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

Project Report - COMS 4995 Parallel Functional Programming

Dave Epstein
ne2260

1 INTRODUCTION

I develop a parallelized n-gram language model in Haskell. The program works on arbitrarily sized corpora of text in any whitespace-delimited language (and can be extended to others by changing the `words` and `splitSpecialChars` functions). The program achieves around 94% of the ideal speedup given by Amdahl's law on 8 HECs (6.14×, on average) and processes the majority of Wikipedia (>3M documents)[1] into a language model in 10 minutes. It lazily builds up and merges a forest of n-gram tries and is optimized for efficient usage in common settings. When the model encounters unseen sequences of text during inference, it uses the stupid backoff strategy to compute a reasonable probability estimate. In the generative setting, the model implements a variant of diverse beam search[2] to create more interesting, varied outputs. The entire program, including imports of dependencies, comments, error handling, and command line parsing, is implemented in 352 lines. The core logic is around 150 lines.

2 BUILDING A LANGUAGE MODEL

The primary part of this program can be thought of as a function that takes in an input file of some text type and returns an n-gram in some data structure. In this section, I describe the approach I use to implement this function (don't worry, I use far more than just one function to do so). The general approach reads in text lazily with `Data.Text.Lazy.IO` to a function `splitIntoDocs`. The corpus is then tokenized by calling `tokenize` and processed into a forest of tries by calling `triesFrom`.

2.1 Processing Input

Since corpus files may be very large (I use one around 6 GB), reading it all into memory before running any processing or computation is exorbitantly wasteful. Lazy IO is a necessity, with three main options: `String`, `ByteString`, and `Text`. `Text` deals most naturally with actual, non-byte-level text at an almost negligible increase in runtime and memory, while also allowing the usage of non-ASCII characters (critical for non-English language models). `String` is a linked list of characters internally as opposed to `Text`'s packed representation, and as such is empirically worse in every way.

The raw `Text` is converted into a `Corpus` composed of `Documents`, which in turn are composed of lines of `Text`, by processing with `splitIntoDocs (pack docSep) . filter (/= empty) . lines`. `docSep`, passed in through the command line, marks the value of a line that delineates between documents in the corpus file. Each `UntokenizedLine` is transformed into a `Line` composed of `Token`s by calling `map . map tokenize`.[3] The user can also specify how many documents to read from the corpus, useful for large corpora, with the `ndocs` parameter.

Once we have the input text file segmented into documents of tokenized lines, we need to slide an n-wide window across a line to generate all possible n-grams. In practice, we want to keep even incomplete n-grams at the end of the sentence to maintain accurate word counts. `nGrams n line :: [NGram]` runs this operation, making sure to pad the beginning of the line with $n-1$ special tokens <s> indicating beginning of sentence. For example, with $n = 3$ and a sentence `["point", "free", "programming"]`, the first n-gram generated is `["<s>", "<s>", "point"]` and the last one is `["programming"]`. These n-grams are computed by first concatenating all `Line`s in a document and then running the n-wide window on the concatenated sentence.

2.2 Storing N-Grams

Now that we have described how to go from `Text -> [NGram]`, we can start talking about the fun stuff. We need an efficient way to store these n-grams in memory that matches the access patterns of realistic language model usage while not being prohibitively expensive (in either space or time) to create. I define realistic language model usage as using either of the two provided test-time functionalities: sentence probability scoring and sentence completion using random, greedy, or diverse beam search. These usage patterns require the following main operations:

(1) `getCount :: NGram -> Int`
(2) `getPrevCount :: NGram -> Int`, which can be defined as `f [] = 0` and `f x:xs = getCount xs` since `type NGram = [Token]`
(3) `merge :: OurDataStructure -> OurDataStructure -> OurDataStructure` to combine different instances (*e.g.* across documents, helpful for parallelizing)
(4) `mergeMany :: [OurDataStructure] -> OurDataStructure` which trivially follows

2.2.1 Bad: Lists and Maps. The most naïve way to store the list of `NGram`s and their counts may be `[(NGram, Int)]`. This is pretty bad because operations (1) and (2) above cost $O(n \cdot N)$ time, and the last two cost $O(n \cdot N^2)$, where N is the total number of n-grams being considered by the function and n is as in n-gram. We can not combine the first two operations into one pass either since the data structure is flat and not hierarchical. Still, some of these costs can be improved by using `Map NGram Int` or some variant, decreasing the cost[4] for operations (1) and (2) to $O(n \log N)$ and operations

[1] https://bit.ly/2OXRSkj
[2] Vijayakumar, Ashwin K., et al. "Diverse beam search: Decoding diverse solutions from neural sequence models." arXiv preprint arXiv:1610.02424 (2016).
[3] `type Token = T.Text`, `type Line = [Token]`, etc.
[4] For all reasonable values of n and N

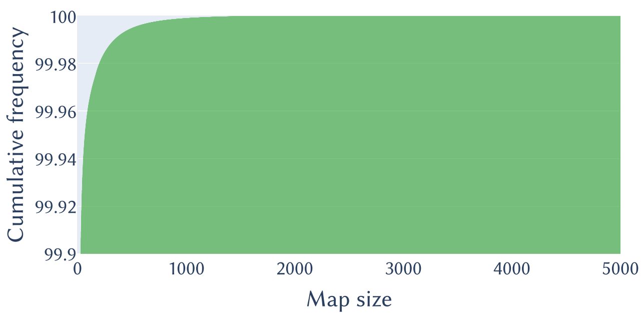

Figure 1: Cumulative distribution of map sizes at each `Trie` node, after processing 1M documents with $n = 5$. The vast majority of tries hold a mapping to under 10 children, making most lookups take effectively constant time. The mean map has 1.3 elements in it.

