(https://github.com/everdark/k9/tree/master/notebooks/ml/natural_language_understanding/subword_units)

On Subword Units

CODE

Segmentation for Natural Language Modeling

Kyle Chung

22 Nov 2019 Last Updated (04 Aug 2019 First Uploaded)

Abstract

Langauge segmentation or tokenization is the very first step toward a natural language understanding (NLU) task. The state-of-the-art NLU model usually involves neural networks with word embeddings as the workhorse to encode raw text onto vector space. A fixed vocabulary is pre-determined in order to facilitate such setup. With the challenge of out-of-vocabulary issue and non-whitespace-delimited language, we use subword units to further decompose raw texts into substrings. In this notebook we summarize the technique of subword segmentation in details with Python coding examples. We also provide a general usage walk-through for Google's open source library *SentencePiece*, a powerful language-agnostic unsupervised subword segmentation tool.

1 Subword Units

Neural network models use a fixed size vocabulary to handle natrual language. When the potentail vocabulary space is huge, especially for a neural machine translation (NMT) task, there will be too many unknown words to a model.

To deal with such challenge, Sennrich, Haddow, and Birch (2015) propose the idea to break up rare words into subword units for neural network modeling. The first segmentation approach is inspired by the *byte pair encoding* (https://en.wikipedia.org/wiki/Byte_pair_encoding) compression algorithm, or BPE in short.

1.1 Byte Pair Encoding

BPE is originally a data compression algorithm that iteratively replaces the most frequent pair of bytes in a sequence with single unused byte. By maintaining a mapping table of the new byte and the replaced old bytes, we can recover the original message from a compressed representation by reversing the encoding

process.

The same idea can be applied at charater-level instead of byte-level on a given corpus, effectively constitute a subword segmentation algorithm.

The procedure to apply BPE to obtain subwords can be summarized as the followings:

1. Extract word-level vocabulary as the initial vocabulary
2. Represent the initial vocabulary at character-level for each word, served as the working vocabulary
3. [Pairing] For each word in the working vocabulary, do character-level BPE and out of all words extract the most frequent pair concatenated as a new subword added into the subword vocabulary
4. [Merging] Replace the most frequent pair with its single concatenated version in the working vocabulary
5. Repeat step 3-4 for a given number of times, depending on a desirable subword vocabulary size
6. The final vocabulary is the initial fullword vocabulary plus the subword vocabulary

Based on the procedure, essentially we do not consider paris across word boundaries to be a potential subword. Number of merge operations is the only hyperparameter for the algorithm.

Since the most frequent subwords will be merged early, common words will remain as one unique symbol in the vocabulary, leaving out rare words splitted into smaller units (subwords).

Here is a toy implementation of BPE applied on a given corpus:

HIDE

```
import re
from collections import defaultdict



class BPE:

  def __init__(self, sents, min_cnt=3, verbose=False):
    self.verbose = verbose
    init_vocab = defaultdict(int)
    for sent in sents:
      words = re.split(r"\W+", sent)
      for w in words:
        if w != "":
          init_vocab[w] += 1
    # Create fullword vocabulary.
    self.word_vocab = {k: v for k, v in init_vocab.items() if v >= min_cnt}
    # Insert space between each char in a word for latter ease of merge operation.
    # We directly borrow the idea from https://www.aclweb.org/anthology/P16-1162.
    self.working_vocab = {" ".join(k): v for k, v in self.word_vocab.items()}
    self.subword_vocab = defaultdict(int)
    # Also build a character-level vocabulary as the base subwords.
    self.char_vocab = defaultdict(int)
    for sent in sents:
      for char in list(sent):
        self.char_vocab[char] += 1

  def _find_top_subword(self):
    subword_pairs = defaultdict(int)
    for w, cnt in self.working_vocab.items():
      subw = w.split()
      for i in range(len(subw) - 1):
        # Count bigrams.
        subword_pairs[subw[i], subw[i+1]] += cnt
    top_subw_pair = max(subword_pairs, key=subword_pairs.get)
    top_subw = "".join(top_subw_pair)
    self.subword_vocab[top_subw] = subword_pairs[top_subw_pair]
    if self.verbose:
      print("New subword added: {}".format(top_subw))
    return top_subw_pair

  def _merge(self, subw_pair):
    bigram = re.escape(" ".join(subw_pair))
    p = re.compile(r"(?<!\S)" + bigram + r"(?!\S)")
    self.working_vocab = {p.sub("".join(subw_pair), w): cnt for w, cnt in self.working_vocab.ite
          ms()}

  def update_subword(self, n_merge=1):
```

```
    for n in range(n_merge):
      top_subw_pair = self._find_top_subword()
      self._merge(top_subw_pair)
```

Let's use Shakespeare as the input text example to run the BPE algorithm. First we download the data from Google's public hosting service:

HIDE

```
# bash
mkdir -p data
```

HIDE

```
import warnings
warnings.simplefilter(action="ignore", category=FutureWarning)

import os
import shutil
import tensorflow as tf

shakes_file = "data/shakespeare.txt"
if not os.path.exists(shakes_file):
  shakes_dl_path = tf.keras.utils.get_file(
    "shakespeare.txt",
    "https://storage.googleapis.com/download.tensorflow.org/data/shakespeare.txt")
  shutil.move(shakes_dl_path, shakes_file)
shakespeare = open(shakes_file, "rb").read().decode(encoding="utf-8")
shakespeare = shakespeare.lower().split("\n")

# Print the first few lines.
for sent in shakespeare[:20]:
  print(sent)
```

```
first citizen:
before we proceed any further, hear me speak.

all:
speak, speak.

first citizen:
you are all resolved rather to die than to famish?

all:
resolved. resolved.

first citizen:
first, you know caius marcius is chief enemy to the people.

all:
we know't, we know't.

first citizen:
let us kill him, and we'll have corn at our own price.
```

HIDE

```
bpe = BPE(shakespeare, min_cnt=10)
print(len(bpe.word_vocab))
```

```
1872
```

HIDE

```
print(list(bpe.word_vocab.items())[:5])  # Print some from fullword vocabulary.
```

```
[('first', 363), ('citizen', 100), ('before', 195), ('we', 938), ('proceed', 21)]
```

