

[1] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1715–1725, Berlin, Germany, August 2016. Association for Computational Linguistics. DOI: 10.18653/v1/P16-1162. URL https://aclanthology.org/P16-1162

The `</w>` is used to mark the ends of words and to distinguish character sequences that occur there from those that occur elsewhere. The two most common symbol pairs that occur in the corpus are `b i` (9) and *i g* (9). Because we saw `bi` first, we will target it for replacement. We will replace all sequences of *b* and *i* with a new symbols, `bi`. This yields:

```
corpus = {"bi g </w>": 5,
          "bi g g e r </w>": 3,
          "bi g g e s t </w>": 1,
          "E v e r e s t </w>": 1}
```

We will add a rule to our grammar (which, plus the set of atomic symbols, defines our vocabulary):

```
bi → b i
```

The most frequent pair is now `bi g`, which we will replace with `big`

```
corpus = {"big </w>": 5,
          "big g e r </w>": 3,
          "big g e s t </w>": 1,
          "E v e r e s t </w>": 1}
```

We will push another rule to our grammar stack:

```
big → bi g
bi → b i
```

We again have a tie for most frequent pair (`big g` and `ge`). We'll target `big g` since we saw it first. We'll replace it with `bigg`:

```
corpus = {"big </w>": 5,
          "bigg e r </w>": 3,
          "bigg e s t </w>": 1,
          "E v e r e s t </w>": 1}
```

The grammar is now:

```
bigg → big g
big → bi g
bi → b i
```

Following the same logic

```
corpus = {"big </w>": 5,
          "bigge r </w>": 3,
          "bigge s t </w>": 1,
          "E v e r e s t </w>": 1}
```

The grammar is now:

```
bigge → bigg e
bigg → big g
big → bi g
bi → b i
```

Then:

```
corpus = {"big </w>": 5,
          "bigger </w>": 3,
          "bigge s t </w>": 1,
          "E v e r e s t </w>": 1}
```

Yielding the grammar:

```
bigger → bigge r
bigge → bigg e
bigg → big g
big → bi g
bi → b i
```

Then:

```
corpus = {"big </w>": 5,
          "bigger</w>": 3,
          "bigge s t </w>": 1,
          "E v e r e s t </w>": 1}
```

Yielding the grammar:

```
bigger</w> → bigger </w>
bigger → bigge r
bigge → bigg e
bigg → big g
big → bi g
bi → b i
```

Then:

```
corpus = {"big </w>": 5,
          "bigger</w>": 3,
          "bigge st </w>": 1,
          "E v e r e st </w>": 1}
```

Yielding the grammar:

```
st → s t
bigger</w> → bigger </w>
bigger → bigge r
bigge → bigg e
bigg → big g
big → bi g
bi → b i
```

Then:

```
corpus = {"big </w>": 5,
          "bigger</w>": 3,
          "bigge st</w>": 1,
          "E v e r e st</w>": 1}
```

Yielding the grammar:

```
st</w> → st </w>
st → s t
bigger</w> → bigger </w>
bigger → bigge r
bigge → bigg e
bigg → big g
big → bi g
bi → b i
```

And so on.

If this continues indefinitely, eventually there will be a token corresponding to each word type, which would defeat the purpose of subword tokenization. Instead, the vocabulary size is limited and the iterations stop when the vocabulary size is reached. In our toy example, we might say that the vocabulary size is set at 128 (the number of possible bytes) plus 8 (non-terminal symbols) In that case, our loop would terminate after `st</w>` was added to the vocabulary.

A non-terminal symbol, in formal language theory, is one that sits on the left-hand side of a production rule.

(1) A few observations:
  a. The larger the vocabulary, the fewer tokens per word.
  b. Common morphemes will tend to coalesce first.
  c. Uncommon words will not coalesce unless the vocabulary is very large.
  d. Subword token boundaries are not guaranteed to align with morpheme boundaries.

Decoding byte-pair encoded text is easy: apply each rule in the grammar, in succession, until the end of the grammar is reached. Encoding BPE text is similar. Start at the end of the grammar. For each pair on the right-hand side, parse it into the symbol on the left-hand side until the grammar is exhausted.

Wordpiece

At a deep level, the Wordpiece algorithm used in BERT and other Google products is rather similar to BPE. The major difference is that, rather than basing the target pair for replacement, at each step in the algorithm, upon relative frequency, it is based on pointwise mutual information:

$$PMI(a,b) = \frac{p(a,b)}{p(a)p(b)} \tag{1}$$

This places the results on slightly firmer information-theoretic ground than is the case for BPE. It means that the pairs that are fused are not simply those that are most frequent (which could result accidentally from *a* and *b* being, independently, very frequent) but because they are more frequent as a pair than would be expected from their frequencies generally.

