Source: KDnuggets

When performing a [natural language processing task](https://www.kdnuggets.com/2017/11/framework-approaching-textual-data-tasks.html), our text data transformation proceeds more or less in this manner:

**raw text corpus → processed text → tokenized text → corpus vocabulary → text representation**

Keep in mind that this all happens prior to the actual NLP task even beginning.

The corpus vocabulary is a holding area for processed text before it is transformed into some [representation](https://www.kdnuggets.com/2018/11/data-representation-natural-language-processing.html) for the [impending task](https://www.kdnuggets.com/2018/10/main-approaches-natural-language-processing-tasks.html), be it classification, or language modeling, or something else.

The vocabulary serves a few primary purposes:

* help in the preprocessing of the corpus text
* serve as storage location in memory for processed text corpus
* collect and store metadata about the corpus
* allow for pre-task munging, exploration, and experimentation

The vocabulary serves a few related purposes and can be thought of in a few different ways, but the main takeaway is that, once a corpus has made its way to the vocabulary, the text has been processed and any relevant metadata should be collected and stored.

This post will take a step by step look at a Python implementation of a useful vocabulary class, showing what is happening in the code, why we are doing what we are doing, and some sample usage. We will start with some code from [this PyTorch tutorial](https://pytorch.org/tutorials/beginner/chatbot_tutorial.html), and will make a few modifications as we go. Though this won't be terribly programming heavy, if you are wholly unfamiliar with Python object oriented programming, [I recommend you first look here](https://realpython.com/python3-object-oriented-programming/).

The first thing to do is to create values for our start of sentence, end of sentence, and sentence padding special tokens. When we tokenize text (split text into its atomic constituent pieces), we need special tokens to delineate both the beginning and end of a sentence, as well as to pad sentence (or some other text chunk) storage structures when sentences are shorter then the maximum allowable space. More on this later.

PAD\_token = 0 # Used for padding short sentences

SOS\_token = 1 # Start-of-sentence token

EOS\_token = 2 # End-of-sentence token

What the above states is that our stat of sentence token (literally 'SOS', below) will take index spot '1' in our token lookup table once we make it. Likewise, end of sentence ('EOS') will take index spot '2', while the sentence padding token ('PAD') will take index spot '0'.

The next thing we will do is create a constructor for our Vocabulary class:

def \_\_init\_\_(self, name):

self.name = name

self.word2index = {}

self.word2count = {}

self.index2word = {PAD\_token: "PAD", SOS\_token: "SOS", EOS\_token: "EOS"}

self.num\_words = 3

self.num\_sentences = 0

self.longest\_sentence = 0

The first line is our \_\_init\_\_() declaration, which requires 'self' as its first parameter (again, [see this link](https://realpython.com/python3-object-oriented-programming/)), and takes a Vocabulary 'name' as its second.

Line by line, here's what the object variable initializations are doing

* self.name = name → this is instantiated to the name passed to the constructor, as something by which to refer to our Vocabulary object
* self.word2index = {} → a dictionary to hold word token to corresponding word index values, eventually in the form of 'the': 7, for example
* self.word2count = {} → a dictionary to hold individual word counts (tokens, actually) in the corpus
* self.index2word = {PAD\_token: "PAD", SOS\_token: "SOS", EOS\_token: "EOS"} → a dictionary holding the reverse of word2index (word index keys to word token values); special tokens added right away
* self.num\_words = 3 → this will be a count of the number of words (tokens, actually) in the corpus
* self.num\_sentences = 0 → this will be a count of the number of sentences (text chunks of any indiscriminate length, actually) in the corpus
* self.longest\_sentence = 0 → this will be the length of the longest corpus sentence by number of tokens

From the above, you should be able to see what metadata about our corpus we are concerned with at this point. Try and think of some additional corpus-related data you might want to keep track of, which we are not.

Since we have defined that metadata which we are interested in collecting and storing, we can move on to performing the work to do so. A basic unit of work we will need to do to fill up our vocabulary is to add words to it.

def add\_word(self, word):

if word not in self.word2index:

# First entry of word into vocabulary

self.word2index[word] = self.num\_words

self.word2count[word] = 1

self.index2word[self.num\_words] = word

self.num\_words += 1

else:

# Word exists; increase word count

self.word2count[word] += 1

As you can see, there are 2 scenarios we can encounter when trying to add a word token to our vocabulary; either it does not already exists in the vocabulary (if word not in self.word2index:) or it does (else:). If the word does not exist in our vocabulary, we want to add it to our word2index dict, instantiate our count of that word to 1, add the index of the word (the next available number in the counter) to the index2word dict, and increment our overall word count by 1. On the other hand, if the word already exists in the vocabulary, simply increment the counter for that word by 1.

How are we going to add words to the vocabulary? We will do so by feeding sentences in and tokenizing them as well go, processing the resulting tokens one by one. Note, again, that these need not be sentences, and naming these 2 functions add\_token and add\_chunk may be more appropriate than add\_word and add\_sentence, respectively. We will leave the renaming for another day.

def add\_sentence(self, sentence):

sentence\_len = 0

for word in sentence.split(' '):

sentence\_len += 1

self.add\_word(word)

if sentence\_len > self.longest\_sentence:

# This is the longest sentence

self.longest\_sentence = sentence\_len

# Count the number of sentences

self.num\_sentences += 1

This function takes a chunk of text, a single string, and splits it on whitespace for tokenization purposes. This is not robust tokenization, and is not good practice, but will suffice for our purposes at the moment. We will revisit this in a follow-up post and build a better approach to tokenization into our vocabulary class. In the meantime, you can read more on text data preprocessing [here](https://www.kdnuggets.com/2017/12/general-approach-preprocessing-text-data.html) and [here](https://www.kdnuggets.com/2018/03/text-data-preprocessing-walkthrough-python.html).