HIDE

```
print(list(bpe.working_vocab.items())[:5])  # (For debug) Print some from the working vocab that
          we are going to perform the merge.
```

```
[('f i r s t', 363), ('c i t i z e n', 100), ('b e f o r e', 195), ('w e', 938), ('p r o c e e
d', 21)]
```

HIDE

```
bpe.update_subword(n_merge=100)  # Do merge update.
print(len(bpe.subword_vocab))
```

```
100
```

HIDE

```
print(list(bpe.working_vocab.items())[:5])  # Check the working vocabulary after merge.
```

```
[('f ir st', 363), ('c it i z en', 100), ('be for e', 195), ('we', 938), ('p ro ce ed', 21)]
```

HIDE

```
print(list(bpe.subword_vocab.items())[:5])  # Print some subwords generated by the first 100 merg
          e operations.
```

```
[('th', 25186), ('ou', 11960), ('the', 11654), ('an', 11587), ('in', 9012)]
```

Each learned subword itself doesn't need to have explicit meaning. It is their potential combinations with other subwords that reveal the actual context, which can be hopefully learned by neural network embeddings for the downstream task.

A modification of the BPE algorithm is to expand subword vocabulary by likelihood instead of the most frequent pair. That is, in each merge operation we choose the new subword that increases the likelihood of the training data the most.

1.2 Probablistic Subword Segmentation

Note that BPE is *deterministic* by a greedy replacement of adjacent symbols, while the actual subword segmentation is potentially ambiguous. A sequence can be represented in multiple subword sequences since even the same word can be segmented differently by different subwords.

Kudo (2018) propose an approach called "subword regularization" to handle this issue in a probablistic fashion. A new subword segmentation under this approach uses a *unigram language model* for generating subword segmentation with attached probability.

Denote X as a subword sequence of length n

$$X = (x_1 \quad x_2 \quad \dots \quad x_n) ,$$

with *independent* subword probability $P(x_i)$. The probability of the sequence hence is simply

$$P(X) = \prod_{i=1}^{n} P(x_i),$$

with

$$\sum_{x \in V} P(x) = 1,$$

where V is a pre-determined vocabulary set.

The most probable sequence (from all possible segmentation candidates) can be found by applying expectation-maximization (https://en.wikipedia.org/wiki/Expectation%E2%80%93maximization_algorithm) with the Viterbi algorithm (https://en.wikipedia.org/wiki/Viterbi_algorithm), which is designed for finding the most likely sequence of hidden states. Here the hidden states are subwords used in a sequence.

The vocabulary set V is indeed undertermined at the very beginning and cannot be solved simultaneously with the maximization task. A workaround is to provide a seed vocabulary as the initial (reasonably big enough) vocabulary, and shrink the seed vocabulary during training until a desired vocabulary size is reached.

To fully understand the unigram language model for segmentation, let's examine the expectation-maximization and Viterbi algorithm accordingly.

1.2.1 Expectation-Maximization

Quoted from wiki page of EM (https://en.wikipedia.org/wiki/Expectation%E2%80%93maximization_algorithm):

> In statistics, an expectation–maximization (EM) algorithm is an iterative method to find maximum likelihood or maximum a posteriori (MAP) estimates of parameters in statistical models, where the model depends on unobserved latent variables.

It's better to understand the EM with a working example.

We use a working example of a Gaussian Mixture to illustrate the core idea of EM. A Gaussian Mixture model is more of a Hello-World example of EM. In its simplest illustration, assume we observe a set of data resulting from either one of two different Gaussians, how can we estimate parameters of these two Gaussians without knowing which Gaussian they came from?[1]

First create some toy data:

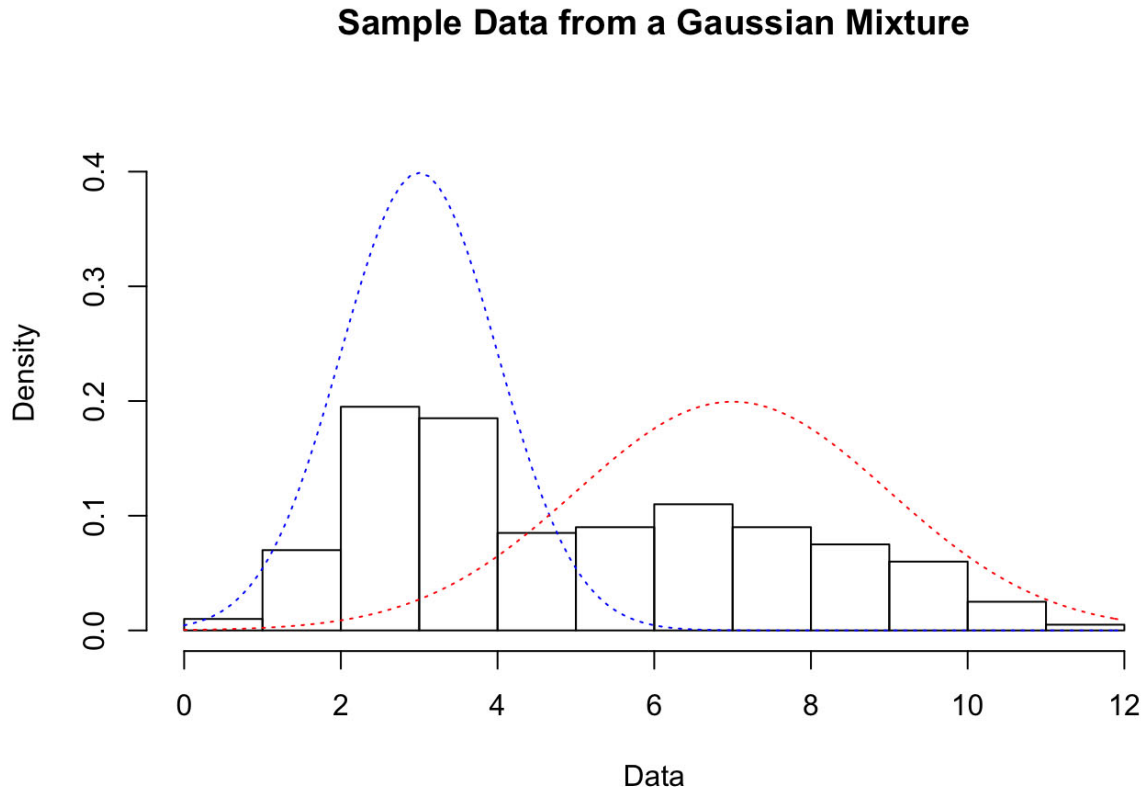
HIDE

```
import numpy as np

np.random.seed(777)
size = 100

g1_mean = 3
g2_mean = 7
g1_std = 1
g2_std = 2

g1_data = np.random.normal(g1_mean, g1_std, size=size)
g2_data = np.random.normal(g2_mean, g2_std, size=size)
mixture_data = np.concatenate([g1_data, g2_data])
```

The following plot illustrates our toy data distribution. The resulting distribution is a equal combination of two Gaussians plotted as dotted curves.