(3) and (4) to $O(n \cdot N)$.[5] In practice, both of these approaches are prohibitively slow and memory-hungry. They both suffer from a factor of N in all their costs since they use lists of tokens as keys. This is particularly bad (since we exactly want to work in the large-N regime) and should be avoidable with a more fitting data structure...

2.2.2 Good: Tries. Let's use tries (annoyingly pronounced just like "trees") instead. Tries are frequently used as an efficient data structure for storing strings, for applications like spell-checking (in fact, Haskell has a very good one in `Data.Trie`). However, we can't use that implementation because we wish to store a `Token` at each node and not a `Char`, so we make our own: `data Trie = Trie Int (Map Token Trie)`. Note that the `Nil` is implicitly provided by a key's absence from the map. In this formulation, an n-gram is stored recursively in a trie, starting from a root node. Now, ops (1) and (2) take $O(n + \log V)$ time on average, where V is the size of the vocabulary we consider[6], with no added cost of calling both together (see L155-158 of `Lib.hs`). Figure 1 shows the sensibility of these complexity derivations. Ops (3) and (4) now take $O(n)$ time on average.[7] Indeed, this is much better than using one global list or map. Note that even in the worst-case scenario, the factor of N is *entirely absent* from big-O costs (and $V << N$).

Storing Children. Since we use a recursive data structure, we need some dynamic way to store children. Above, I suggest the `Map`, but we can use `List` or `HashMap` instead (or, stupidly, `IntMap` with a separately maintained `Token -> Int` function, but this is a worse version of what `HashMap` does internally). Empirically, `Map` and `HashMap` are best, and they perform almost identically. I suspect the overhead of `HashMap` cancels out its faster lookup and merge operations in a typical use-case. For simplicity, I just use `Map`.

Strictness. Having chosen `Map` to store `Trie`s recursively, the decision between a strict and lazy version remains, with profound effects on parallelism. The lazy version offloads all work on its keys to an on-demand basis, which is much less parallelizable than fully evaluating a key as part of a sparked job. As such, performance significantly improves by switching from `Map.Lazy` to `Map.Strict`.

Tries as Monoids. Joyously, our `Trie`s are `Monoid`s (see L23-28 of `Lib.hs`). We now have operations (3) and (4) given by `(<>)` and `mconcat`, and elegantly defined using `Map.unionWith (<>)` to recurse.

2.3 Constructing a Model

Now, all that is left is to define a function `triesFrom :: Corpus -> Trie` (the actual type signature in code is slightly different, and returns a forest of tries `[Trie]`). This function builds a trie for each document separately using `buildTrie :: [NGram] -> Trie`, optionally prunes infrequent paths in the trie with `pruneTrie`, and then optionally merges the tries using `mergeTries` (which utilizes `mconcat` but is not identical to it when parallelizing). Of course, this structure is designed with parallelism at the forefront, and it is rich in `map`s of non-trivial functions that are well-suited to running concurrently on separate cores.

We define `buildTrie = foldl' insertNGram mempty`, and `insertNGram` traverses the input trie, calling itself to appropriately augment nodes along the n-gram's path (L126-134 in `Lib.hs`). Thus, `buildTrie` inserts n-grams one at a time starting from an empty trie. This operation is $O(n \cdot N)$ where there are N n-grams in the input to the function.[8] Note that this time complexity increases to $O(N^2)$ on one large list and $O(N \log N)$ on one large map (the naïve structures discussed in Section 2.2.1), but stays the same if we were to use a list instead of a map to store elements at trie nodes. In practice, there is some constant factor that scales cost which is exponentially larger when using lists (thanks to `Map`'s $O(\log N)$ insert that doesn't require a linear search).

The tries can then be optionally processed to prune low-frequency paths to save memory and improve runtime slightly, and merged into one final large trie. Thanks to Haskell's laziness, though, it is *much* faster not to eagerly merge all tries into one (very expensive and, since it is a `fold`, not good for parallelizing). Instead, we lazily merge tries when it is absolutely necessary, preferring strongly to run independent computations on each trie and merge their results, as discussed later in Section 4.3.

3 USING A LANGUAGE MODEL

Now that we have a built language model (in the form of a trie forest) we should use it to do some cool things. In the project proposal, I mentioned diverse beam search, which I implement and discuss in Section 3.2.2. I also implement stochastic and greedy sentence completion (the latter is beam search with $\beta = 1$), and write a simple function to evaluate the likelihood of a sentence given the training corpus. All the above functionalities use stupid backoff internally. This functionality can be used, *e.g.*, to discriminate between candidate speech transcripts or sentence translations to find the most fluent one.

