At its core, an n-gram model is a sliding window over text, where "n" represents the number of words in that window. For example, a sliding window with one word is considered a unigram, two words is a bigram, and so on, generalizing to n-gram. Though relatively simple in context, they can be used to create complex models of language. These models are probabilistic, with a probability assigned to each n-gram based on its frequency in a large corpus.

As explained, the results of a probabilistic language model heavily depend on the corpus selected. For instance, if we build a model that identifies the probability of something being English, but our corpus only consists of Shakespearean text, it would not produce good results for identifying the probability of a 21st-century text being "English."

N-gram models have many uses in the world of natural language processing. They can be used to generate language models for speech recognition, machine translation, and language generation. Additionally, they can aid in text prediction, auto-completion of phrases, and spell-checking. An interesting use case of n-grams is sentiment analysis, where they can identify the most common n-grams associated with positive or negative sentiment in a piece of text.

Now that we understand the basics of N-gram models, let us further look into the mathematics of it. As mentioned, N grams are based upon probabilities of words appearing in a corpus. The most simple probability is to calculate the probability of one word, which logically follows as: $P(a\ single\ word) = \frac{number\ occurrences\ of\ the\ word}{Total\ number\ of\ words}$. However, calculating these probabilities continually gets more complicated as you increase the number of words you want to calculate the probability of. Thus, we use Markov assumptions to simplify this calculation and can generalize our calculating of probabilities to this equation $\Pi\ (k = 1 \dots n) = P(w_k | w_{k-1})$, where w_k is the current word and w_{k-1} is the previous word. We can see that for the probability of words in a sequence, we only really need to look at the probability of the previous word, not all of the previous words.

Upon looking at the calculation of these probabilities, we can see a glaring issue that could present itself in our calculations. What if the word we want to calculate the probability of does not present itself in the text, meaning its $\frac{0}{Total\ number\ of\ words}$, thus will result in an overall probability of 0 for the entire phrase. Of course, it's unreasonable to expect that every word ever will appear in its corpus, yet there is an easy solution to this issue: smoothing.

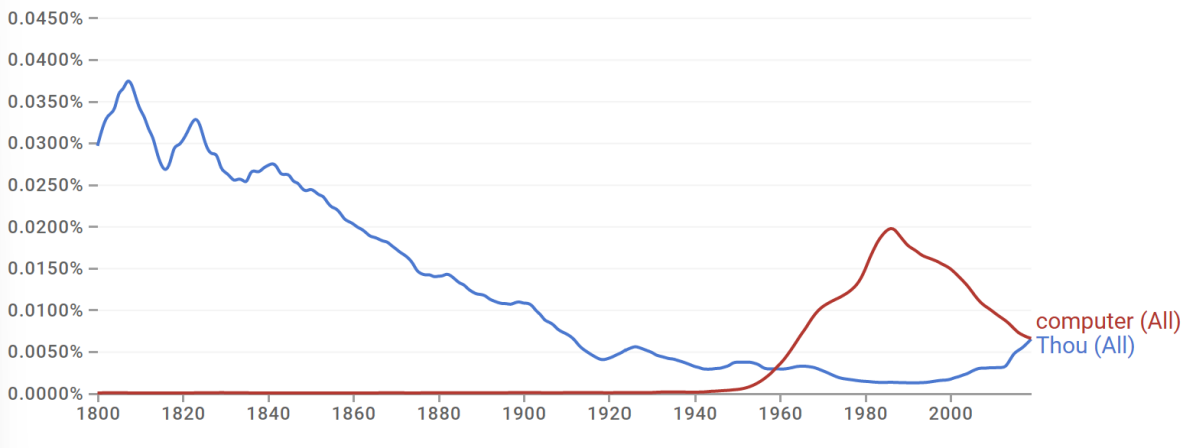
There are many different ways to apply smoothing to a probabilistic distribution, but a relatively easy one is called LaPlace, where the new probability formula is equal to $\frac{number\ occurrences\ of\ the\ word + 1}{Total\ number\ of\ unique\ unigrams + vocabulary\ count}$. This is a simple change that guarantees we will never encounter the "0" issue that we examined earlier. However, you may notice that we are changing our original formula quite a bit, resulting in a big difference in our probabilities, thus an even simpler approach was developed: the Good-Turing technique where $P = \frac{the\ probability\ of\ a\ word\ that\ occurs\ once}{the\ total\ word\ count}$.

Now that we've established an understanding behind the basic math of N grams, we can easily explain the logic behind text generation using n grams. Essentially, one has to create probability dictionaries for different n grams. The simplest would contain the probabilities of unigrams and bigrams. The higher value of N is used and the larger a corpus is , the better your text generation will be. The tradeoff is that it becomes more time consuming, as N increases. However let us continue with our example of using bigrams. The most logical (and naive approach) is an approach that only needs the first word to be provided. Given the start word, you

look at the probabilities of the bigrams that start with this word and concatenate the one with the highest probability to our final phrase.

How are we to know if our language model is considered to be a “good” model? One possible way is to have human annators evaluate the result, aka the intrinsic approach. This can be very time consuming and expensive, so it is best to use alternative approaches in the early stages of model development. It is better to begin with intrinsic methods, which use a metric to compare models. One such metric could be perplexity (aka branching factor), which measures how well the language model will predict the some text in the test data, with a lower perplexity indicating a stronger model. Intuitively a lower branching factor, means less choices for the next “node/word” thus, having low perplexity is better.

One interesting application of n-gram models is Google’s n-gram viewer. This application allows for a visualization of the appearance of inputted n-grams in corpus throughout a range of time. For example, this is the result of using it on 2 ngrams: Thou, and Computer. I intentionally chose an “older” word and a more modern word, so we could see large gaps in the graph that logically would make sense.



We can see that “thou” peaked in the 1800’s and has been on a steady decline, which makes sense, as it is an old fashioned word. In contrast, the use of the word “computer”did not make an appearance until its start in the digital revolution occurring in the latter half of the 20th century.