HIDE

```
# R
x <- py$mixture_data
hist(x, xlab="Data", main="Sample Data from a Gaussian Mixture",
     probability=TRUE, ylim=c(0, .45))
curve(dnorm(x, mean=py$g1_mean, sd=py$g1_std),
      col="blue", lty="dotted", add=TRUE, yaxt="n")
curve(dnorm(x, mean=py$g2_mean, sd=py$g2_std),
      col="red", lty="dotted", add=TRUE, yaxt="n")
```



To formulate th EM algorithm, we first identify what is the hidden state (unobserved/latent variable) in a given modeling task. In the Gaussain Mixture example, the hidden state (for each observed data point) is the indicator of the original Gaussian that generats the data.

The EM algorithm contains two step in one iteration:

1. [Expectation Step] Formulate the *expected* maximum likelihood based on a probabilistic distribution of hidden state; that is, an expected maximum likelihood estimator (MLE) conditional on hidden states
2. [Maximization Step] Figure out model parameters by maximizing the expected MLE

The solution to model parameters can be found (without even knowing the hidden state but only its expectation) after serveral iterations.

Here is a simple EM implementation for a Guassian Mixture model:

HIDE

```
from scipy import stats



class GaussianMixtureEM:

  def __init__(self, data):
    self.data = data
    # Initialize hidden state probabilistic weight.
    # Note that there is no guarantee that g1 will correspond to the first half of the data.
    # The actual estimated assignment is determined by the final weights.
    self.g1_weights = np.random.uniform(size=len(data))
    self.g2_weights = 1 - self.g1_weights
    # Initialize parameter guessings.
    self.update_estimates()

  def update_estimates(self):
    self.g1_mean = self.estimate_mean(self.g1_weights)
    self.g2_mean = self.estimate_mean(self.g2_weights)
    self.g1_std = self.estimate_std(self.g1_mean, self.g1_weights)
    self.g2_std = self.estimate_std(self.g2_mean, self.g2_weights)

  def estimate_mean(self, weights):
      return np.sum(self.data * weights) / np.sum(weights)

  def estimate_std(self, mean, weights):
      variance = np.sum(weights * (self.data - mean)**2) / np.sum(weights)
      return np.sqrt(variance)

  def em(self, n_iter=10):
    for n in range(n_iter):
      g1_lik = stats.norm(self.g1_mean, self.g1_std).pdf(self.data)
      g2_lik = stats.norm(self.g2_mean, self.g2_std).pdf(self.data)
      self.g1_weights = g1_lik / (g1_lik + g2_lik)
      self.g2_weights = g2_lik / (g1_lik + g2_lik)
      self.update_estimates()


mix_model = GaussianMixtureEM(mixture_data)
mix_model.em(30)
```

HIDE

```
true_params = (g1_mean, g2_mean, g1_std, g2_std)
mle_params = (
  np.mean(mixture_data[:size]),
  np.mean(mixture_data[size:]),
  np.std(mixture_data[:size]),
  np.std(mixture_data[size:])
)
# To ensure the ordering so we can have a nicely printed comparison.
em_params = np.concatenate([
  np.sort([mix_model.g1_mean, mix_model.g2_mean]),
  np.sort([mix_model.g1_std, mix_model.g2_std])
])



for i, (t, m, e) in enumerate(zip(true_params, mle_params, em_params)):
  if i == 0:
    print("True Value | MLE (Known Hidden States) | EM (Unknown Hidden States)")
  print("{:^11}|{:^27}|{:^27}".format(t, np.round(m, 4), np.round(e, 4)))
```

```
True Value | MLE (Known Hidden States) | EM (Unknown Hidden States)
     3     |          3.0071           |          2.9194
     7     |          7.1051           |          7.0672
     1     |          0.9586           |          0.9047
     2     |          1.9048           |          1.8579
```

As one can see, we are able to properly estimate the model parameters (in this case the first and second moments for two different Gaussians) without knowing the hidden variables (the indicator of which data point comes from which Gaussian).

1.2.2 Viterbi Algorithm

To apply EM to our subword segmentation problem, we can use the Viterbi algorithm in the maximization step to find out the most probable sequence of subwords.

Let's use the Shakespeare text to illustrate the Viterbi algorithm. For a given text line we'd like to calculate the most probable subwords that can recover the text line. In order to achieve that we need to have a vocabulary for subwords that can attribute each subword to a probability. We can use BPE to build such vocabulary. For complete coverage we will also include character-level subwords into the vocabulary.

HIDE

```
# Build a seeding subword vocabulary using BPE.
# For illustration purpose we didn't go for a huge seeding size.
# In actual application the seeding vocabulary should be much larger than what we have here.
bpe = BPE(shakespeare, min_cnt=10)
bpe.update_subword(n_merge=1000)
subword_cnt = {**bpe.char_vocab, **bpe.subword_vocab}
tot_cnt = np.sum(list(subword_cnt.values()))

# The initial estimate of subword probability.
subword_logp = {k: np.log(v / tot_cnt) for k, v in subword_cnt.items()}
```

A Viterbi procedure contains two steps: a forward step and a backward step. The forward step is to determine the most probable subword given each end-of-character position in a word. The backward step is to recover the entire subword sequence that maximizes the likelihood of the sequence.