Since both general usage modes follow the same "prompt-compute-return" loop, I write a function `promptLoop :: String -> (String`

[5] Assuming the two maps never differ in size by more than some large constant factor
[6] The log V term comes from the fact that every word in the vocabulary will appear in the root node's `Map`. If every word realistically could follow every other word in some language, this becomes $O(n \log V)$. In practice, that does not happen.
[7] With the same assumptions about the long-tail distribution of language
[8] Again under the assumption of very low average map size (or some constant upper bound)

Sentence	Score
he briefed reporters on the main contents of the statement	4.546
he introduced reporters to the main contents of the statement	0.535
he briefed to reporters the main contents of the statement	0.191

Figure 2: Sentence probability scores for three candidate sentence translations (higher means more likely).[9] The language model is able to distinguish between small changes to select the most fluent translation.

`-> IO String) -> IO ()` which takes in a prompt string and a `stdin`-to-output function and generalizes control flow.

3.1 Sentence Probability

The `lineScore` function takes in the trie forest and an untokenized line (which is transformed into `[NGram]` using text processing functions described in Section 2.1) and returns a *probability score* defined as:

$$s(\{w_i\}_0^N) = \sum_{\{w_i\}_0^N} \log P(w_i|w_{i-n}, \ldots, w_{i-1})$$
$$P(w_i|w_{i-n}, \ldots, w_{i-1}) = \frac{C(w_{i-n}, \ldots, w_i)}{C(w_{i-n}, \ldots, w_{i-1})} \tag{1}$$

We work in log space and add to avoid numerical underflows. In practice, when $C(w_{i-n}, \ldots, w_i) = 0$, the model defines:

$$P(w_i|w_{i-n}, \ldots, w_{i-1}) = \lambda \cdot P(w_i|w_{i-n+1}, \ldots, w_{i-1}) \tag{2}$$

This shrinks the context by one gram and scaling the score down by a constant factor λ (I use $\lambda = 0.4$). Since this formulation no longer yields a probability distribution where all terms sum up to 1, some NLP literature prefers using S for score instead of P for probability. I avoid this notation here for conciseness.

The function `nGramScore` returns the probability score for one n-gram by computing the `getCount` and `getCountUpto` values sketched out in Section 2.2. This function is interesting since it computes these counts for each trie in the forest separately and parallelly, combines them by summing tuples, then computes the final score. This is several times faster than lazily merging the tries and taking the counts in the final merged trie, highlighting the importance of correct order of operations. Further, each n-gram's score is computed independently and is parallelized.

Figure 2 demonstrates this functionality and shows some scores assigned to three candidate translations output by a Chinese-to-English translator. As expected, the most fluent sentence gets the highest score.

3.2 Sentence Completion

The `lineComplete` function does some processing on the input string received by `promptLoop` and feeds its last n-gram to either of two monadic text-generation functions: `randomSearch` or `beamSearch`. Both these functions call themselves to generate the next token based on previous input and output. They terminate with a 1% chance at each iteration, giving an expected sentence length of 100 tokens. This parameter is arbitrary and can easily be consumed as a command line parameter if desired.

(1) The first $20 million is always the hardest, because it takes years to build up the amount of existing support that an open entry creates. But in general, once a winner is determined, it's difficult for anybody to "squeeze" the other bidders out of the process.
(2) The first blast of the trumpet comes from right in the middle of the crowd, and it hits just above the one chord the trumpet has needed to play: B. You can use the same concept here for things like A, E and G.
(3) The first Fat Truckers album is for sale, a 12-inch that features four songs—one for each of the five main intersections of each company's North American route—with song titles such as "All Of These Cities Won't Be There For Long" and "Up On The Roof."
(4) The first circle of the Coronado Peninsula was closed off on March 29th, 1962 in an attempt to create a reef habitat that would protect the southern edge of the Coronado Peninsula from encroachment by an expanding freeway.
(5) The first four years of the war involved about 110,000 combatants and 140,000 civilians. The troops and civilian forces suffered several thousand casualties in the first six months, with about 650,000 people being injured.

Figure 3: "The first…": Some sentence completions with diverse beam search ($\beta = 5$). Capitalization and spacing were adjusted for readability.

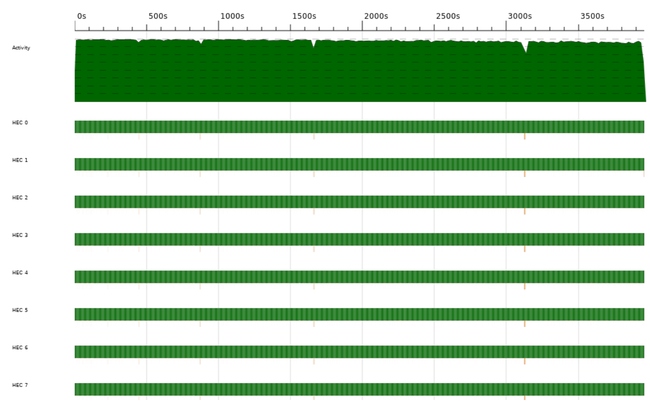

Figure 4: Building a forest with `triesFrom` on 1M documents with 8 cores. Threadscope shows near-perfect utilization throughout the entire program runtime (full evaluation is forced using `deepseq`). Note that times on the top axis are not wall-clock.

