In-class exercise: N-gram Language Models

1 Reading Data with SpaCy

This week, we'll get to use the spaCy python library for the first time. SpaCy is designed to make it easy to use pre-trained models to analyze very large sets of data. At the beginning of the semester, we'll be using spaCy as a skeleton for building our own NLP algorithms.

Install SpaCy and import it to verify that it is working.

We will use SpaCy to tokenize our text. Pause now and read about space language processing pipelines.

For this part, we only want SpaCy to tokenize our text, so we will set the **pipeline** to **[]**. Since some of our documents will be long, but since we're not doing any memory-intensive processing, we will tell SpaCy that it's okay to load large documents all at once instead of a little bit at a time. To specify these instructions, add these lines to the top of your **zipf.py** file:

```
from spaCy.lang.en import English

nlp = English(pipeline=[], max_length=5000000)
```

2 Bigram analysis

Write a function called **get_bigrams** that takes as input a spaCy Document, and returns a list of all of the bigrams in the document.

Write a function called **get_trigrams** that takes as input a spaCy Document, and returns a list of all of the trigrams in the document.

3 Language Modeling with Trigrams

We will train our model on two novels: *Emma* and *Persuasion*. We will need to a SpaCy doc for each text and merge them using the **Doc.from_docs()** function:

```
def make_doc(files):
    docs = []
    for f in files:
        with open(f,'r', encoding='latin1') as fn:
            docs.append(nlp(fn.read()))
    return Doc.from_docs(docs)
```

3.1 N-gram counts

Now we will need to store counts for each of the bigrams and trigrams in our text. Write a function called **get_ngram_counts(docs)** that takes your training doc and uses your **get_unigram**, **get_bigram**, and **get_trigram** functions to calculate the n-gram counts within the training text.

3.2 Next word probability

We now have all the pieces to calculate the probability of a word given the two words that come before it. Write a function called **calc_ngram_prob**. It should take a trigram, a bigram Collection, and a trigram Collection, and it should return the probability of the trigram according to the MLE n-gram formula:

$p(w_n|w_{n-2}w_{n-1}) = \frac{C(w_{n-2}w_{n-1}w_n)}{C(w_{n-2}w_{n-1})}$

For numerical stability, we will work with log probabilities. Wrap the division term in a call to **math.log** to convert it from a probability to a log probability.

You will also need to handle cases where the bigram or trigram is not in the given collection. Return negative infinity for out of vocabulary n-grams.

3.3 Possible next words

Write a function that takes a sequence of text and returns a list of possible next words along with their probabilities. Call it **get_possible_next_words**. It should take as arguments the sequence of text as a string, a Collection of bigrams, and a Collection of trigrams. It should use **calc_ngram_prob** to calculate the probabilities of each of the candidates and return a list.

3.4 Most likely next word

Now write a function wrapper function that uses **get_possible_next_words** to find the most likely next word for a sequence of text. Call this **predict_next_word**. It should take as arguments the sequence of text as a string, a Collection of bigrams, and a Collection of trigrams. It should return the most likely next word.

Check in

The most likely next word following "an agreeable" should be "manner", with log probability -1.5 (22%).

3.5 Generating text

We can now generate text! Write a function called **generate_text**. It should take as arguments a string representing the text to complete; n, a number of words to generate; a Collection of bigrams; and a Collection of trigrams.

3.6 Generating more text

It's boring to always generate the most likely completion. Write a function called **sample_next_word**. It should be like **predict_next_word**, except that it returns a word from the set of candidates produces by **get_possible_next_words** sampled according to its probability. You can use the **random.choices** function to help you sample.

Note: random.choices expects non-negative weights. Before passing in the probabilities as weights, you will need to them back to normal probabilities using the **math.exp function**.

Next, augment your **generate_text** with an optional mode parameter. We will use **mode** to signal whether we want the most likely next word (mode="top") or a word sampled according to its probability (mode="random").