The Forward Step

HIDE

```
def viterbi_forward(word, subword_logp):
  """Forward step of Viterbi given a single word."""
  # Create storage array for best substring recorded at each end-of-character position.
  best_subw_slices =  [None] * (len(word) + 1)
  neg_loglik = np.zeros(len(word) + 1)
  # Search the best substring given every possible end of position along the word.
  for eow in range(1, len(word) + 1):
    # For every end-of-word position:
    neg_loglik[eow] = np.inf
    for bow in range(eow):
      # For every possible beginning-of-word position given the end-of-word position:
      subw = line[bow:eow]
      if subw in subword_logp:
        logp = subword_logp[subw]
        # Compute subword probability:
        # P(current segment) + P(the best segment just before the current segment).
        s = neg_loglik[bow] - logp
        if s < neg_loglik[eow]:
          neg_loglik[eow] = s
          best_subw_slices[eow] = (bow, eow)
  return neg_loglik, best_subw_slices
```

We demonstrate the segmentation given a single fullword:

HIDE

```
# Take a text line from Shakespeare.
line = shakespeare[213]
words = line.split()

w = words[0]
print(w)  # Word for segmentation demo.
```

```
whereby
```

HIDE

```
subw_scores, subw_slices = viterbi_forward(w, subword_logp)
print(subw_slices)  # Best segment indices at each subword end.
```

```
[None, (0, 1), (0, 2), (2, 3), (2, 4), (0, 5), (5, 6), (5, 7)]
```

HIDE

```
subw = [w[idx[0]:idx[1]] for idx in subw_slices if idx is not None]
print(subw)  # Best segmented subword at each subword end.
```

```
['w', 'wh', 'e', 'er', 'where', 'b', 'by']
```

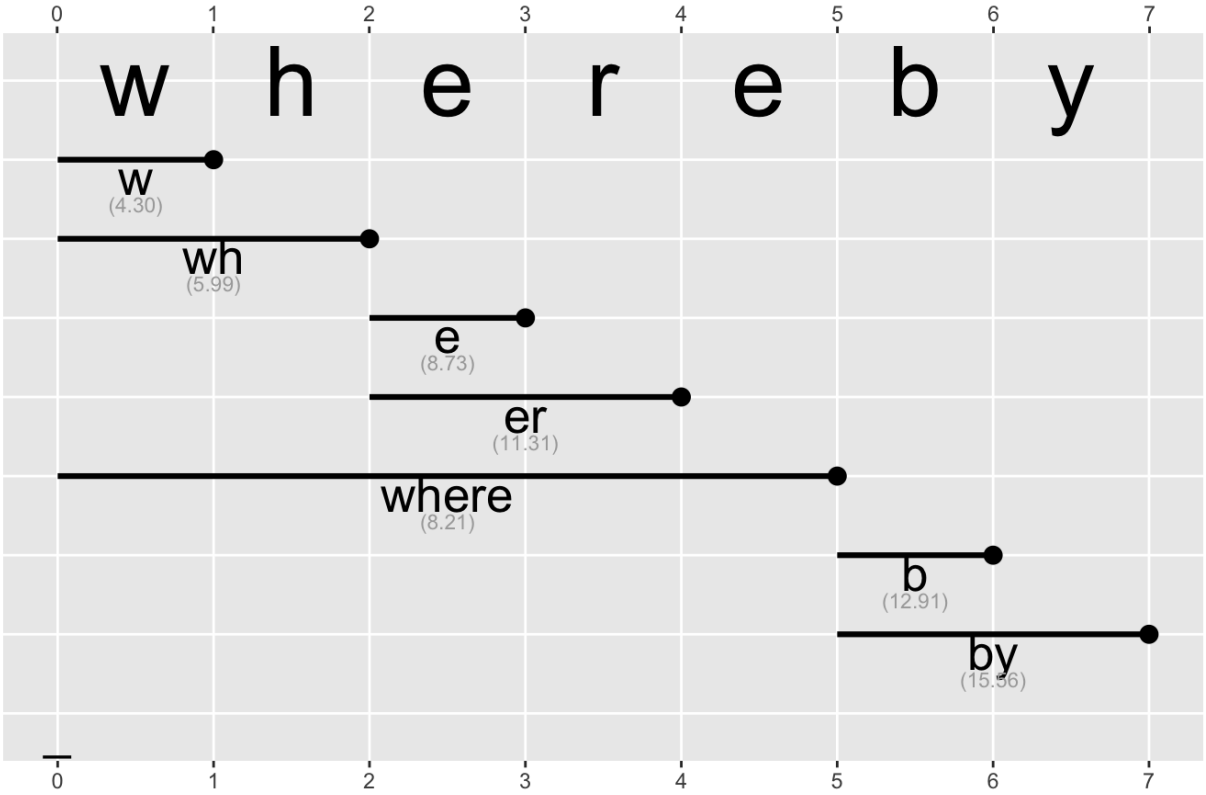
Let's visualize the segmentation!

The number beneath the a subword is the corresponding negative log-likelihood for that subword up to the corresonding end-of-character position. The lower the better.

HIDE

Hide Source

```
# R
library(ggplot2)
library(data.table)

subw_demo <- rbindlist(py$subw_slices)
setnames(subw_demo, c("start", "end"))
subw_demo <- subw_demo[, subw:=py$subw]
subw_demo <- rbind(list(0, 0, ""), subw_demo)  # To plot one more block at the top.
subw_demo <- rbind(subw_demo, list(0, 0, "_"))  # To plot one more block at the bottom.
subw_demo <- subw_demo[, subw:=factor(subw, levels=rev(subw))]
subw_demo <- subw_demo[, score:=c(py$subw_scores, 0)]

# To also plot the entire word by character.
w_df <- data.table(x=rep("", nrow(subw_demo) - 2), y=0:(nrow(subw_demo)-3) + .5,
                   label=unlist(strsplit(py$w, NULL)))

viterbi_demo_p <- ggplot(subw_demo, aes(x=subw, y=(start + end) / 2, label=subw)) +
  geom_linerange(aes(x=subw, ymin=start, ymax=end), size=1.1) +
  geom_text(vjust=1, size=7) +
  geom_text(data=subw_demo[2:(nrow(subw_demo) - 1)],
            aes(label=sprintf("(%.2f)", score)), vjust=3.5, size=3, color="darkgrey") +
  geom_text(data=w_df, aes(x=x, y=y, label=label), size=14) +
  geom_point(aes(x=subw, y=end), data=subw_demo[2:(nrow(subw_demo) - 1)], size=3) +
  scale_y_continuous("", breaks=0:(nrow(subw_demo)-2), sec.axis=dup_axis()) +
  theme(axis.title.y=element_blank(),
        axis.text.y=element_blank(),
        axis.ticks.y=element_blank(),
        panel.grid.minor=element_blank()) +
  coord_flip()

viterbi_demo_p
```



We iterate over every position in a given word. At each end-of-character position, we determine the best segment by finding the one with highest likelihood given the current vocabulary.