3.2.1 *Stochastic.*

In stochastic text generation of n-grams, the i^{th} word is chosen with probability $P(w_i|w_{i-n}, \ldots, w_{i-1})$ as defined in Equation 1. A random number $r \in [0, 1]$ is drawn and used to select the next word, where each word covers up an interval of the $[0, 1]$ number line of length $P(\cdot)$. Computing this value across a forest of tries for many words is somewhat expensive, so we merge trie nodes as necessary with a light wrapper around `mconcat` and run the computations on the result in the `randomSearchTok` function.

In this variant of stupid backoff, if the previous $n - 1$ words of context never appeared in sequence in the training set, the function resorts to using $n - 2$ words of context instead. Outputs from this function are not insensible, since they are conditioned on training data, but are still essentially nonsensical over long sequences. To alleviate this, we need a better sampling strategy...

3.2.2 *Diverse Beam Search.*

One slight adjustment we can make to random search to output likelier sentences is to sample greedily

[9]Speech and Language Processing (3rd ed. draft), Jurafsky and Martin
[9]https://www.oreilly.com/library/view/parallel-and-concurrent/9681449335939/ch02.html

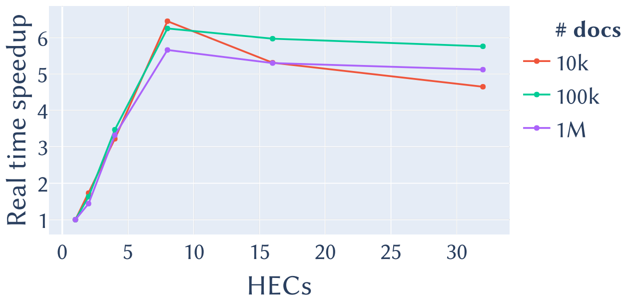

Figure 5: Real time speedup on 1-32 cores ($n = 5$). Best performance is with 8 HECs which achieves up to a 6.5× speedup, 94% of the limit given by Amdahl's law.

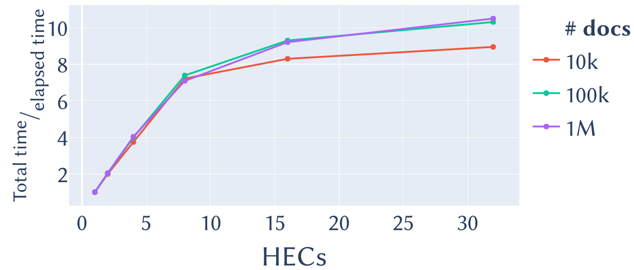

Figure 6: Total to elapsed time ratio on 1-32 cores ($n = 5$). Marlow[10]uses this metric to measure the effectiveness of code parallelization. This ratio gives some intuition into the peak at real time speedup at 8 HECs: the effectiveness increases at a much slower rate after that point.

instead, picking $\arg\max_{w_i} P(w_i)$. However, these sentence tend to be quite boring (think of spamming the middle suggestion given by your phone's autocomplete) and do not properly display the breadth of language learned by even our simple n-gram model.

To reintroduce diversity into sentence completions, we can use a greedy search where we take the β best alternatives from the previous iteration (or the input sentence, at iteration 0), expand them, pick the top β, and repeat. Of course, with $\beta = 1$ this is a regular greedy search. In practice, $\beta \in [3, 20]$ is used, and I set $\beta = 5$. While standard beam search somewhat improves the variety of outputs from the model, it still tends to have high overlap between different beams of output.

To discourage this, the score assigned to an n-gram can be augmented with a diversity constraint. The simplest one, which is often used in practice with good results, assigns a penalty α to the i^{th} word of the b^{th} beam w_i^b if any of $w_i^{b'} = w_i^b$ $(b \neq b')$. With this system in place ($\alpha = 20$ in practice), the model outputs an interesting variety of sentences (as in Figure 3).

4 PARALLELIZING

While the structures described above were of course designed with parallelism in mind, they do not require it to run. On one core, the program can process the entirety of the $> 3M$-article Wikipedia corpus into a language model in under an hour. When multiple cores are brought into the equation and the program is intelligently parallelized (there are a few small pitfalls in algorithm design that, when avoided, make a big difference), speedups of 6–7× are possible, in close agreement with the theoretical limit yielded by Amdahl's law.

