Revisiting the finite vocabulary

In the first note, we discussed the following description of a probability distribution. Let $\mathcal{V}$ be a finite vocabulary, and $w_1, \ldots, w_n$ be a sequence in $\mathcal{V}^*$.[1] Then let p be a probability distribution over $\mathcal{V}^*$; we can write out the probability of a sequence as:

$$p(w_1, \ldots, w_n) = \prod_{i=1}^{n} p(w_i \mid w_{<i}), \tag{1}$$

where $w_{<i}$ is shorthand for $w_1, \ldots, w_{i-1}$. We already saw how estimating the parameters of a simple model of this distribution led to interesting word representations---for the words in $\mathcal{V}$.

In this note, we again consider the problem of representing sequences of text, but ask **how do we choose the elements/words of $\mathcal{V}$?** and further, **how does it make sense to have a finite vocabulary in the first place?** These questions correspond to the notion of **tokenization** -- mapping from strings of text to discrete *tokens* from our finite vocabulary (and back).

As we will see, the answers to these questions are (1) we estimate $\mathcal{V}$ from data (not shocking) and (2) we sort of fudge the finite vocabulary assumption by filling that finite vocabulary with **pieces of words, or short chunks of text, that compose to form (larger) words or chunks.**

Problems with a finite vocabulary

Let's say you've defined a vocabulary $\mathcal{V}$, say, by taking a document, splitting on whitespace, and taking every unique word in the document. Here, there's even a quick unix command for this!

```
cat file.txt | tr ' ' '\n' | sort | uniq > vocab.txt
```

Imagine this is the whole text of the file. We could take it and define a vocabulary from it:

```
I record the record on the tape deck.
```

and encode it as a sequence of integers corresponding to the indices of the words in the finite vocabulary (I've made up an ordering here):

```
[4, 0, 1, 0, 2, 1, 3, 5]
```

So, here, I have 6 words in $\mathcal{V}$. In the ordering, `I` is 4, `record` is 0... you may see issues here already. For example, the two instances of the string `record` are actually different words with the same string. There are some odd results from our choice to split on whitespace as well, for example, we have a word `deck.` which includes the trailing period. These aren't the only (subtle?) issues.[2] But **the key issue arises as soon as I try to encode a new file using this encoding**:

```
I love to record records.
```

In trying to convert this text to our finite vocabulary, I see `I` and map it to 4 as before, but then I see `love` and I'm stuck -- it's not an element of $\mathcal{V}$. In practice, vocabularies are much larger---think tens or hundreds of thousands---but the problem remains the same. It's just impossible to enumerate a finite set of words that is future-proof. There

[1] The star in $\mathcal{V}^*$ is the Kleene star; $\mathcal{V}^*$ is the set of all finite-length strings composed of elements of $\mathcal{V}$.

[2] What happens if I have multiple space characters in a row?

are always string of characters you missed, or that will appear for the first time after you fixed $\mathcal{V}$.

Possible solutions for the finite vocabulary problem

In the past in NLP, many techniques have been developed for trying to deal with this **out-of-vocabulary** problem. You could map every word not in $\mathcal{V}$ to some special ``unknown'' word. This is lossy -- it's impossible to reconstruct the original document, so this isn't great. You could say that really we should just have each *latin character* be an element of $\mathcal{V}$, leading to a vocabulary of 27 (26 letters and space.) Naturally, this is also lossy; it's missing all kinds of characters, capital letters, etc.... but it's getting at the idea that by defining our vocabulary to have small units that compose into larger ones, we can represent a lot of strings.

If we take the ``small units'' hypothesis to the extreme, one could say, isn't each string just a sequence of bytes? How many values can a byte take on? That's 8 bits, so $2^8 = 256$ values. Why don't we just define a vocabulary of bytes? This is a pretty interesting idea, and is actively being explored in NLP research, but it isn't the dominant paradigm. The short reason for why is that it seems to be really useful to have *some* much larger units of text in your vocabulary, both because (1) it makes the encoded sequences shorter, and (2) it seems to be a useful learning starting point to have word-like objects, whereas learning directly from bytes is harder.

