Parallelized N-Gram Language Modeling with Stupid Backoff for Text Generation

Project Proposal - COMS 4995 Parallel Functional Programming

Dave Epstein (UNI: ne2260)

November 17, 2019

A classical problem in natural language processing is language modeling – assigning a probability to a word based on its context (e.g. previous words in a sentence). For example, given the text "Please search Piazza before posting your. . . ", one might reasonably predict "question" as a next word, but not so much "Paul Blaer". Modern approaches train deep neural nets parametrized by hundreds of millions of weights on billion-word datasets to learn nuances of language. However, statistical methods to model language remain relatively strong, especially in industrial applications which require more interpretability and reliability than these neural networks tend to provide. I propose **parallelizing the generation of an n-gram language model on large text corpuses (10M-1B words)**. Once we obtain a language model, a practical task to probe it, which often yields entertaining results, is sampling it to generate sentences. For example, when prompted with "Haskell is a great programming language for. . . ", the state-of-the-art OpenAI GPT-2 model gives the following text: ". . . for building programs and creating websites."[1] So close!

An n-gram is nothing but a sequence of n words (so bigrams are two words, trigrams are three words, etc.). An n-gram model estimates the probability of an n^{th} word w_n given the $n-1$ previous words $w_1, \ldots, w_{n-1}$. We can calculate this as follows:

$$P(w_n | w_1, \ldots, w_{n-1}) = \frac{C(w_1, \ldots, w_n)}{C(w_1, \ldots, w_{n-1})} \tag{1}$$

That is, out of all the times that the first $n-1$ words appear, how often does the n^{th} word follow? Traditionally, these corpora are calculated by adding special tokens `<s>` and `</s>` to mark beginning and end of sentence. In large corpora containing many documents, it is sensible to compute sequence counts on each document separately, and therefore this task is highly parallelizable. Once sequence counts have been computed across the whole corpus, results can be combined and processed to yield the conditional probability model.

In practice, there are some interesting challenges in implementing such a language model. What data structures do we use to store the counts, so that the probabilities are both fast to compute and easy to decode with (good for both reading and writing)? How do we keep models processed on huge datasets in memory, considering the long-tail distribution of the English language, especially as we increase the n in n-gram? What do we do when the probability the model assigns to some sequence is zero, but it shouldn't be (just because it hasn't seen "Haskell is super awesome" in its dataset, it shouldn't say that isn't sensible English)? What is the best way to go from one of these fancy language models to actually generating cool sentences?

While the algorithm for building a simple n-gram language model is not too tricky, the first two questions provide nontrivial Haskell challenges, while the second two provide nontrivial NLP challenges. You guys probably care more about the first two questions, and others, like, how well does this parallelize (I think very well)? But I don't know the answers to them yet. I am just pretty sure that good answers exist. I will go into a bit more detail on the model I propose to implement now, instead.

On very large datasets, to deal with the erroneous assignment of zero probability to reasonable sequences, one can define "stupid lookback" when $P(w_i | \ldots) = 0$, which essentially shrinks the previous word context

[1] More output, courtesy of https://talktotransformer.com/: . . . There is a lot to love about Haskell, including: You can write your code in a single, easy-to-understand language. If you're a developer and you're looking to move your career forward, you have to understand and love Haskell. I learned Haskell in 2010 from a friend of mine who is a software developer, but that was a long time ago.

as much as necessary until it has appeared before, down-weighing the resultant probability by a heuristic hyperparameter λ at each shrink:

$$P(w_i|w_{i-k+1},\ldots,w_{i-1}) = \lambda P(w_i|w_i,\ldots,w_{i-k+2}) \text{ when } C(w_{i-k+1},\ldots,w_{i-1}) = 0 \tag{2}$$

This is a recursive definition that goes from n-gram to $n-1$-gram all the way to unigram modeling (naive word counts) and terminates when a context is found. This type of formulation is natural to implement in Haskell.

To generate output using our model, we just need to randomly sample some starting tokens (we can use the learned probability distribution for this, if we want, or input our own prompt) and calculate likely next words. One way to generate output given some context using the n-gram model is to just greedily pick the word with highest assigned probability (or sample according to the distribution) and repeat the process. The problem with this is that the mode English sentence is quite boring (try spamming the middle option in your keyboard's autocomplete). We can use **beam search** to mitigate this problem, where we again sample our model repeatedly, but maintain the β most likely "beams" of text[2] - so $\beta = 1$ degenerates to greedy sampling. A relatively recent paper[3] proposes a simple method to even further improve beam search, by introducing a diversity penalty to samples, to encourage higher variation in sampled sequences. I will implement this variant so that the learned language models can give funnier sentences.

The main speedup I think can be attained by parallelizing will be in processing massive text corpora to yield n-gram models, as described above. However, it is possible that the beam search decoding process can also be parallelized well for large values of β. In summary, I propose generating a relatively straightforward statistical language model on huge amounts of text entirely from scratch using Haskell. I believe that the problem can be elegantly solved in Haskell, and hope to get some neat demos of funny sentences for when I turn the final project in.

References

1. https://web.stanford.edu/ jurafsky/slp3/3.pdf

[2]See this short video for more detail: https://www.youtube.com/watch?v=UXW6Cs82UKo

[3]Diverse Beam Search: Decoding Diverse Solutions from Neural Sequence Models. Ashwin K Vijayakumar, Michael Cogswell, Ramprasath R. Selvaraju, Qing Sun, Stefan Lee, David Crandall, Dhruv Batra.