There is another notational difference: in Wordpiece, non-initial tokens[2] are typically denoted by two hash marks. For example, prior to fusion, *big* might be represented as:

[2] Tokens that do not occur at the beginning of a word

```
("b", "##i", "##g")
```

This makes it easy to decode Wordpiece representations.

Unigram Tokenization

BPE and Wordpiece start with small vocabularies and build them up iteratively; Unigram tokenization starts with a very large vocabulary, then winnows it down by eliminating items that increase the loss on a (character-level) unigram language modeling task.[3] Unigram tokenization is founded on information theory: finding the optimal code length.

[3] Taku Kudo. Subword regularization: Improving neural network translation models with multiple subword candidates. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 66–75, Melbourne, Australia, July 2018. Association for Computational Linguistics. DOI: 10.18653/v1/P18-1007. URL https://aclanthology.org/P18-1007

Let us start with an example and some definitions. Consider the word *hat* with the characters h, a, and t. We can tokenize it into character-length tokens (and compute the unigram probabilities) as follows:

The numbers are made up, but the algorithm is real.

$$\begin{aligned} P([\text{"}h\text{"},\text{"}a\text{"},\text{"}t\text{"}]) &= P(\text{"}h\text{"}) \times P(\text{"}a\text{"}) \times P(\text{"}t\text{"}) && (2)\\ &= \frac{5}{210} \times \frac{36}{201} \times \frac{20}{210} && (3)\\ &= 0.000389 && (4)\end{aligned}$$

We can also tokenize *hat* into other units in our vocabulary:

$$\begin{aligned} P([\text{"}ha\text{"},\text{"}t\text{"}]) &= P(\text{"}ha\text{"}) \times P(\text{"}t\text{"}) && (5)\\ &= \frac{6}{210} \times \frac{20}{210} && (6)\\ &= 0.0022676 && (7)\end{aligned}$$

The unigram probability of a sequence is the product of the probabilities of each of the items in that sequence.

$$P(x) = \prod_{i=1}^{M} p(x_i), \quad \forall i\, x_i \in \mathcal{V}, \sum_{x \in \mathcal{V}} p(x) = 1 \qquad (8)$$

In (2) and (5) we see two different probabilities for different tokenizations of the same string. We see, as shown in (5), that a longer token sometimes results results in a higher probability that two shorter tokens (when the longer token is a sequence that is very frequent in the corpus). The reason for this is probably obvious: all of the probabilities are going to be less than one, so—other things being equal—longer sequences are going to have lower probabilities than shorter sequences. However, this is only true if the longer token is not very rare.

Finding the optimal segmentation, under a Unigram tokenization model, is given by

$$\mathbf{x}^* = \arg\max_{\mathbf{x} \in SX} P(\mathbf{x}), \tag{9}$$

where $S(X)$ is the set of segmentation candidates built from the input sequence X. Of course, computing this by brute force would be very expensive, so this is obtained using the Viterbi algorithm.[4]

[4] Andrew Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE transactions on Information Theory*, 13(2):260–269, 1967

If $\mathcal{V}$ is known, it is possible to estimate the subword occurrence probabilities $p(x_i)$ using a version of the EM algorithm. In this case, the EM algorithm maximizes the following marginal likelihood $\mathcal{L}$, under the assumption that $p(x_i)$ are hidden variables:

$$\begin{aligned} \mathcal{L} &= \sum_{s=1}^{|D|} \log(P(X^{(s)})) && (10) \\ &= \sum_{s=1}^{|D|} \log \left(\sum_{\mathbf{x} \in S(X^{(s)})} P(\mathbf{x}) \right) && (11) \end{aligned}$$

Here is the training algorithm as originally described in Kudo (2018):

1. Heuristically make a reasonably big seed vo- cabulary from the training corpus.
2. Repeat the following steps until $|\mathcal{V}|$ reaches a desired vocabulary size.
   (a) Fixing the set of vocabulary, optimize $p(x)$ with the EM algorithm.
   (b) Compute the loss_i for each subword x_i, where loss_i represents how the likelihood $\mathcal{L}$ is reduced when the subword x_i is removed from the current vocabulary.
   (c) Sort the symbols by loss_i and keep top η% of subwords (η is 80, for example). Note that we always keep the subwords consisting of a single character to avoid out-of-vocabulary.