Finally, you could consider trying to get rid of the finite vocabulary assumption entirely... intuitively it's hard to figure out exactly what this would mean, but hey nothing has worked so far. One idea for getting rid of the finite vocabulary is to model text as *pixels in an image*, and try to generate pixel by pixel.[3] This is actually also an active area of research! But has similar drawbacks... hard to learn, and long sequences.

The subword solution for finite vocabularies

The key idea in defining a useful finite vocabulary is to **have each element in the vocabulary correspond to a chunk of text which can combine with other chunks to form (larger) words**, and **estimate these chunks from data, so that common long chunks get their own place in the vocabulary**.

Let's start by looking at an example:

```
I want to go to the beeaaaach that would be the bestest undoubtedly
```

this text might be tokenized as

```
[I, _want, _to, _go, _to, _the, _bee, aaa, ach,
 _that, would, _be, _the, _best, est, _undoubtedly]
```

I've done a few things here; first, I've written words that are preceded by a space as tokens with a preceding underscore. Note that for `beeaaaach`, only the `_bee` token has an underscore. the token `bee` (no underscore) would be a different element of the vocabulary. The other thing I've done here is split the two odd/rare words into mulitple pieces. This means that the vocab I chose doesn't have `beeaaaach` or `bestest`, but it does have all the component subwords that you see here. The word `_undoubtedly`, however, is very long, but it did show up in my vocab, so it doesn't get split up.

This demonstrates the intuition of subword vocabularies. Many (common) words will

[3] Though this gets into definition questions, i.e., is this just modeling text as a very long sequence of elements from a finite RGB pixel value vocabulary, etc.

show up in the vocabulary. For many (uncommon) words, they will be split into multiple pieces. In the extreme, we fall back on splitting a word into its characters (or even bytes.)

The Byte-Pair Encoding Algorithm

We now consider how to estimate subword vocabularies. The intuition to learn here is that we want common chunks of text to correspond to their own elements in the vocabulary, so that we break texts into roughly as few chunks as possible.

There are many methods for doing this in modern NLP, but we'll discuss a simple algorithm called **byte-pair encoding** (BPE.) Intuitively, BPE begins with a small character-level vocabulary and iteratively (1) encodes a corpus with the existing vocabulary, (2) counts adjacent elements to check which pair of elements show up next to each other most, (3) concatenates this most-commonly adjacent pair of elements into one string and adds it to the vocabulary. Taking an example, we might start with

```
the hearth heist is
```

encode this as

```
t,h,e, ,h,e,a,r,t,h, ,h,e,i,s,t, ,i,s
```

derive counts:

```
(t, h) -> 2
(h, e) -> 3
(e, ' ') -> 1
(' ', h) -> 2
(e, a) -> 1
(a, r) -> 1
(r, t) -> 1
(h, ' ') -> 1
(e, i) -> 1
(i, s) -> 2
(s, t) -> 1
(' ', i) -> 1
```

take the max---`(h, e)`---and concatenate them---`he`. Add it to the vocabulary, and re-encode:

```
t,he, ,he,a,r,t,h, ,he,i,s,t, ,i,s
```

and recompute counts

```
(t, he) -> 1
(he, ' ') -> 1
(' ', he) -> 1
(he, a) -> 1
(a, r) -> 1
(r, t) -> 1
(t, h) -> 1
(he, i) -> 1
(i, s) -> 2
(s, t) -> 1
(t, ' ') -> 1
(' ', i) -> 1
```

And so the max pair is `(i,s)`, so we add `is` to our vocabulary and continue building larger units until we hit our maximum desired vocabulary size.

Now let's discuss the algorithm somewhat more formally.