4.1 Amdahl Strikes Again

To compute the parallelizable portion of the task P in $S = \frac{1}{(1-P)+\frac{P}{N}}$, I run `untokCorpus `deepseq` ()` to force the full processing of the text file on one core (the tokenization is parallelized, so is not included in this calculation). I then run the full computation of the trie forest also on one core and set P as the ratio of these two times. Averaging across three runs, I find $P \approx 2.95\%$, giving a maximum theoretical speedup of 6.61× on 8 HECs or 33.9 as $N \to \infty$. In practice, the overhead of running on more cores seems to start overpowering the benefits after 8 HECs. Figure 5 shows wall-clock speedup on different corpus sizes as a function of number of cores.

4.2 A Universal Parallelization Strategy

Compared to most other languages, Haskell has a truly beautiful ability to parallelize. I define a helper function `par' = (`using` parBuffer bufferSize rdeepseq)` which I apply systematically to eight different `map` operations, which range from tokenizing corpous documents to computing beam search candidates. Each one of these improvements significantly improves speedup metrics and together they yield a very satisfying graph of CPU utilization (Figure 4).

Further indicating the effectiveness of this simple approach is the total-to-elapsed ratio in Figure 6 (computed by running programs with `+RTS -s`) that Marlow uses to quantify the power of going parallel. At 32 HECs, this ratio tops out at 10.5. I opt for `parBuffer` instead of `parList` because they behave similarly on shorter lists but the former is not strict in the spine of the list, especially important for the size of corpus being considered in this project. I find this universal, relatively naïve approach to be highly effective.

The program scales effectively (around $O(N)$) to large corpora. Figure 7 shows program run-time as a function of corpus size. One could even download a more recent dataset (~3.3B words) used to train state-of-the-art machine learning models, which is around double the size of the corpus I use, and build an n-gram model in under half an hour on an 8-core machine!

4.3 Lazy N-Gram Merging

A sensible approach, and indeed the one I initially adopted, is to eagerly merge all the tries collected for each document into one big trie. This becomes quite an expensive operation when merging larger tries, and tries get large very fast with a function like `foldl' mergeTwoTries mempty`. One way to mitigate this damage is to split the forest of tries into subforests of a fixed small size (say k), merge each subforest separately and parallely, and return a new forest (smaller by a factor of k), repeating until only one large trie is left. This approach is already significantly faster empirically, and is easily implemented in `mergeTries` (L146-150), but still slows down runtime since the last larger merges keep only a few HECs busy. The

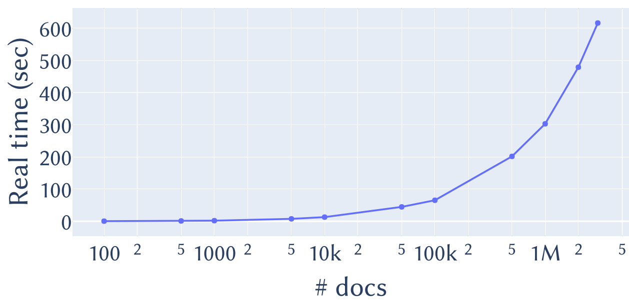

Figure 7: Execution time on 8 cores on corpora of 100-3M documents. The x-axis is in log scale to show finer patterns in data. Note that the function appears to grow on the order of N, the size of the corpus.

key observation is that using the language model does *not* require pre-merging the forest into one large trie. Many operations (such as assigning a sentence a probability, as in Section 3.1) can be done on each trie separately and cheaply merged even across millions of tries. Other operations do require merging tries to sensibly implement, but merging the trie nodes belonging to some (on average, quite infrequent) n-gram is many times cheaper than pre-merging the whole forest. Offsetting the merge operation either to cheap data derived from each trie separately or to much smaller subtries was probably the single largest contributor to faster runtime.

5 USAGE

The program can be simply built by executing `stack build` from the program directory. It can then be run with `stack exec -- stupidlm-exe [OPTIONS]`. Running without any options yields the following helptext:

```
Usage: stupidlm-exe (-m|--mode MODE) (-f|--corpus-file FILENAME)
                    [-n|--ngrams NUMBER] [--ndocs NUMBER]
                    [-s|--doc-separator SEPARATOR]
                    [-t|--freq-threshold NUMBER]
                    [--complete-mode MODE]
Build a language model from a given corpus, then use it to assign
probability scores to input lines, or to auto-complete them
(similar to your smartphone keyboard).

Available options:
  -h,--help                Show this help text
  -m,--mode MODE           Operating mode (buildOnly, score, complete)
  -f,--corpus-file FILENAME
                           Path to corpus file
  -n,--ngrams NUMBER       Size of n-gram (default: 3)
  --ndocs NUMBER           Number of documents to parse (-1 to parse
                           all) (default: -1)
  -s,--doc-separator SEPARATOR
                           Line that separates documents in
                           corpus (default: "---END.OF.DOCUMENT---")
  -t,--freq-threshold NUMBER
                           Prune all n-grams that occur fewer than k
                           times (default: 0)
  --complete-mode MODE     Operating mode if --mode is complete
                           (beamSearch, greedy, random)
                           (default: "beamSearch")
```

The `buildOnly` mode processes a forest of tries and `deepseq`s through them to force full evaluation. The other two modes implement the functionalities described in Section 3.