How to prepare the seed vocabulary? Obvious choice: take the union of the character types and the most frequent substrings in the corpus. This can be done in $O(T)$ time and $O(20T)$ space using Nong et al., (2009)'s Enhanced Suffix Array algorithm.[5]

[5] Ge Nong, Sen Zhang, and Wai Hong Chan. Linear suffix array construction by almost pure induced-sorting. In *2009 data compression conference*, pages 193–202. IEEE, 2009

The final vocabulary $\mathcal{V}$ that results from training includes three kinds of segmentation candidates:

- characters
- subwords
- word

The choice between then, in segmenting a particular word, is probabilistic.

SentencePiece

SentencePiece is a tokenizer/detokenizer that does not assume that work sequences can be pre-tokenized by whitespace.[6] SentencePiece has two implementation: one of which is based on BPE and one of which is based on Unigram tokenization. In SentencePiece output, the character "⊠" is used to indicate a space, to detokenizing is simply a matter of deleting all spacess, then replacing "⊠" with a space.

[6] Taku Kudo and John Richardson. SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 66–71, Brussels, Belgium, November 2018. Association for Computational Linguistics. DOI: 10.18653/v1/D18-2012. URL `https://aclanthology.org/D18-2012`

Unsupervised Morphological Segmentation

Morfessor

Morfessor is a much more complicated set of algorithms that are specifically aimed at segmenting words into morphemes (and, unlike the unsupervised tokenization approaches described above, replicating linguists' segmentation of wrods).

Descriptions of Morfessor and its variants use some specialized terminology that can be confusing (since all of the terms have other meanings in other contexts):

atoms The smallest pieces of text with which the algorithms are concerned. For our purposes, these are CHARACTERS

compounds Sequences of atoms (the input of the learner algorithm). For our purposes, these are **words**

constructions Lexical units which are intermediate between atoms and compounds. What the algorithm is trying to discover. For our purposes, these are **morphs**

(2) Notes
   a. Compounds are sequences of constructions which are sequences of atoms
   b. Sequences can contain only a single item
      i. Constructions may consist of a single atom
      ii. Compounds may consist of a single construction

The following description of Morfessor is drawn from Virpioja et al.[7].

[7] Sami Virpioja, Peter Smit, Stig-Arne Grönroos, and Mikko Kurimo. Morfessor 2.0: Python Implementation and Extensions for Morfessor Baseline. Technical report, Aalto University, School of Electrical Engineering, 2013. URL `http://urn.fi/URN:ISBN:978-952-60-5501-5`

The analysis of a compound (word) w into analysis a is expressed by the tokenization function:

$$a = \phi(w; \theta) \tag{12}$$

where θ denotes the parameters of the model (a lexicon and a grammar)

The cost function is derived using MAP (maximum a posteriori estimation): find the mostly likely θ given and training data D_W:

$$\theta_{MAP} = \arg\max_{\theta}(\theta|D_W) = \arg\max_{\theta} p(\theta)p(D_W|\theta) \tag{13}$$

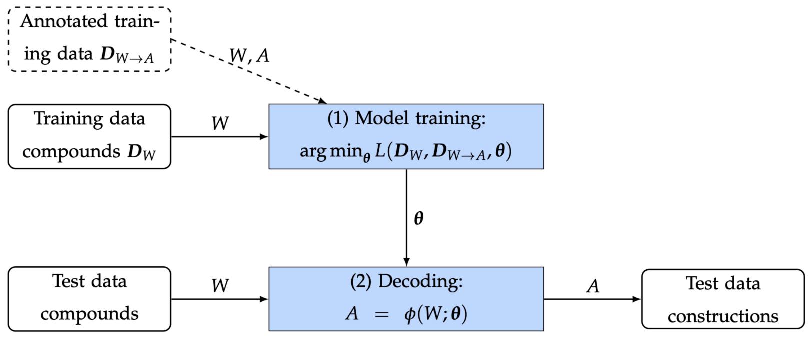
Annotated training data $\boldsymbol{D}_{W\to A}$
W, A
Training data compounds $\boldsymbol{D}_W$
W
(1) Model training: $\arg\min_{\boldsymbol{\theta}} L(\boldsymbol{D}_W, \boldsymbol{D}_{W\to A}, \boldsymbol{\theta})$
$\boldsymbol{\theta}$
Test data compounds
W
(2) Decoding: $A = \phi(W; \boldsymbol{\theta})$
A
Test data constructions

Figure 1: Overview of the Morfessor workflow. Parameters θ are chosed tha minimize the cost function with compounds W in training data D_W. θ is used to tokenize text data compounds.