For example, when we are at position 2, we compare the log-liklihood of `(wh)` and `(w, h)`. The former is more likely so we record its negative log-likelihood along with the segment indices and then move onto the next position. Now at position 3 we compare `(whe)` with `(w, he)` and `(wh, e)` and again return the best segment. Notice that we don't compare `(w, h, e)` since in the previous position we already rule `(w, h)` out due to its being inferior (less likely) to `(wh)`.

The Backward Step

After the forward step is finished, we can track backward the path of segments that generates the highest overall likelihood.

HIDE

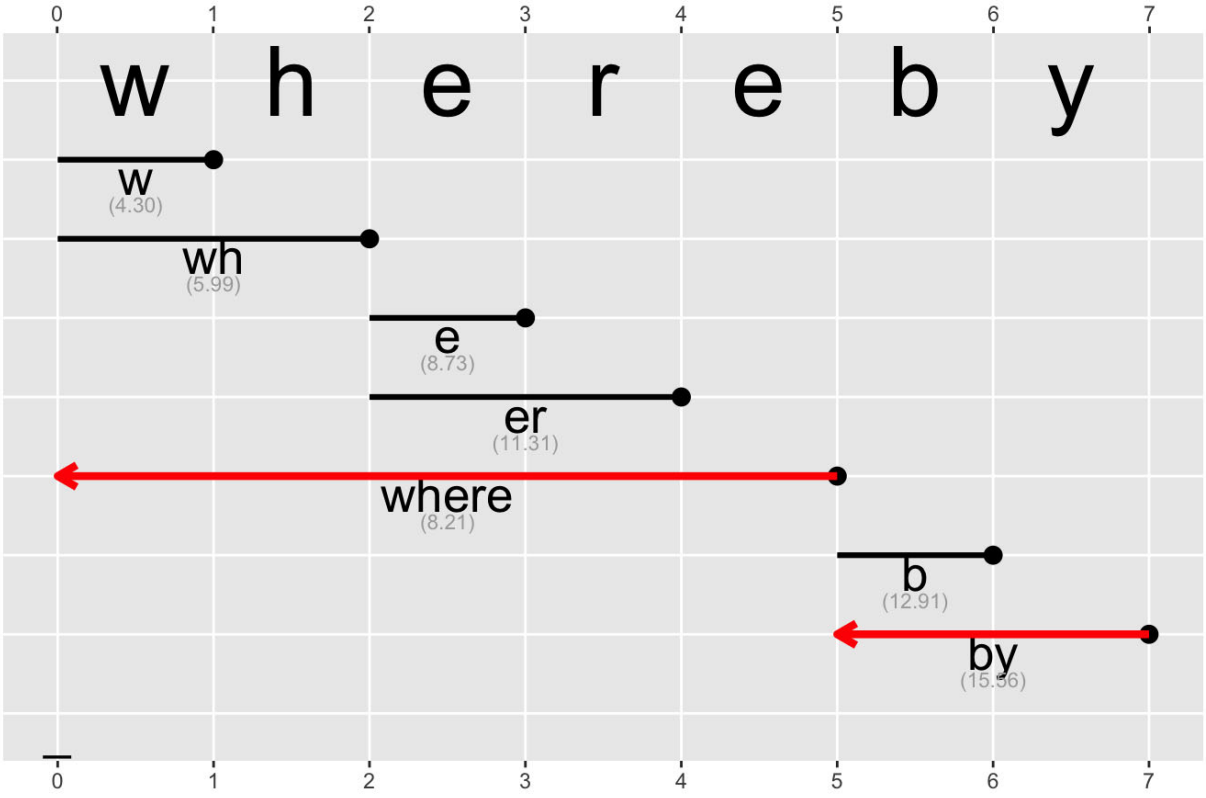
```
def viterbi_backward(word, subw_slices):
  """Backward step of Viterbi to return the best path."""
  subwords = []
  subword_slices = []
  next_slices = subw_slices[-1]
  while next_slices is not None:
    subw = word[next_slices[0]:next_slices[1]]
    subwords.append(subw)
    subword_slices.append(next_slices)
    next_slices = subw_slices[next_slices[0]]
  subwords.reverse()
  return subwords, subword_slices


# Test with a single word.
best_subw, best_slices = viterbi_backward(w, subw_slices)
print(best_subw)
```

```
['where', 'by']
```

This is easily understandable if we also visualize the backward step:

HIDE

```
Show Source
```



Now combining the forward and backward steps for a complete segmentation of a text line:

HIDE

```
def viterbi(line):
  out = []
  words = line.split()
  for w in words:
    subw_scores, subw_slices = viterbi_forward(w, subword_logp)
    subw, subw_idx = viterbi_backward(w, subw_slices)
    out.extend(subw)
  return out

print(line)  # Input sentence.
```

```
whereby they live: and though that all at once,
```

HIDE

```
print(viterbi(line))  # Output subword sequence.
```

```
['where', 'by', 'th', 'ey', 'live:', 'an', 'd', 'thoug', 'h', 'th', 'at', 'al', 'l', 'at', 'onc
e,']
```

1.2.3 EM with Viterbi

Now we know the idea of EM, and we know how to find the optimal segment path by Viterbi, we can put them together to forumlate the optimization task of our probablistic subword segmentation.