6 CODE (352 LINES)

src/Lib.hs

```haskell
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE ViewPatterns      #-}

module Lib (splitIntoDocs, tokenize, pruneTrie, triesFrom, lineScore, lineComplete,
  Trie, docNGrams, par') where

import           Control.DeepSeq            (NFData, rnf)
import           Control.Monad              (ap, join)
import           Control.Parallel.Strategies (parBuffer, rdeepseq, using)
import           Data.Char                  (isPunctuation, isSymbol)
import           Data.List                  (foldl', foldl1', intercalate,
                                             sortBy, unzip3, zip4)
import           Data.List.Split            (chunksOf, divvy, splitOn)
import qualified Data.Map.Strict            as M
import           Data.Maybe                 (mapMaybe)
import qualified Data.Text.Lazy             as T
import           System.Random              (randomRIO)

--- Types ---
data Trie = Trie Count (M.Map Token Trie)

instance Semigroup Trie where
  (Trie c1 m1) <> (Trie c2 m2) = Trie (c1 + c2) $ M.unionWith (<>) m1 m2

instance Monoid Trie where
  mempty = Trie 0 M.empty
  mconcat = foldl1' (<>)

instance NFData Trie where
  rnf (Trie c m) = rnf c `seq` rnf m

instance Show Trie where
  show t = show_ t 1
    where
      show_ (Trie c m) d
        | null m = show c
        | otherwise = "(" ++ show c ++ ", " ++
          dictShow (sortBy (\(_, Trie a _) (_, Trie b _) -> compare b a) (M.toList m)) d
          ++ ")"
      dictShow l d = "{" ++ intercalate ", " (map (\(k, v) -> "\n" ++
                replicate (d * 2) ' ' ++ show k ++ ": " ++ show_ v (d + 1))
              l) ++ "\n" ++ replicate ((d - 1) * 2) ' ' ++ "}"

-- Types for building the trie --
type Token = T.Text

type Line = [Token]

type NGram = [Token]

type Document = [Line]

type Corpus = [Document]
```

```
type UntokenizedLine = T.Text

type UntokenizedDocument = [UntokenizedLine]

type UntokenizedCorpus = [UntokenizedDocument]

-- Types for using the trie --
type Count = Int

type ApproxCount = Float

type Score = Float

type BeamState = (NGram, Score, Line)

type PotentialBeamState = (NGram, Score, Line, [(Token, Score)])

--- Constants ---
bosToken :: Token
bosToken = "<s>"

bufferSize :: Int
bufferSize = 100

chunkSize :: Int
chunkSize = 10

backoffWt :: Float
backoffWt = 0.4

penaltyWt :: Float
penaltyWt = 20

--- Utility functions ---
par' :: NFData a => [a] -> [a]
par' = (`using` parBuffer bufferSize rdeepseq)

--- Functions for processing text ---
splitIntoDocs :: UntokenizedLine -> [UntokenizedLine] -> UntokenizedCorpus
splitIntoDocs docSep = splitOn [docSep]

naiveNGrams :: Int -> Line -> [NGram]
naiveNGrams _ []       = []
naiveNGrams n l@(_:xs) = take n l : naiveNGrams n xs

nGrams :: Int -> Line -> [NGram]
nGrams n toks = naiveNGrams n (replicate (n - 1) bosToken ++ toks)

-- Adapted from Data.Text's split function to keep separators
splitKeepSeps :: (Char -> Bool) -> T.Text -> [T.Text]
splitKeepSeps _ t@(T.null -> True) = [t]
splitKeepSeps p t = loop t
  where
    loop s
      | T.null s' = [l]
      | otherwise = l : T.singleton (T.head s') : loop (T.tail s')
      where
```

```haskell
        (l, s') = T.break p s

tokenize :: UntokenizedLine -> Line
tokenize = filter (/= T.empty) . splitSpecialChars . T.words . T.toLower
  where
    splitSpecialChars =
      concatMap $ splitKeepSeps $ oneOf [isPunctuation, isSymbol]
    oneOf ps = or . ap ps . return

--- Functions for building n-gram tries ---
docNGrams :: Int -> Document -> [NGram]
docNGrams = (. join) . nGrams

insertNGram :: Trie -> NGram -> Trie
insertNGram (Trie c m) [] = Trie (c + 1) m
insertNGram (Trie c m) (gram:grams) = Trie (c + 1) $ M.alter go gram m
  where
    go Nothing  = Just $ insertNGram mempty grams
    go (Just t) = Just $ insertNGram t grams

buildTrie :: [NGram] -> Trie
buildTrie = foldl' insertNGram mempty

allTriesFrom :: Int -> Corpus -> [Trie]
allTriesFrom n = par' . map (buildTrie . docNGrams n)

triesFrom :: Int -> Count -> Corpus -> [Trie]
triesFrom n = (par' .) . (. allTriesFrom n) . mapMaybe . pruneTrie

pruneTrie :: Count -> Trie -> Maybe Trie
pruneTrie gramThreshold (Trie c m)
  | c < gramThreshold = Nothing
  | otherwise = Just $ Trie c $ M.mapMaybe (pruneTrie gramThreshold) m

mergeTries :: [Trie] -> Trie
mergeTries [ts] = ts
mergeTries ts = mergeTries $ mergeTriesOnce ts
  where
    mergeTriesOnce = par' . map mconcat . chunksOf chunkSize

--- Functions for using n-gram tries to compute probability of text ---
-- (Count of w_i...w_i+n-1, Count of w_i...w_i+n-2)
counts :: NGram -> Count -> Maybe Trie -> (Count, Count)
counts _ prev_c Nothing                 = (0, prev_c)
counts [] prev_c (Just (Trie c _))      = (c, prev_c)
