Predictive Text: Parallelized Sentence Auto-Completion Using N-Gram Models

Parallel Functional Programming Fall 2022 Project Proposal

Elisa Luo (eyl2130)

Introduction & Background

In natural language processing, a n-gram is a sequence of n-consecutive words or characters from a given sample of text. For sentences, we can use the unit of 1 word. For example, ("orangutan") is a 1-gram, ("thank", "you") is a 2-gram, and ("please", "go", "away") is a 3-gram.

The n-gram model is a probabilistic one. To perform text generation, our main task is to compute the $P(w|c)$, the probability of a word w given some context c. In a n-gram model, this context c is provided by the previous n words, using a Markov approximation as follows:

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

The Task At Hand

Although building a n-gram model is algorithmically simple, using large corpuses of text (e.g., all Wikipedia articles is ~21GB) will present a challenging problem in terms of speed/scalability, and some good opportunities for non-trivial speedups achieved through parallelization.

The end goal is to use the language model to complete sentences, given the beginning of it. For example, upon giving the first result upon Googling "sentence generator AI"[1] the promt: *"When I got out of bed this morning…"*, It yielded the following sentence: *"When I got out of bed this morning, and walked into the office and pulled out something at the desk: a lot of work needed to be done."*

Summary of Subtasks:

1. Read file(s) containing the corpus of text to build the model from
2. Tokenize the text data into words
3. Construct the language model (i.e., build database of n-gram -> frequency counts)
4. Use the model to complete/generate sentences

[1] https://deepai.org/machine-learning-model/text-generator

Each of these sub-tasks present will present different challenges and opportunities for optimization - it's hard to say what exactly these challenges might be yet, but here are some I can think of right now:

1. Reading the file
    - since these files are going to be extremely large in size, we should probably use Lazy IO instead of reading it all into memory first.
    - **Parallelization:** The main performance improvement will be from parallelizing the construction of the language model. To do this, we can split corpus of data into *k* chunks running on *k* cores, and concurrently build frequency count databases for each chunk (i.e., sub-tasks 1-3). Then we will need to combine these databases into a unified language model.
2. Tokenizing into words
    - We may want to do some preprocessing on the words. For example, we could remove any non-ASCII characters. Also, there are different ways to tokenizing the data to consider. I can use the Python NLTK tonenize package[2] as inspiration.
3. Constructing the language model
    - We need to build the database of n-gram -> frequency counts from the text corpus
    - What data structure to use that will be both fast to build and to use (for a language model)? The obvious/naive solution would be to use a map. However, a much more efficient data structure, especially for larger *n,* is probably going to be a Trie[3].
4. Using the language model
    - Broadly speaking, given the last *n* words of a prompt, we need to gather the words of highest probability to be next word. We can perform this process recursively until a sentence is formed.
    - There many be room for algorithm optimizations that will generate more likely/realistic sentences.

To summarize, I propose to build, in Haskell, a n-gram language model using a large corpus of text with the goal of auto-completing sentences. The main opportunity for significant speedup can be achieved through parallelizing the generation of the language model. Moreover, there are several existing Python implementations to compare against.

[2] https://www.nltk.org/api/nltk.tokenize.html
[3] G. Pibiri, R. Venturini - Efficient Data Structures for Massive N -Gram Datasets