The cost function is:

$$L(\theta, D_W) = -\log p(\theta) - \log p(D_W|\theta) \tag{14}$$

To compute the log likelihood, the equation is as follows:

$$\begin{aligned} \log p(D_W|\theta) &= \sum_{j=1}^{N} \log p(W = \mathbf{w}_j|\theta) && (15) \\ &= \sum_{j=1}^{N} \log \sum_{a\in\Phi(w_j)} p(A = \mathbf{a}|\theta) && (16) \end{aligned}$$

where

$$\phi(\mathbf{w}) = \mathbf{a} : \phi^{-1}(\mathbf{a}) = \mathbf{w} \tag{17}$$

Y assigns each compound $\mathbf{w}_j$ to a single analysis in $\Phi(w_j)$. Given the analyses for all words in the data, $\mathbf{Y} = (\mathbf{y}_1, \ldots, \mathbf{y}_N)$.

$$\begin{aligned} \log p(D_W|\theta, \mathbf{Y}) &= \sum_{j=1}^{N} \log p(y_i|\theta) && (18) \\ &= \sum_{j=1}^{N} \log p(m_{j1}, \ldots, m_{j|y_j|}, \#_w|\theta) && (19) \\ &= \sum_{j=1}^{N} \left(\log p(\#_w|\theta) + \sum_{i=1}^{y_i} \log p(m_{ji}|\theta) \right) && (20) \end{aligned}$$

In its most general form, the parameters θ are divided into a lexicon $\mathcal{L}$ and a grammar $\mathcal{G}$.

$\mathcal{L}$ includes the properties of "constructions"

$\mathcal{G}$ stipulates how "constructions" can be combined to form "compounds"

The prior in Morfessor assigns higher propabilities to lexicons with (1) fewer constructions (2) shorter constructions (where, by "stored" we mean that $p(m_i|\theta) > 0$). Thus, the probability of $\mathcal{L}$ can be expressed (for a lexicon of μ constructions) as

$$p(\mathcal{L}) = p(\mu) \times p\left(\text{properties}(m_1), \ldots, \text{properties}(m_\mu)\right) \times \mu! \qquad (21)$$

Each construction has two aspects: *form* and *usage*. *Form* is simply the atoms of which the construction is composed. Forms are considered to be independent. We can calculate the probability of the form $\boldsymbol{\sigma}_i$ of a construction m_i in terms of its length distribution $p(L)$ and its categorical distribution $p(C)$:

$$p(\boldsymbol{\sigma}_i) = p(L = |\boldsymbol{\sigma}_i|) \prod_{j=1}^{|\sigma_i|} p(C = \boldsymbol{\sigma}_{ij}) \qquad (22)$$

Algorithm 1: Batch training Morfessor with a global algorithm

```
function GlobalBatchTrain($D_W$, $\varepsilon$)
    $\boldsymbol{\theta}, \mathbf{Y} \leftarrow$ InitModel($D_W$)
    $L_{old} \leftarrow \infty$
    $L_{new} \leftarrow L(D_W, \boldsymbol{\theta}, \mathbf{Y})$
    while $L_{new} < L_{old} - \varepsilon$ do
        $\boldsymbol{\theta}, Y \leftarrow$ GlobalSearch($D_w, \boldsymbol{\theta}, \mathbf{Y}$)
        $L_{old} \leftarrow L_{new}$
        $L_{new} \leftarrow L(D_W, \boldsymbol{\theta}, \mathbf{Y})$
    return $\boldsymbol{\theta}, \mathbf{Y}$
```

References

Taku Kudo. Subword regularization: Improving neural network translation models with multiple subword candidates. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 66–75, Melbourne, Australia, July 2018. Association for Computational Linguistics. DOI: 10.18653/v1/P18-1007. URL `https://aclanthology.org/P18-1007`.

Taku Kudo and John Richardson. SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 66–71, Brussels, Belgium, November 2018. Association for Computational Linguistics. DOI: 10.18653/v1/D18-2012. URL `https://aclanthology.org/D18-2012`.

Ge Nong, Sen Zhang, and Wai Hong Chan. Linear suffix array construction by almost pure induced-sorting. In *2009 data compression conference*, pages 193–202. IEEE, 2009.

Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual*

Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 1715–1725, Berlin, Germany, August 2016. Association for Computational Linguistics. DOI: 10.18653/v1/P16-1162. URL `https://aclanthology.org/P16-1162`.

Sami Virpioja, Peter Smit, Stig-Arne Grönroos, and Mikko Kurimo. Morfessor 2.0: Python Implementation and Extensions for Morfessor Baseline. Technical report, Aalto University, School of Electrical Engineering, 2013. URL `http://urn.fi/URN:ISBN:978-952-60-5501-5`.

Andrew Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE transactions on Information Theory*, 13 (2):260–269, 1967.