counts (gram:grams) _ (Just (Trie c m)) = counts grams c $ M.lookup gram m

countsStupidBackoff :: NGram -> Maybe Trie -> (ApproxCount, ApproxCount)
countsStupidBackoff [] _ = (backoffWt ^ (100 :: Int), 1) -- Word not seen in training
countsStupidBackoff ngram trie
  | c == 0 =
    let (c_bo, prev_c_bo) = countsStupidBackoff (tail ngram) trie -- Back off stupidly
     in (backoffWt * c_bo, prev_c_bo)
  | otherwise = (fromIntegral c, fromIntegral prev_c)
  where
    (c, prev_c) = counts ngram 0 trie
```

```haskell
nGramScore :: [Trie] -> NGram -> Score
nGramScore tries ngram =
  let (c, prev_c) =
        sumTuples (par' $ map (countsStupidBackoff ngram . pure) tries)
   in log c - log prev_c
  where
    sumTuples = foldl1' (\(a1, b1) (a2, b2) -> (a1 + a2, b1 + b2))

lineScore :: [Trie] -> Int -> UntokenizedLine -> Score
lineScore tries n =
  sum . par' . map (nGramScore tries) . divvy n 1 .
  (replicate (n - 1) bosToken ++) . tokenize

--- Functions for using n-gram tries to generate text ---
trieFind :: NGram -> Maybe Trie -> Maybe Trie
trieFind [] t                             = t
trieFind _ Nothing                        = Nothing
trieFind (gram:grams) (Just (Trie _ m)) = trieFind grams $ M.lookup gram m

trieFindMerge :: NGram -> [Trie] -> Trie
trieFindMerge n = mergeTries . par' . mapMaybe (trieFind n . Just)

topBToks :: Int -> Trie -> [(Token, Score)]
topBToks b (Trie c m) =
  map (\(tok, Trie c1 _) -> (tok, log (fromIntegral c1) - log (fromIntegral c))) $
  take b $ sortBy (\(_, Trie c1 _) (_, Trie c2 _) -> compare c2 c1) $ M.toList m

-- Penalize with Hamming distance, number of shared (ordered) tokens between two lines
-- sim("Functional programming is cool and advanced", "Advanced Programming is boring") = 2
diversify :: [BeamState] -> [(BeamState, Score)]
diversify = f []
  where
    f _ [] = []
    f hs (x@(_, _, h):xs) =
      (x, penaltyWt * fromIntegral (sim h hs)) : f (h : hs) xs
    sim _ [] = 0
    sim h hs =
      length $ filter id $ foldl1' (zipWith (||)) $ map (zipWith (==) h) hs

-- Each current state has b candidate expansions
-- We define state as a tuple of score, history, current n gram, and list of expansions
-- Process each expansion into its own state by merging its information with parent state
scoreCandidates :: [PotentialBeamState] -> [(BeamState, Score)]
scoreCandidates = diversify . concatMap f
  where
    f (n, s, h, cs) = map (\(tok, s') -> (tail n ++ [tok], s + s', tok : h)) cs

beamSearch :: Int -> [Trie] -> [NGram] -> [Score] -> [Line] -> UntokenizedLine
 -> IO String
beamSearch b tries ngrams scores hists input = do
  randExitFlag <- randomRIO (0, 100 :: Int)
  let histStr = unlines $ map (T.unpack . T.unwords . (input :) . reverse) hists
      beamTries = par' $ map ((`trieFindMerge` tries) . tail) ngrams
      candidates = par' $ map (topBToks b) beamTries
      scoredCands = scoreCandidates $ zip4 ngrams scores hists candidates
      sortedCands = sortBy finalScoresAsc scoredCands
      (ngrams', scores', hists') = unzip3 $ map fst $ take b sortedCands
```

```haskell
  if randExitFlag == 0
    then return histStr
  else beamSearch b tries ngrams' scores' hists' input
  where finalScoresAsc ((_, s1, _), p1) ((_, s2, _), p2) = compare (s2 - p2) (s1 - p1)

wordAtIndex :: Int -> [(Token, Trie)] -> Token
wordAtIndex _ [] = "" -- Couldn't find any child word (i.e. method was called on leaf)
wordAtIndex i ((tok, Trie c _):rest)
  | i <= 0 = tok
  | otherwise = wordAtIndex (i - c) rest

randomSearchTok :: NGram -> [Trie] -> IO Token
randomSearchTok ngram tries = do
  let (Trie c m) = trieFindMerge ngram tries
  idx <- randomRIO (0, c - 1)
  let tok = wordAtIndex idx $ M.toList m
  if T.null tok -- Couldn't find next word, back off stupidly!
    then randomSearchTok (tail ngram) tries
  else return tok

randomSearch :: [Trie] -> NGram -> IO String
randomSearch _ [] = error "Random search failed" -- Not reachable
randomSearch tries (_:grams) = do
  randExitFlag <- randomRIO (0, 100 :: Int)
  if randExitFlag == 0
    then return "..."
  else do
    tok <- randomSearchTok grams tries
    rest <- randomSearch tries (grams ++ [tok])
    return $ T.unpack tok ++ " " ++ rest

lineComplete :: String -> [Trie] -> Int -> UntokenizedLine -> IO String
lineComplete mode tries n line = do
  let lastNGram =
        (last . divvy n 1 . (replicate (n - 1) bosToken ++) . tokenize) line
      launchBeamSearch b = beamSearch b tries [lastNGram] [0] [[]] line
  case mode of
    "beamSearch" -> launchBeamSearch 5
    "greedy" -> launchBeamSearch 1
    "random" -> do
      putStr $ T.unpack line ++ " "
      randomSearch tries lastNGram
    _ ->
      error $ "Mode " ++ mode ++
      " is unsupported. Please choose from [beamSearch, greedy, random]."
```

