Markov Text Generator Implementation

I built a bare-bones Markov text generator that takes 4 arguments. The first argument is a partially completed sentence (each word as a string in a list), the second argument is the number of n-grams to use, the corpus to use, and then an optional argument of whether or not to use deterministic mode (default is false). Below are several examples of the application of this function, using Jane Austen's *Sense and Sensibility* as the corpus, with different n-gram values in both deterministic and non-deterministic mode.

```
sentence = ['she', 'thought', 'that']
output1 = finish_sentence(sentence, 3, corpus, deterministic=False)
output2 = finish_sentence(sentence, 3, corpus, deterministic=False)
print(' '.join(output1))
she thought that i felt no desire for the stocks were at the time ;
print(' '.join(output2))
she thought that i shall call hills steep , the consolation , beyond the
most
```

Above is the output for using the given input sentence with a trigram model. As shown, I ran it in non-deterministic mode, and ran it twice to show that it is in fact using probability. As you can see, each time it is a different sentence. Neither sentence is entirely a proper English sentence, but it does more or less follow a general structure. Below I demonstrate how as the n-gram value goes down, the "sentence" resembles a proper English sentence less and less.

```
sentence = ['he', 'made']
output3 = finish_sentence(sentence, 2, corpus, deterministic=True)
output4 = finish_sentence(sentence, 1, corpus, deterministic=True)
print(' '.join(output3))
he made her , and the same time , and the same time , and
print(' '.join(output4))
he made , , , , , , , , , , , , ,
```

This case is interesting, especially $output4$, which shows that a unigram model set to deterministic will always return a comma. Even in the case of a bigram, we can see that common words, such as "and" and "the" occur quite frequently, and the code appears to become stuck in a loop. The final example included here is the function in non-deterministic mode with a relatively high n-gram.

```
sentence = ['i', 'think', 'that', 'she', 'is']
output3 = finish_sentence(sentence, 5, corpus, deterministic=True)
output4 = finish_sentence(sentence, 1, corpus, deterministic=True)
print(' '.join(output3))
i think that she is not well , she has had a nervous complaint on
```

In this case, we can see that the code begins to replicate actual strings of text from *Sense and Sensibility*. From all of these cases, we can see the inherent trade-off, both between a high and low n-gram, and between deterministic and non-deterministic.