Here is the complete procedure:

1. Initialize a large seeding subword vocabulary from the training corpus
2. [Expectation] Estimate each subword probability by the corresponding frequency counts in the vocabulary
3. [Maximization] Use Viterbi to segment the corpus, returning the optimal segments
4. Compute the loss of each new subword from optimal segments
5. Shrink the vocabulary size by dropping the subwords with top X% smallest losses
6. Repeat step 2 to 5 until the vocabulary size reaches a desired number

The loss of a subword in step 4 is the reduction in overall training corpus segment likelihood if that subword is removed from the current vocabulary.

In the above example we use BPE to generate the initial vocabulary. Another common choice is to the suffix array (https://en.wikipedia.org/wiki/Suffix_array) algorithm. to generate the common substrings.

2 SentencePiece

Kudo and Richardson (2018) propose a language-agnostic subword segmentation algorithm called SentencePiece (https://github.com/google/sentencepiece). SentencePiece is a powerful and efficient open-sourced unsupervised langugae tokenizer that can help build end-to-end vocabulary for neural networks. In this section we will walk-through the idea of SentencePiece along with demo examples.

2.1 Model Characteristics

In this section we briefly summarize several useful characteristics about the segmentation model.

2.1.1 Language-Agnostic Handle

SentencePiece is langauge independent becuase under the hood it simply treats a given sentence as a sequence of *Unicode* characters. And most written langauge in the world we speak can be encoded in Unicode. This is particualrly useful for non-whitespace-segmented languages such as Chinese, Korean and Japanese.

Technically speaking, in SentencePiece whitespace is also treated as just another normal symbol. To achieve this, a meta symbol ▁ (U+2581) is used to replace the whitespace at the very beginning.

The entire vocabulary is built from subwords directly based on raw text input so there is also no need to do pre-processing to provide a fullword initial vocabulary. This bypasses the need for any language-specific preprocessing logic, resulting in a unified framework for natural langauge segmentation.

2.1.2 Lossless Tokenization

Because of the special (or rather, non-special) treatment of whitespace as a normal symbol, SentencePiece is able to acheive lossless tokenization. That is, the entire raw text can be recovered from its segmented tokens with zero ambiguity.

The reason why mosty of the other tokenization algorithm won't be able to achieve this is because of the pre-tokenization stage that use whitespace to pre-process the corpus into a working vocabulary. The subword segmentation is then operating on the working vocabulary instead of the raw text lines in the corpus.

Notice that in our toy example in the previous section we also relies on a pre-tokenization before the subword segmentation kicks in.

2.1.3 Self-Contained Integer Mapping

Yet another goodness about SentencePiece is its flexible cabapility to generate the integer mappings that is required for neural network modeling in the embedding lookup subtask. It can handle all the following meta symbols:

- `bos` : beginning of word
- `eos` : ending of word
- `unk` : out-of-vocabulary word
- `pad` : padding for fixed-length output, especially useful in recurrent neural network modeling

Once the model is trained, the mappings are self-contained in the model file and can be used to encode/decode raw text/integer sequence on the fly.

2.2 Command Line Usage

HIDE

```
spm_train --version
```

```
sentencepiece 0.1.82
```

SentencePiece is written in C++ with both a command line tool and a Python wrapper. There are a handful of options that can be used to setup the model configuration.

HIDE

```
# bash
spm_train --help
```

```
sentencepiece

Usage: spm_train [options] files

   --accept_language (comma-separated list of languages this model can accept)  type: string  def
ault:
   --add_dummy_prefix (Add dummy whitespace at the beginning of text)  type: bool  default: true
   --bos_id (Override BOS (<s>) id. Set -1 to disable BOS.)  type: int32  default: 1
   --bos_piece (Override BOS (<s>) piece.)  type: string  default: <s>
   --character_coverage (character coverage to determine the minimum symbols)  type: double  defa
ult: 0.9995
   --control_symbols (comma separated list of control symbols)  type: string  default:
   --eos_id (Override EOS (</s>) id. Set -1 to disable EOS.)  type: int32  default: 2
   --eos_piece (Override EOS (</s>) piece.)  type: string  default: </s>
   --hard_vocab_limit (If set to false, --vocab_size is considered as a soft limit.)  type: bool
default: true
   --input (comma separated list of input sentences)  type: string  default:
   --input_format (Input format. Supported format is `text` or `tsv`.)  type: string  default:
   --input_sentence_size (maximum size of sentences the trainer loads)  type: int32  default: 0
   --max_sentence_length (maximum length of sentence in byte)  type: int32  default: 4192
   --max_sentencepiece_length (maximum length of sentence piece)  type: int32  default: 16
   --model_prefix (output model prefix)  type: string  default:
   --model_type (model algorithm: unigram, bpe, word or char)  type: string  default: unigram
   --normalization_rule_name (Normalization rule name. Choose from nfkc or identity)  type: strin
g  default: nmt_nfkc
   --normalization_rule_tsv (Normalization rule TSV file. )  type: string  default:
   --num_sub_iterations (number of EM sub-iterations)  type: int32  default: 2
   --num_threads (number of threads for training)  type: int32  default: 16
   --pad_id (Override PAD (<pad>) id. Set -1 to disable PAD.)  type: int32  default: -1
   --pad_piece (Override PAD (<pad>) piece.)  type: string  default: <pad>
   --remove_extra_whitespaces (Removes leading, trailing, and duplicate internal whitespace)  typ
e: bool  default: true
   --seed_sentencepiece_size (the size of seed sentencepieces)  type: int32  default: 1000000
   --self_test_sample_size (the size of self test samples)  type: int32  default: 0
   --shrinking_factor (Keeps top shrinking_factor pieces with respect to the loss)  type: double
default: 0.75
   --shuffle_input_sentence (Randomly sample input sentences in advance. Valid when --input_sente
nce_size > 0)  type: bool  default: true
   --split_by_number (split tokens by numbers (0-9))  type: bool  default: true
   --split_by_unicode_script (use Unicode script to split sentence pieces)  type: bool  default:
true
   --split_by_whitespace (use a white space to split sentence pieces)  type: bool  default: true
   --treat_whitespace_as_suffix (treat whitespace marker as suffix instead of prefix.)  type: boo
l  default: false
   --unk_id (Override UNK (<unk>) id.)  type: int32  default: 0
   --unk_piece (Override UNK (<unk>) piece.)  type: string  default: <unk>
   --unk_surface (Dummy surface string for <unk>. In decoding <unk> is decoded to `unk_surface`.)
```

```
type: string  default:  ??
   --use_all_vocab (If set to true, use all tokens as vocab. Valid for word/char models.)  type:
bool  default: false
   --user_defined_symbols (comma separated list of user defined symbols)  type: string  default:
   --vocab_size (vocabulary size)  type: int32  default: 8000
```

SentencePiece supports not only the unigram langauge model we just discussed in the previous section, it also supports BPE algorithm, and two naive tokenizer one for word-level the other for character-level.