After splitting our sentence on whitespace, we then increment our sentence length counter by one for each word we pass to the add\_word function for processing and addition to our vocabulary (see above). We then check to see if this sentence is longer than other sentences we have processed; if it is, we make note. We also increment our count of corpus sentences we have added to the vocabulary thus far.

We will then add a pair of helper functions to help us more easily access 2 of our most important lookup tables:

def to\_word(self, index):

return self.index2word[index]

def to\_index(self, word):

return self.word2index[word]

The first of these functions performs the index to word lookup in the appropriate dictionary for a given index; the other performs the reverse lookup for a given word. This is essential functionality, as once we get our processed text into the vocabulary object, we will want to get it back out at some point, as well as perform lookups and reference metadata. These 2 functions will be handy for much of this.

Putting this all together, we get the following.

|  |  |
| --- | --- |
|  | class Vocabulary: |
|  | PAD\_token = 0 # Used for padding short sentences |
|  | SOS\_token = 1 # Start-of-sentence token |
|  | EOS\_token = 2 # End-of-sentence token |
|  |  |
|  | def \_\_init\_\_(self, name): |
|  | self.name = name |
|  | self.word2index = {} |
|  | self.word2count = {} |
|  | self.index2word = {PAD\_token: "PAD", SOS\_token: "SOS", EOS\_token: "EOS"} |
|  | self.num\_words = 3 |
|  | self.num\_sentences = 0 |
|  | self.longest\_sentence = 0 |
|  |  |
|  | def add\_word(self, word): |
|  | if word not in self.word2index: |
|  | # First entry of word into vocabulary |
|  | self.word2index[word] = self.num\_words |
|  | self.word2count[word] = 1 |
|  | self.index2word[self.num\_words] = word |
|  | self.num\_words += 1 |
|  | else: |
|  | # Word exists; increase word count |
|  | self.word2count[word] += 1 |
|  |  |
|  | def add\_sentence(self, sentence): |
|  | sentence\_len = 0 |
|  | for word in sentence.split(' '): |
|  | sentence\_len += 1 |
|  | self.add\_word(word) |
|  | if sentence\_len > self.longest\_sentence: |
|  | # This is the longest sentence |
|  | self.longest\_sentence = sentence\_len |
|  | # Count the number of sentences |
|  | self.num\_sentences += 1 |
|  |  |
|  | def to\_word(self, index): |
|  | return self.index2word[index] |
|  |  |
|  | def to\_index(self, word): |
|  | return self.word2index[word] |

[**view raw**](https://gist.github.com/mmmayo13/4fb6adb245532622d3e9ec12a8bb1f29/raw/39470d868757edd10ae26a234a4945b9fed4afb0/vocabulary.py)[**vocabulary.py**](https://gist.github.com/mmmayo13/4fb6adb245532622d3e9ec12a8bb1f29#file-vocabulary-py) hosted with ❤ by [**GitHub**](https://github.com/)

Let's see how this works. First, let's create an empty vocabulary object:

voc = Vocabulary('test')

print(voc)

<\_\_main\_\_.Vocabulary object at 0x7f80a071c470>

Then we create a simple corpus:

corpus = ['This is the first sentence.',

'This is the second.',

'There is no sentence in this corpus longer than this one.',

'My dog is named Patrick.']

print(corpus)

['This is the first sentence.',

'This is the second.',

'There is no sentence in this corpus longer than this one.',

'My dog is named Patrick.']

Let's loop through the sentences in our corpus and add the words in each to our vocabulary. Remember that add\_sentence makes calls to add\_word:

for sent in corpus:

voc.add\_sentence(sent)

Now let's test what we've done:

print('Token 4 corresponds to token:', voc.to\_word(4))

print('Token "this" corresponds to index:', voc.to\_index('this'))

This is the output, which seems to work well.

Token 4 corresponds to token: is

Token "this" corresponds to index: 13

Since our corpus is so small, let's print out the entire vocabulary of tokens. Note that since we have not yet implemented any sort of useful tokenization beyond splitting on white space, we have some tokens with capitlized first letters, and others with trailing punctuation. Again, we will deal with this more appropriately in a follow-up.

for word in range(voc.num\_words):

print(voc.to\_word(word))

PAD

SOS

EOS

This

is

the

first

sentence.

second.

There

no

sentence

in

this

corpus

longer

than

one.

My

dog

named

Patrick.

Let's create and print out lists of corresponding tokens and indexes of a particular sentence. Note this time that we have not yet trimmed the vocabulary, nor have we added padding or used the SOS or EOS tokens. We add this to the list of items to take care of next time.

sent\_tkns = []

sent\_idxs = []

for word in corpus[3].split(' '):

sent\_tkns.append(word)

sent\_idxs.append(voc.to\_index(word))

print(sent\_tkns)

print(sent\_idxs)

['My', 'dog', 'is', 'named', 'Patrick.']

[18, 19, 4, 20, 21]

And there you go. It seems that, even with our numerous noted shortcomings, we have a vocabulary that might end up being useful, given that it exhibits much of the core necessary functionality which would make it eventually useful.

A review of the items we must take care of next time include:

* perform normalization of our text data (force all to lowercase, deal with punctuation, etc.)
* properly tokenize chunks of text
* make use of SOS, EOS, and PAD tokens
* trim our vocabulary (minimum number of token occurrences before stored permanently in our vocabulary)

Next time we will implement this functionality, and test our Python vocabulary implementation on a more robust corpus. We will then move data from our vocabulary object into a useful data representation for NLP tasks. Finally, we will get to performing an NLP task on the data we have gone to the trouble of so aptly preparing.