Algorithm. The input to our algorithm is a (body of) text D and a target vocabulary size k. Fix an initial vocabulary $\mathcal{V}_0$ by taking all characters in D, (i.e., unicode characters, ascii characters, or even bytes.) Let D_0 be the encoding of D into a sequence of elements from $\mathcal{V}_0$. For concreteness, the encoding algorithm greedily iteratively computes the longest prefix of the (remaining) text that is an element of the vocabulary, yields the corresponding token, removes that prefix from the text, and continues until the remaining text is empty.[4] Compute the counts of all pairs of adjacent elements in D_0, that is, build a dictionary $\{(w_1, w_2) : \text{count}(w_1, w_2; D_0) \mid w_1, w_2 \in \mathcal{V}_0\}$. Let (w, w') be a pair of elements with maximum count in D_0.[5] Add the concatenation of w, w', that is, ww', to the vocabulary to make $\mathcal{V}_1$:

$$\mathcal{V}_1 = \mathcal{V}_0 \cup \{ww'\} \tag{2}$$

Re-encode D using the vocbulary $\mathcal{V}_1$ to construct D_1. Re-compute adjacency counts using D_1, and repeat. If at any point some iteration of vocabulary $\mathcal{V}_i$ has size $|\mathcal{V}_i| = k$, terminate and return $\mathcal{V}_i$.

Once we've defined this tokenization, in order to use it gainfully in neural networks, we represent each token in $\mathcal{V}$ as a single integer identifier.

Example. As a quick example of a tokenized string, we might look at a string

```
Zuko made his uncle tea.
```

printing each token string, we get

```
['Z', 'uko', ' made', ' his', ' uncle', ' tea', '.']
```

and the corresponding integer sequence (which is arbitrary but must be the same whenever we use the encoding) could be

```
[236953, 39325, 1603, 914, 39143, 11115, 236761]
```

Problems in subword tokenization

Language imbalance. If we think of subword tokenization as a compression strategy---represent common substrings as atomic identifiers---then what text will it do a good job of compressing? In general, the answer is text that looks like the corpus we trained the subword tokenizer on. What that means for us is, if our training corpus was mostly English say, then documents that look like an English news article will be split into *few, large* subword tokens (per byte), while documents that look like a Tamil article will be split into *many, small* subword tokens (per byte.) Even more concretely, the English word *appreciate* may be one subword, while the tamil பாராட்டுகிறேன் may be many. This is true even if you have a lot (in an absolute sense) of Tamil data, because once you fix a vocabulary size, say, $52,256$, then different languages must in some sense compete for those vocabulary slots, and the languages with more data will get more slots. Longer sequences of subwords have a few issues, including (1) models tend to perform worse on such sequences; intuitively, longer-distance reasoning over long sequences is hard. And

[4] Note that this algorithm does not guarantee that you'll get the optimal (shortest) encoding of the text under your vocabulary. Why? Feel free to reach out to me about it.

[5] May not be unique; it doesn't matter, so pick any such one.

(2) as we'll see later in the class, long sequences have high computational cost; often scaling as $\mathcal{O}(n^2)$ where n is the sequence length.

Number representation. Subword tokenization doesn't naturally lead to strong representation of numbers. Put another way, subword tokenization is a reasonable compression scheme for language semantics, but not so much for mathematical semantics. For example, if I want to compute $100,000 + 1$, it's pretty odd to have the following tokenization:

```
['$', '100', ',', '000', ' +', ' 1', ' =', ' 100', ',', '101']
```

The semantics of numbers---the properties of addition, multiplication, etc. and how they are represented in text---aren't really well-preserved by tokenization. A simple way to (possibly) improve the situation is to always tokenize each digit, i.e., tokenizing 100 as `['1', '0', '0']` so that when we add `1` we just get a single token change. However, this has its own issues; for example, you maybe do want to represent, e.g., the year 1970[6] as its own token `1970`, and we always have to compete with exploding sequence lengths if we tokenize into too-small parts.

Unicode and bytes

So far in this note we've assumed the notion of a *character* over which we're building our subword vocabulary. But what is a character? This has important practical considerations for the rich, multilingual, symbol-heavy text distributions that we'll be learning from on the web.

