Messing with N-Grams

What are N-Grams

Ngrams are an algorithmic way of replicating the way humans think about patterns in speech. The order in which words are placed can help us predict the next word. If we were to take a sample from a text of length n, and slide that sample 1 word at a time, we would create an ngram. The resulting list would contain a list of elements, each of length n, in which we would have every possible list of n words from the text.

We can then count each time each unique ngram appears in the text. This dictionary of unique ngrams and their count in the text could be used as the probability of each sequence appearing in a corpus. This list of probabilities be used to build a language model capable of predicting how likely a given of sequence of words is to appear in the corpus. Or, even better, figure out how likely the next word in a sequence is given the last n words.

Applications of N-Grams

Given enough scale, this simple pattern recognition tool can preform various tasks at high efficiency:

- Broadly, they can be used to verify how likely a sequence of words may fit in a language. I can see this being used for checking the success of a translation without directly analyzing the common phrases.
- You can use ngrams to generate text (naively) by selecting text that fits the previous text with the highest probability as calculated by the ngram model. This could be used to generate large texts, or just simple auto suggestions on a keyboard.
- You may use an ngram model to identify common phrases for the purposes of assigning an analyzing sentiment. This could be better than analyzing the sentiment of just individual words in a text.
- Ngrams could be used to find what genre a text is in. Maybe a rom-com has a lot of phrases about love and having "meet cutes" and such.

Probabilities

When calculating if a given ngram, we focus on the unigram (n = 1) and bigram (n = 2) cases. Note, for the equations below, $C(*)$ denotes counting the occurrence of a given word w_n

- The probability of a unigram appearing in a given text is simply it's the number of that unigram in a text divided by the total unigrams in the text.

$$P(w_1) = \frac{C(w_1)}{\Sigma_n C(w_n))}$$

- The probability of a bigram is the probability of the first word appearing multiplied by the probability of the second word following the first:

$$P(w_1, w_2) = P(w_1)P(w_2|w_1) = \frac{C(w_1)}{\Sigma_n C(w_n))} \frac{C(w_1, w_2)}{C(w_1)}$$

The Source Text

We would usually like our language models to have a perfect understanding of the language they are representing. Yet, our sources or corpora, we use for training only represent a fraction of the actual language. This means our model will always be bottle necked by the expansiveness of it's training data. If a source text speaks a certain way, doesn't use a particular word, or has even certain beliefs, the output ngram model will still only embed the patterns of that given text. Thus reproducing the unique usage of words and phrases when applied in actual use cases.

Smoothing

One of the problems described above, is that words might be entirely lacking from the example text that are needed in application. Mathematically, the probability for a given word in that data set would be 0, even though that word may very well exist in the superset language of the text. Even worse, when multiplying probabilities the 0 will reduce the chances of a given sequence to 0, even if it should be just quite low.

A simple approach to stopping this issue, and getting rid of 0's, is to simply add 1 to every unique word count in the dictionary. This requires you add the number of unique words in the denominator of the previous function:

$$P(w_1) = \frac{C(w_1) + 1}{\Sigma_n C(w_n)) + \Sigma_n C(\text{set}(w_n))}$$

To simplify with N being the total number of unigrams in the source text and V being the total unique unigrams in the source text:

$$P(w_1) = \frac{C(w_1) + 1}{N + V}$$

Text Generation

By creating a dictionary of what uniqrams or bigrams are most probable in a given dictionary, an algorithm could look at a given word, evaluate what is most likely to come next, and generate that text. Allow this to run for some time and you'll have probabilistic word generation. The issue is, given a size n ngram, a model can only see a limited context at a given time. Even given a large context, a language model will not actually understand the meaning of the text. It only analyzes the frequency of n-grams and can generate text that is perhaps grammatically correct but still meaningless.

Evaluation of Language Models

Instead of getting humans to rate models, we can use a measure called perplexity, denoted by PP. Perplexity is the:

> Inverse probability of seeing the words we observe, normalized by the number of words. This is an exponentiation of the entropy, the average number of bits needed to encode the information in a random variable.

This confused me to all heck, but to put it simply, a perplexity score of 1 means a model can predict the next word correctly every time, while a low score means it is often wrong.

Googles N-gram Viewer

Googles N-gram viewer allows users to search for some words or phrases, and it searches Google Books to find how frequent those words are used. It creates graphs that show how frequently each word or phrase appears in books published throughout the time it has available. I find it typically has a hard time distinguishing languages in old books. And also, can't really keep up with language that isn't regularly put in books. Here is an example: