Ngram Narrative

N-grams are a sliding window of size n over text. For example, consider the sample text "The driver is busy". The unigrams, which are n-grams of size n=1, would be [("The"), ("driver"), ("is"), ("busy")] and the bigrams, which are n-grams of size n=2, would be [("The", "driver"), ("driver", "is"), ("is", "busy")]. This simple tool can be used to build a probabilistic model for language. Using the sizes and counts of each unigram, bigram,..., up to n-gram, we can calculate the probability that one word would follow another word, at least relative to the corpus we trained on.

N-grams, along with their associated counts and probabilities, have various applications. They can allow us to both classify phrases into one corpus or another and it can allow us to generate text for a corpus. For instance, considering classifications, in the N-grams homework, we have three corpuses, one in English, one in French, and one in Italian. Using various probabilistic and statistical methods, we can find the probability of a certain sentence appearing in the English, French, or Italian corpus. We can use that as an indicator for whether or not the certain sentence is English, French, or Italian. The larger and more varied each corpus is, the more accurate the guess. On the other hand, this will also take longer and require more computing. Furthermore, spelling correction, machine translation, speech recognition, and auto suggestion typing/messaging and searching are all applications where N-grams can be and are used.

Unigrams and bigrams have proven so useful due to the probabilities that we are able to calculate using them. But how do we actually calculate these probabilities? Let us consider unigrams for now, as the calculations are nearly identical for bigrams and n-grams even higher. Let u be an example unigram list. u is a list of single tokens which may or may not have repeats. len(u) tells us how many unigrams are in the corpus. set(u) gives us all the unique unigrams in the corpus. With these tools, we can find the probabilities for each unigram.
uProb = {t: u.count(t)/len(u) for t in set(u)} is a dictionary which gives us the probability for each unigram. The process is the same for bigrams, except instead of dividing by len(bigrams), we divide by u.count(b[0])

The source text, or corpus, is very important for building a language model. Depending on the focus, tone, and subject of the source text, certain n-grams will be more common or less common. For instance, a newspaper from November or December of 2016 will likely have ("President", "Trump") as a common bigram but a newspaper from before that date will probably have that as a very uncommon bigram or may not even have it at all. This will lead to quite different outcomes when we seek to generate text from probabilities trained on different corpuses

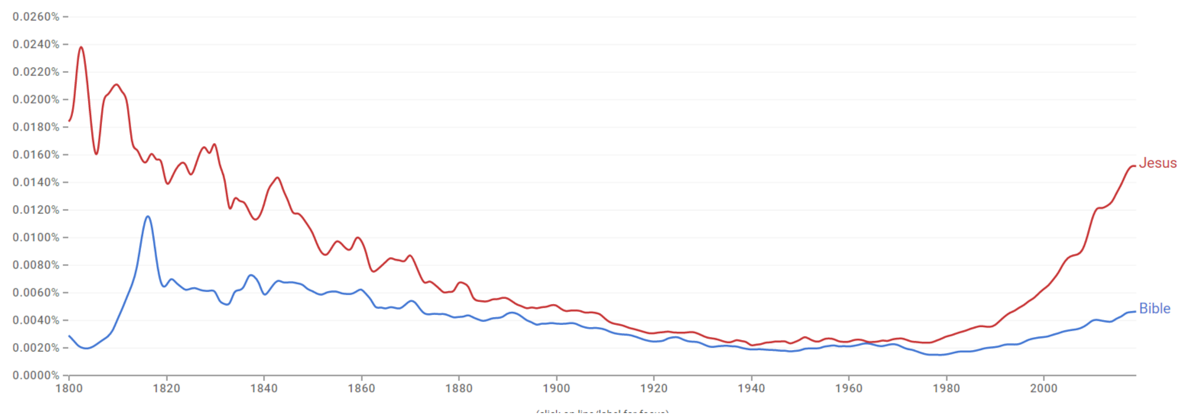
Smoothing is very important for calculating probabilities. One of the dangers of finding probabilities is that division is involved. And in any branch of math, whether it be probability or not, if a division occurs, we should immediately ask ourselves: "Is there a possibility of division by zero occurring? And if so, how do we prevent that from happening?" In our case, if an n-gram isn't in one of our dicts, there is the danger that we would divide by zero. To prevent this, we invoke smoothing. Smoothing replaces zero values with non-zero values instead to avoid the

danger of division by zero. One simple approach to smoothing is known as Laplace smoothing. Instead of calculating (count of a specific n-gram)/(total number of n-grams), we will use the probability:
(1 + count of a specific n-gram)/([total number of n-grams] + [total number of unique n-grams])

One application of n-grams is the ability to generate text for a corpus. We will now go over a naive approach to using n-grams to generate language. We will be using unigrams and bigrams. From each unigram and bigram, create a dictionary for each whose key is one of the unigrams/bigrams and whose value is the probability that unigram/bigram appears in the corpus. Once we have the probability dictionaries, we just need a start word. From that start word, we look at all the bigrams which have that start word and we choose the bigram with the highest probability. We then shift the start word and repeat the process until we hit a terminating character, usually a period. Trigrams will work better than bigrams and 4-grams will do better still. In general, the higher the n-gram we use, the better the result

We can evaluate language models using either extrinsic or intrinsic evaluations. Extrinsic evaluations use humans annotating in accordance with a predefined metric. As it involves humans, these evaluations are time-consuming and somewhat expensive so they are usually not the first option sought after. The other type of evaluation is known as intrinsic. With intrinsic evaluations, we use a metric known as perplexity. Perplexity is the measure of how well the language model predicts the text in the test data. The lower perplexity, the better

Using the Google Books Ngram Viewer, we can see how common or uncommon various n-grams are in published books. The viewer displays a graph showing how those phrases have occurred in a corpus of books over the years. Below is an example:


Jesus
Bible