N-gram models

Computational Linguistics (LING 455)

Rutgers University September 23, 2021

Motivating *n*-grams

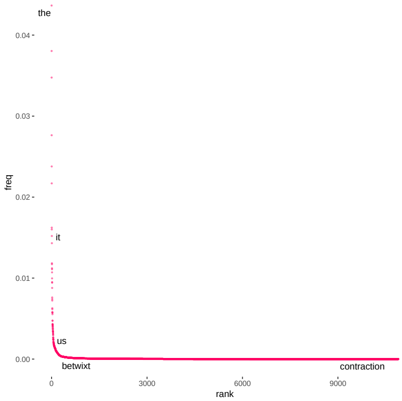
What we can do

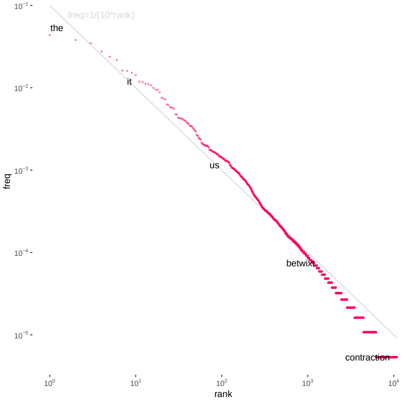
$$\begin{tikzcd}
\text{raw text} \arrow[d] & \texttt{String} \arrow[d] & \texttt{the dog. The} \arrow[d] \\
\text{cleaned} \arrow[d] & \texttt{String} \arrow[d] & \texttt{the dog the} \arrow[d] \\
\text{tokens} \arrow[d] & \texttt{[String]} \arrow[d] & \texttt{["the","dog","the"]} \arrow[d] \\
\text{types} \arrow[d] & \texttt{[[String]]} \arrow[d] & \texttt{[["dog"],["the","the"]]} \arrow[d] \\
\text{with counts} \arrow[d] & \texttt{[(Int, String)]} \arrow[d] & \texttt{[(1,"dog"),(2,"the")]} \arrow[d] \\
\text{with freqs} & \texttt{[(Double, String)]} & \texttt{[(.33,"dog"),(.67,"the")]}
\end{tikzcd}$$

As Haskell code

```haskell
-- from W3.hs
main :: IO ()
main = do
  raw <- readFile "files/great-expectations.txt"
  let cleaned = clean raw
  let tokens = tokenize cleaned
  let types = getTypes tokens
  let withCounts = sort (map addCount types)
  let finally = relFreqs (length tokens) withCounts
  writeFile "out/freqs.txt" (unlines (map show finally))
```

Zipfian distribution for *Great Expectations*





What’s missing?

Language is more than a jumble of words!

- We want to keep track of **sentences** (and, eventually, of phrases/constituents smaller than sentences).
- We want to understand how words can be arranged in a text. This requires us to keep track of **order** in some way.

Last week was a great starting point, but we largely ignored these:

- We didn’t even bother trying to segment text into sentences.
- Grouping word tokens into types forgets everything about how those words were arranged to begin with.

Sentences, and the Brown corpus

We'll be working with the Brown corpus this week (link). It's bigger and, of course, more modern than `great-expectations.txt`.

- And it very helpfully tells us where sentences end!

Some versions of the corpus delimit sentences with three spaces `␣␣␣` or a double line-break. Others give each sentence **its own line**.

In the latter case, chunking into sentences is easy:

```
*W4> lines "Here is one sentence\nHere is another"
["Here is one sentence","Here is another"]
```

Chunking the Brown corpus into sentences

```haskell
getSentences :: String -> [String]
getSentences = lines
```

```haskell
main = do
  s <- readFile "files/brown_notags.txt"
  let sents = getSentences s
  -- ...
```

Cleaning and tokenizing each sentence

We **individually** clean and tokenize these sentences, using `map`:

```haskell
import W3 hiding (main)
-- ...
main = do
  s <- readFile "files/brown_nolines.txt"
  let sents    = getSentences s
  let tokened = map (tokenize . clean') sents
  --                ^^ \sen -> tokenize (clean' sen)
```

```haskell
clean' xs = filter (\x -> not (isControl x)) (clean xs)
-- scrubbing out some extra garbage
-- equiv to `clean' = filter (not . isControl) . clean`
```

An example

```
*W4> raw <- readFile "files/brown_notags.txt"
*W4> let sents = getSentences raw
*W4> let tokened = map (tokenize . clean') sents
*W4> head tokened
["the","fulton","county","grand","jury","said","friday",
"an","investigation","of","atlantas","recent","primary",
"election","produced","no","evidence","that","any",
"irregularities","took","place"]
```

Probabilities of ~~words~~ sentences

A corpus gives quantitative info about how frequently different kinds of linguistic events are observed. So far we have restricted attention to the frequencies or probabilities of words.