app/Main.hs

```haskell
module Main where

import           Control.DeepSeq     (deepseq)
import           Control.Monad       (join)
import           Data.Monoid         ((<>))
import qualified Data.Text.Lazy      as T
import qualified Data.Text.Lazy.IO   as TIO
import           Lib
import           Options.Applicative
import           System.Exit         (die, exitSuccess)
```

```haskell
import           System.IO           (hFlush, stdout)

-- Adapted with permission from https://bit.ly/35WGzyB
main :: IO ()
main = join . customExecParser (prefs showHelpOnError) $
  info (helper <*> parser)
    (fullDesc <>
     header
       "Parallelized N-Gram Language Modeling with Stupid Backoff for Text Generation" <>
     progDesc ("Build a language model from a given corpus, then use it to assign" ++
     " probability scores to input lines, or to auto-complete them" ++
     " (similar to your smartphone keyboard)."))
  where
    parser :: Parser (IO ())
    parser =
      work <$>
      strOption
        (long "mode" <> short 'm' <> metavar "MODE"
        <> help "Operating mode (buildOnly, score, complete)") <*>
      strOption
        (long "corpus-file" <> short 'f' <> metavar "FILENAME"
        <> help "Path to corpus file") <*>
      option auto
        (long "ngrams" <> short 'n' <> metavar "NUMBER" <> help "Size of n-gram"
        <> value 3 <> showDefault) <*>
      option auto
        (long "ndocs" <> metavar "NUMBER" <> value (-1) <> showDefault <>
         help "Number of documents to parse (-1 to parse all)") <*>
      strOption
        (long "doc-separator" <> short 's' <> metavar "SEPARATOR" <>
         value "---END.OF.DOCUMENT---" <> showDefault <>
         help "Line that separates documents in corpus") <*>
      option auto
        (long "freq-threshold" <> short 't' <> metavar "NUMBER" <> value 0 <> showDefault
         <> help "Prune all n-grams that occur fewer than k times") <*>
      strOption
        (long "complete-mode" <> metavar "MODE" <> value "beamSearch" <> showDefault <>
         help "Operating mode if --mode=complete (beamSearch, greedy, random)")

work :: String -> String -> Int -> Int -> String -> Int -> String -> IO ()
work mode corpusFile n nDocs docSep thresh completeMode = do
  corpusText <- TIO.readFile corpusFile
  let untokCorpus =
        splitIntoDocs (T.pack docSep) $ filter (/= T.empty) $ T.lines corpusText
      corpus = par' $ map (map tokenize) untokCorpus
      maybePartialCorpus = if nDocs < 0 then corpus else take nDocs corpus
      nGramTries = triesFrom n thresh maybePartialCorpus
  if mode == "buildOnly" then do
    putStrLn "Building n-gram model..."
    putStrLn $ nGramTries `deepseq` "Built."
    exitSuccess
  else if mode == "score" then scoreLoop nGramTries n
  else if mode == "complete" then completeLoop nGramTries n completeMode
  else die $ "Mode must be one of [score, complete]. You gave " ++ mode ++ "."

promptLoop :: String -> (String -> IO String) -> IO ()
promptLoop prompt f = do
```

```
  putStr prompt
  hFlush stdout
  line <- getLine
  if null line
    then return ()
  else do
    out <- f line
    putStrLn out
    promptLoop prompt f

scoreLoop :: [Trie] -> Int -> IO ()
scoreLoop tries n = promptLoop "Score> " $ return . show . lineScore tries n . T.pack

completeLoop :: [Trie] -> Int -> String -> IO ()
completeLoop tries n mode = promptLoop "Complete> " $ lineComplete mode tries n . T.pack
```