Using unicode. In Python3 for example, all strings are unicode strings. A character in a python string is a unicode code point, which might be a latin character like `p` or a Tamil ப. In terms of bytes, however, in UTF-8, `p` is the sequence `(112,)` whereas ப is a 3-byte sequence `(224, 174, 170)`. So, already unicode is implicitly choosing a compression strategy for us. Can we just use unicode codepoints as characters for our tokenizer?

Well, no, but not using it could also lead to problems. The issue with using unicode codepoints is that there are **a lot**, so we have to either (1) include all of them in the seed character set in BPE (see our algorithm description above) or (2) face the possibility that we'll see some text once our tokenizer is trained that we're unable to tokenize.[7] For example, if I never saw ப when training my tokenizer, but a user specifies the character, my system is unable to represent it properly. If we include all unicode code points in our character set, well, there are `154,998` characters in the 16th Unicode standard[8], and our target total vocabulary size was *probably* between 50,000 and 200,000, so this isn't ideal. We want most of our token slots used by common multi-character strings.

Using bytes. We often instead use bytes (resulting from the unicode encoding) as the fundamental character for BPE. So, when we initialize our BPE with all characters, we just start with all byte valyes. A byte is 8 bits, so we start with the characters `(0,)` through `(255,)`. Now, when we see a byte string that we'd like to tokenize, we can **always** represent that string under our tokenization, since every individual byte is a token in our vocabulary.

Now, when learning our tokenizer, the Tamil ப, won't be a single character; it's just the

[6] The year of the Linux epoch.

[7] What do you do if you see text you can't tokenize? Maybe you use an ``out-of-vocabulary'' special symbol, but subword tokenization was supposed to avoid this! Now you can't reconstruct the original text from the original text.

[8]

byte sequence `(224, 174, 170)`. Our tokenizer might learn a token for the whole 3-byte sequence, or say just `(174, 170)`. One issue with this strategy is that not all byte sequences are valid unicode, and we're defining a probability distribution over strings of *bytes*, not strings of *unicode codepoints*. So, our resulting language model might generate a byte sequence that doesn't correspond to valid unicode! In some sense, this is the opposite of the problem we had for unicode; instead of being unable to generate some strings (if we don't use all codepoints as characters,) we're now able to generate ``too much,'' or byte sequences that represent nothing at all.

Here's an example from a tokenizer I trained. Here's a sentence:

```
Nice!
```

Its token sequence (written out as their bytes) is:

```
[(78, 105, 99), (101, 33)]
```

We can print the individual tokens by decoding them each individually to their strings of unicode code points:

```
['Nic', 'e!']
```

But when I try the same with தமிழ், I get:

```
[(224, 174), (164,), (224, 174), (174,), (224, 174, 191),
(224, 174), (180,), (224, 174), (176,), (224, 175, 141)]
```

and then each of these tokens isn't necessarily a valid unicode code point in itself, so if I try to print out the string for each token (without constructing the whole bytestring) I get unicode errors which alas are hard to print here, but you get the gist:

```
>>> [bytes(x).decode('utf-8', errors='ignore') for x in tokenize(tamil_text)]
['', '', '', '', ' ', '', '', '', '', '']
```

This shows that if I were to not generate the right sequence of tokens, I could also end up generating invalid unicode.

Overall, however, the byte-level option is best; we train models to generate byte sequences that are valid unicode, and they usually do. Even if they don't, it's usually the case that *most* of the bytestring is valid unicode, so if the model makes a small error somewhere it's bad but it doesn't necessarily break the whole string.

How big do we make the subword vocabulary?

A larger vocabulary incurs a cost in computing probabilities per token---we have to compute the probability of every token as the possible next token---and also means that the rarest tokens will be less frequent (and so perhaps less well-handled by our language model.) A smaller vocabulary splits texts into longer sequences of tokens, which is bad for the scaling of our language models. In general, this is an empirical question handled differently by each model GPT-2 (from OpenAI), released in 2019, had 50,257 tokens. Roughly the trend has been towards larger vocabularies; GPT-4o (from OpenAI) apparently has roughly 200,000, and recent (2025) Gemma models (from Google DeepMind) have roughly 256,000.