Several options are relevant to the unigram model:

- `--seed_sentencepiece_size=1000000` : This is the initial seeding vocabulary for the EM-Viterbi algorithm to start with
- `--shrinking_factor=0.75` : This is by how much the seeding vocabulary is reduced based on the loss-ranking dropout procedure
- `--num_sub_iterations=2` : This is the hyperparameter for the EM-Viterbi algorithm

2.3 Python Usage

The command line tool is very concise. But since we usually train the model for a downstream neural network model that is developed in Python, we will turn our focus into SentencePiece's Python wrapper. It turns out that it is not much different. We will still use the command line option style to control the behavior of the tool.

Let's train a small vocabulary using the Shakespeare:

HIDE

```
import sentencepiece as spm

spm_args = "--input=data/shakespeare.txt"
spm_args += " --model_prefix=m"
spm_args += " --vocab_size=1000"
spm_args += " --model_type=unigram"
spm.SentencePieceTrainer.Train(spm_args)
```

2.3.1 Vocabulary Inspection

The resulting `.model` file comes with a vocabulary file `.vocab` that can be used for investigation. (It is not required for segmentation since the vocab is self-contained in the model file already.)

The `.vocab` file contains all final subwords sorted in descending order for its log-likelihood.

HIDE

```
# R
# We use R here for the sake of a nicely formatted table done by Rmd notebook. :)
vocab <- fread("m.vocab", header=FALSE, encoding="UTF-8")
setnames(vocab, c("subword", "logp"))
vocab <- vocab[, p:=round(exp(logp), 5)]
vocab[]
```

subword <chr>	logp <dbl>	p <dbl>
<unk>	0.00000	1.00000
<s>	0.00000	1.00000
</s>	0.00000	1.00000
,	-3.00909	0.04934
s	-3.37574	0.03419
▁	-3.38395	0.03391
:	-3.65529	0.02585
.	-3.92404	0.01976
d	-4.19285	0.01510
▁I	-4.24765	0.01430

1-10 of 1,000 rows Previous **1** 2 3 4 5 6 ... 100 Next

HIDE

```
# The summation of all subwords should approximate unity.
sum(exp(vocab$logp)[-(1:3)])
```

```
[1] 0.9914569
```

Note the common presence of the special symbol (U+2581) that is used to encode the original whitespace. Usually a subword with this symbol means it is a legitimate fullword, or at least a subword that is the prefix rather than the suffix part of a fullword.

2.3.2 Encoding-Decoding

Once trained, we can load the model with a single line:

HIDE

```
sp = spm.SentencePieceProcessor()
sp.Load("m.model")
```

Here are the general usage of the encoding/decoding APIs:

HIDE

```
subword_pieces = sp.EncodeAsPieces("This is a test.")
print(subword_pieces)
```

```
['▁This', '▁is', '▁a', '▁t', 'est', '.']
```

HIDE

```
print("".join(subword_pieces).replace("\u2581", " "))  # Recover the whitespace.
```

```
 This is a test.
```

HIDE

```
subword_ids = sp.EncodeAsIds("This is a test.")
print(subword_ids)  # Convert sentence to integer sequence.
```

```
[343, 38, 15, 211, 153, 7]
```

HIDE

```
print(sp.DecodePieces(subword_pieces))  # Lossless Detokenization from subwords.
```

```
This is a test.
```

HIDE

```
print(sp.DecodeIds(subword_ids))  # Lossless Detokenization from subword ids.
```

```
This is a test.
```

HIDE

```
# By default the first 3 ids are token for unknown, bos and eos.
for i in range(3):
  print(sp.IdToPiece(i))
```

```
<unk>
<s>
</s>
```

HIDE

```
# The word "test" is not common enough in Shakespeare.
# So it is not part of our small vocabulary.
print(sp.PieceToId("test"))  # Map to integer representing unknown.
```

```
0
```

2.3.3 Subword Sampling

Remember that the unigram model for segmentation is probabilistic, which means that one sentence can be represented by more than one sequence of segments (or subwords). By default the most probable segmentation is used when we are doing encoding-decoding task with SentencePiece. But we can also generate segments using sampling based on the subword probability in our vocabulary:

HIDE

```
for _ in range(5):
  print(sp.SampleEncodeAsPieces("This is a test", nbest_size=-1, alpha=.1))
```

```
['▁Th', 'is', '▁', 'is', '▁', 'a', '▁', 't', 'e', 'st']
['▁T', 'h', 'i', 's', '▁is', '▁a', '▁', 't', 'e', 's', 't']
['▁This', '▁', 'is', '▁a', '▁', 'te', 's', 't']
['▁This', '▁', 'i', 's', '▁a', '▁', 't', 'est']
['▁This', '▁', 'is', '▁a', '▁', 't', 'e', 's', 't']
```

Or we can simply generate the top 5 most probable segmentations:

HIDE

```
for seg in sp.NBestEncodeAsPieces("This is a test", 5):
  print(seg)
```

```
['▁This', '▁is', '▁a', '▁t', 'est']
['▁This', '▁is', '▁a', '▁', 't', 'est']
['▁This', '▁is', '▁a', '▁', 'te', 'st']
['▁This', '▁is', '▁a', '▁t', 'e', 'st']
['▁This', '▁is', '▁', 'a', '▁t', 'est']
```

3 Exercise: Chinese Words Segmentation

In this section we demonstrate the segmentation on a non-whitespace-separated langauge: traditional chinese. We will use a subset of wikipedia articles (from Wikimedia Downloads (https://dumps.wikimedia.org/)). We use the WikiExtractor (https://github.com/attardi/wikiextractor) tool

to process the raw dump, and OpenCC (https://github.com/BYVoid/OpenCC) to convert the texts into traditional chinese (Taiwan standard).

HIDE

```
# Prepare zh-wiki dumps
OUTDIR=data/zhwiki_texts
if [ ! -d $OUTDIR ]
then
  # Download the data if not found locally.
  WIKIDUMP=zhwiki-latest-pages-articles1.xml-p1p162886
  if [ ! -f data/"$WIKIDUMP" ]
  then
    wget https://dumps.wikimedia.org/zhwiki/latest/$WIKIDUMP.bz2
    mv $WIKIDUMP.bz2 data
    bzip2 -d -k $WIKIDUMP.bz2
  fi
  # Extract clean text from xml dump.
  WikiExtractor.py --json --no_templates --processes 4 \
    -o $OUTDIR --min_text_length 10 data/$WIKIDUMP
fi
```

HIDE

```
import jsonlines

infile = "data/zhwiki_texts/AA/wiki_00"
def parse_wiki_sent(infile):
  # Simple sentence delimiter.
  _DELIMITER_HALF = ["!", "\?"]
  _ZH_DELIMITER_FULL = ["。", "！", "？"]
  _GENERAL_DELIMITER = ["\n"]
  _DELIMITER = _DELIMITER_HALF + _ZH_DELIMITER_FULL + _GENERAL_DELIMITER
  split_pat = r'{}'.format('|'.join(_DELIMITER))

  # Read paragraphs.
  paragraphs = []
  with jsonlines.open(infile) as jlines:
    for j in jlines:
      paragraphs.append(j["text"])

  # Break by newline.
  lines = []
  for p in paragraphs:
    lines.extend(p.lower().split("\n"))

  # Further break by sentence. We use a naive approach here just for simplicity.
  sents = []
  for l in lines:
    if len(l) >= 20:
      sents.extend(re.split(split_pat, l))

  return sents

sents = parse_wiki_sent(infile)

# Show raw cleaned texts.
for s in sents[:3]:
  print(s + "\n")
```

```
数学是利用符号语言研究數量、结构、变化以及空间等概念的一門学科，从某种角度看屬於形式科學的一種

數學透過抽象化和邏輯推理的使用，由計數、計算、量度和對物體形狀及運動的觀察而產生

數學家們拓展這些概念，為了公式化新的猜想以及從選定的公理及定義中建立起嚴謹推導出的定理
```

HIDE

```
# Write out sentence files from processed wiki dumps.
indir = "data/zhwiki_texts"
outfile = "data/zhwiki_sents.txt"

if not os.path.exists(outfile):
  with open(outfile, "w") as f:
    for root, dirs, leafs in os.walk(indir):
      for l in leafs:
        infile = os.path.join(root, l)
        if os.path.isfile(infile):
          sents = parse_wiki_sent(infile)
          for s in sents:
            if s != "":
              f.write(s + "\n")
```

HIDE

```
# Bash
# Convert to traditional chinese in Taiwan standard.
if [ ! -f "data/tw_zhwiki_sents.txt" ]
then
  opencc -i data/zhwiki_sents.txt -o data/tw_zhwiki_sents.txt -c s2twp.json
fi
```

HIDE

```
head -n5 data/tw_zhwiki_sents.txt
```

```
基督宗教歷史，指基督宗教、基督教世界及各宗派教會的歷史，從公元一世紀耶穌及其門徒的時代開始，直到現代
基督宗教於公元一世紀中葉發源於羅馬帝國統治的黎凡特地區
起先是被壓迫的宗教，但很快從耶路撒冷傳播到整個近東，包括亞蘭，亞述，美索不達米亞，腓尼基，小亞細亞，約旦和埃及等地
在301年成為亞美尼亞國教，319年成為喬治亞國教，325年成為阿克蘇姆帝國國教，380年成為羅馬帝國國教
公元431年以弗所公會議後，聶斯脫利派分離，形成東方亞述教會
```

HIDE

```
import sentencepiece as spm

# Skip training if the model file is already there, to save notebook rendering time.
if not os.path.exists("twzh.model"):
  spm_args = "--input=data/tw_zhwiki_sents.txt"
  spm_args += " --model_prefix=twzh"
  spm_args += " --vocab_size=30000"
  spm_args += " --model_type=unigram"
  spm.SentencePieceTrainer.Train(spm_args)
```

HIDE

```
twzh_model = spm.SentencePieceProcessor()
twzh_model.Load("twzh.model")
```

```
True
```

HIDE

```
twzh_model.EncodeAsPieces("光復香港，時代革命")
```

```
['▁', '光復', '香港', ',', '時代', '革命']
```

HIDE

```
twzh_model.EncodeAsPieces("我們與惡的距離")
```

```
['▁我們', '與', '惡', '的距離']
```

HIDE

```
twzh_model.EncodeAsPieces("所有的模型都是錯的，但有些是有用的")
```

```
['▁', '所有的', '模型', '都是', '錯', '的', ',', '但', '有些', '是', '有用的']
```

The result is not perfect, but considering the effort we put it is quite good already. For a bigger corpus that may be more representative to the downstream application, the unsupervised tokenization is a very convenient way to quickly dive into the modeling phase with little pre-processing effort of the traditional NLP tasks.

4 References

Kudo, Taku. 2018. “Subword Regularization: Improving Neural Network Translation Models with Multiple Subword Candidates.” *arXiv Preprint arXiv:1804.10959*.

Kudo, Taku, and John Richardson. 2018. “Sentencepiece: A Simple and Language Independent Subword Tokenizer and Detokenizer for Neural Text Processing.” *arXiv Preprint arXiv:1808.06226*.

Sennrich, Rico, Barry Haddow, and Alexandra Birch. 2015. “Neural Machine Translation of Rare Words with Subword Units.” *arXiv Preprint arXiv:1508.07909*.

---

1. We borrow the example from this stack-overflow thread (https://stackoverflow.com/questions/11808074/what-is-an-intuitive-explanation-of-the-expectation-maximization-technique/43561339#43561339), with a modification to use a general initial guess by randomizing the hidden state weights.↩