Reg Gonzalez
CS 4395.001

N-grams Narrative

(a.) What are N-grams and how can they be used to build a language model?

An n-gram is a contiguous piece of n words over some text (sometimes referred to as a "sliding window" of text that's n words long). For example, a bigram consists of two words while a unigram consists of one word. N-gram models are important in natural language processing, particularly in regards to creating probabilistic models for languages.

N-grams are helpful in creating probabilistic models in several ways. One way in which n-grams are useful is that they can help with word prediction. By having many n-gram models, you may be able to predict the next word in a sentence or query by seeing how frequent your n-grams appear in a corpus (a body of text). For example, let's say you had the sentence "Go to the ticket booth and buy the ____". Clearly, the word in the blank should be "tickets". If you had a model that had "buy the tickets" as a trigram that commonly occurred in the corpus, you might be able to predict that word. N-grams don't just help with predicting words, but they can also aid in spelling errors and finding out which words should be clumped together to form single entities (i.e., instead of "high" and "school" being separate, they can form together to be the bigram "high school" if they occur frequently enough).

N-grams are used to build language models by being able to take predict the next word in a corpus given the previous words. For this assignment, we used n-grams in order to predict whether a line from a test file was either English, French, or Italian. By using unigrams and bigrams, we were able to calculate the probability of a line of text's language. The unigrams and bigrams helped us by givine us a knowledge base of words and phrases that are common to each language. We were able to calculate the probability of what the next word in the text would be given the unigrams and bigrams that we extracted from an English corpus, a French corpus, and an Italian corpus. In the end, I got an accuracy score of 96.33%, which is pretty high.

(b.) Applications where N-grams could be used

- Word prediction
- Spelling error detection
- Identifying/classifying languages
- Autocomplete

Reg Gonzalez
CS 4395.001

N-grams Narrative

(c.) Description of how probabilities of unigrams & bigrams are calculated

Unigram probabiltiies are pretty straightforward to calculate. Because a unigram is just a single word, to calculate the probability of a unigram, you would just divide the number of occurrences of that single unigram over the toal number of words in the corpus.

The probability of bigrams are calculated using this equation:

$$P(w_i w_{i+1}) = \frac{C(w_i w_{i+1})}{C(w_i)}$$

That is, the count of the bigram in the corpus divided by the first word in the bigram.

When calculating probabilities though, it's also important to remember smoothing. Smoothing refers to the idea of filling in zero probability values with very small values in order to make the overall probability non-zero. For example, let's say that we had a corpus and we wanted to calculate the probability of the trigram "he drank coffee." If the word "coffee" occurs in the text 0 times, then the entire probability would be 0. We would avoid this by "smoothing," or filling in that 0 value with a very small number. One popular technique for smoothing is laplace smoothing, which adds 1 to the 0 count.

(d.) Importance of source text when building a language model

Source text is extremely important when building a language model. The corpus you decide to build your model off of will affect the accuracy of your model and how it interacts with your input data. For example, if you are building a model off of random tweets and trying to use it on text from a computer science textbook, your model really isn't going to work well with that. You want to make sure that the data you're sending in as input is similar enough to the model you've built in order to get more accurate and better results.

Reg Gonzalez
CS 4395.001

N-grams Narrative

(e.) Importance of smoothing

When calculating probabilities, smoothing is a very important part of that process. As explained earlier, smoothing means replacing zero values with small non-zero values. We do this in order to get non-zero probabilities, since it only takes one zero probability in order to mess up your overall probability.

One simple approach to smoothing, as described earlier, is laplace smoothing. Here, we simply add 1 to the 0 count. Because we're calculating probabilities, we don't know which ones will have a count of 0 until we actually calculate them. To compensate for this, we add 1 to all the counts. To balance this out even more, in the denominator we add the vocabulary count (i.e., the number of unique words in the corpus).

(f.) Description of how language models can be used for text generation & its limitations

There is a naive approach to how language models can generate language. You can build a function, give it a starting word, and have the function look through all the bigrams you've used to built the language model using the starting word as the first part of the bigram. You can then compare all the bigrams with the starting word in the first position can pick the one with the highest probability. Once the function has picked that bigram, it can simply just print out that phrase or concatenate it to the sentence. This would theoretically go on until the last token was some sort of ending punctuation (e.g., a period, question mark, exclamation point, etc.).

A limitation of this approach is that it's a very simple way of generating language. It's also limited by a corpus that would be relatively small. The approach I described also uses bigrams, and generally speaking, you're not going to get great language generation with just bigrams. N-grams with a higher 'n' value would work better.

There are other ways of generating language though. You can use NLTK's .generate() function, which does get better results, especially for larger corpora. You can create dictionaries, but that is fairly time-consuming. If you're going to create dictionaries, pickling/unpickling them would save you time.

Reg Gonzalez
CS 4395.001

N-grams Narrative

(g.) Description of how language models can be evaluated

There are two ways of evaluating a language model: an extrinsic evaluation and an intrinsic evaluation. The extrinsic evaluation consists of having *human* annotators examine the answers of a language model and compare it to some sort of predefined metric. The problem with this method is that it can be very expensive and time-consuming. The intrinsic evaluation uses an internal metric and compares language models to one another. One metric that can be used for intrinsic evaluations is perplexity. Perplexity is calculated with the following equation:

$$PP(W) = P(w_1 w_2 \dots w_N)^{-\frac{1}{N}}$$

In all, we *do not* want a high perplexity score. This is because perplexity is related to entropy (the amount of "chaos" in the data). We want low entropy, meaning that we'd also want low perplexity. Perplexity can also be thought of as the branching factor for a language model. This means the number of choices we can make for a word given the previous word. Corpora that are more specialized in one domain may have a lower perplexity score than other corpora that come from a wide variety of sources.

(h.) Google's N-gram Viewer

Google's N-gram Viewer is a search engine tool that charts the amount of times a search string (of n-grams) has been searched by year, from 1500 to 2019. This tool can be used in conjunction with other NLP research or just to see how language/search terms have evolved over the years.

Here is an use case example of Google's N-gram Viewer. I compared the search terms "The Cabinet of Dr. Caligari," "The Wolf Man," and "The Evil Dead."