Programming for Computational Linguistics 2018/2019

Final Project Checkpoint 3 — `train`

These checkpoint documents provide a detailed recommendation for how to proceed with the project. They will break the requirements into small, well-defined steps. It is not necessary to follow this plan — you can choose to implement the requirements however makes the most sense to you. However, if you are unsure of how to proceed, this should provide you a good starting point.

Now we can convert text into a sequence of tokens (from `tokenize`), and we can convert a sequence of tokens into a sequence of n-grams (from `get_ngrams`

Now, let's get to implementing our `train(sequences)` method. This method should take a list of token sequences as input, and it should use those to "train" our language model. In this case, training means remembering some statistics about the n-grams in the training data `sequences`. We will also want to remember the set of all words seen, as this will be useful later.

There are many, very different ways this function might be implemented. Your choice of how to remember statistics from the training texts will affect how you implement future methods like `p_next` and `generate`. We will explore one possible approach in these checkpoint documents, but if you have a different idea, feel free to do this however you'd like.

What we want to remember is this: For every $(n-1)$-gram that we see, we want to remember which words follow it, and how often. That way, when we are using our language model for prediction, we can look up the last $n-1$ words of the input, and pick a likely candidate for the n^{th} word in the n-gram.

More specifically, here's what we need to do to get our `train` function to work:

- We need two new instance variables: `counts`, a dict, and `vocabulary`, a set. We haven't learned about sets yet, so look those up in the python documentation to learn how they work. Both of these instance variables should be initialized to be empty in the constructor.
- After `train(sequences)` is called, `vocabulary` should contain all words which occur in `sequences`. Start writing `train(sequences)` such that it does this.
- After `train(sequences)` is called, `counts` should be a dict of dicts. The keys of the (outer) dict should be the $(n-1)$-grams which occur in

the texts, and the values should be dicts. The keys of these (inner) dicts are all the words which follow their respective $(n-1)$-grams in the texts. The values of these inner dicts are the number of times each word follows its respective $(n-1)$-gram. Add code to `train(sequences)` such that this holds. Remember you can use your function `get_ngrams` to help you.

For a concrete example, after you have implemented this step, you should see something like this:

```
>>> lm = LanguageModel(3)
>>> lm.train([['the', 'cat', 'runs'],['the', 'dog', 'runs
    ']])
>>> lm.vocabulary
{None, 'the', 'cat', 'runs', 'dog'}
>>> lm.counts
{(None, None): {'the': 2}, (None, 'the'): {'cat': 1, 'dog
    ': 1}, ('the', 'cat'): {'runs': 1}, ('cat', 'runs'): {
   None: 1}, ('runs', None): {None: 2}, ('the', 'dog'): {
    'runs': 1}, ('dog', 'runs'): {None: 1}}
```

Make sure that your output matches this (keeping in mind that the order of dicts and sets is unimportant). Note that `lm.counts[('the', 'cat')]['runs']` is equal to the number of times the word *runs* follows the bigram prefix $\langle \textit{the}, \textit{cat} \rangle$ in the text.