Let's start to think bigger. The frequency, or **probability**, of a chunk of words (say, a sentence) is gotten at by counting up the occurrences of that chunk, and dividing by the number of same-length sequences:

- $P(\texttt{I like pizza}) = \dfrac{C(\texttt{I like pizza})}{C(\textit{all 3-word sequences})}$ $\qquad$ (C = count)

This is a *more general* notion than the frequency of a word. Word frequencies are what happens when we have 1-word sequences!

Alas

This doesn't take us very far. If words are Zipfian, then sentences *really* are.[1] Almost every sentence in the Brown corpus is unique!

What's more, language is fundamentally **creative**. It's pretty easy to say something that's utterly novel in the history of humanity.

- No matter how big your corpus, it's going to give $P(s) \approx 0$ for almost any sufficiently interesting sentence s.

Yet obviously some sentences are likelier than others:

- $P($`I like pizza`$) > P($`Red ideas rest`$) > P($`Like pizza I`$)$

How should we calculate, or more likely, *estimate* such a P?

[1] In the admittedly imprecise sense that the infrequent stuff is *extremely* infrequent.

Another formulation

Conditional probabilities: $P(x \mid y)$ is the likelihood of x **given** y.

- Rain might be pretty unlikely in southern California, but it might be likelier given the approaching thunderclouds.
- $P(\texttt{to}) < P(\texttt{to} \mid \texttt{want})$

The probability of sequence of words is the product of, for each word, the probability of observing that word given what came before:

- $P(\texttt{I like pizza}) = P(\texttt{I})P(\texttt{like} \mid \texttt{I})P(\texttt{pizza} \mid \texttt{I like})$

This is an application of the **chain rule of probability**:

- $P(X_1 \ldots X_n) = \prod_{k=1}^{n} P(X_k \mid X_1 \ldots X_{k-1})$

Problem

The problems with this method are the same as before:

- Word sequences are Zipfian. Most sequences have vanishingly small frequencies, even in outlandishly big corpora.
- Language is creative, and this only gives non-0 probabilities for things that have been observed before.

A guessing game

The ␣

A guessing game

```
The ␣
The girl ␣
```

A guessing game

```
The ␣
The girl ␣
The girl walked ␣
```

A guessing game

```
The ␣
The girl ␣
The girl walked ␣
The girl walked the ␣
```

A guessing game

```
The ␣______␣
The girl ␣______␣
The girl walked ␣______␣
The girl walked the ␣______␣
The girl walked the bicycle
```

How did you guess?

`The girl walked the ␣`

How did you guess?

`The girl walked the ␣`

You might base this on the entire sentence — you may have heard a few instances of "The girl walked the *X*" in your life.

How did you guess?

`The girl walked the ___`

You might base this on the entire sentence — you may have heard a few instances of “The girl walked the *X*” in your life.

More likely: you focused on the phrase “walked the *X*”:

```
walked the dog      >
walked the bicycle  >
walked the soda
```

Markov assumption

Approximate the probability of w_n given prior context $w_1 \dots w_{n-1}$ as the probability of w_n given *the **immediately** preceding words.*

- 1 word $P(w_n \mid w_1 \dots w_{n-1}) \approx P(w_n \mid w_{n-1})$
- 2 words $P(w_n \mid w_1 \dots w_{n-1}) \approx P(w_n \mid w_{n-2} w_{n-1})$
- …

n-gram Hypothesis

We can reliably predict the n^{th} word based on the last $n-1$ words.

Bigrams

What are the **bigrams** for the following sentence?

`the quick brown fox jumped over the lazy dog`

Bigrams

What are the **bigrams** for the following sentence?

`the quick brown fox jumped over the lazy dog`

Bigrams

What are the **bigrams** for the following sentence?

```
the quick brown fox jumped over the lazy dog
```

Bigrams

What are the **bigrams** for the following sentence?

```
the quick brown fox jumped over the lazy dog
```

Bigrams

What are the **bigrams** for the following sentence?

```
the quick brown fox jumped over the lazy dog
```

Bigrams

What are the **bigrams** for the following sentence?

```
the quick brown fox jumped over the lazy dog
```

Bigrams

What are the **bigrams** for the following sentence?

```
the quick brown fox jumped over the lazy dog
```

Bigrams

What are the **bigrams** for the following sentence?

```
the quick brown fox jumped over the lazy dog
```

Trigrams

What are the **trigrams** for the following sentence?

```
the quick brown fox jumped over the lazy dog
```

Trigrams

What are the **trigrams** for the following sentence?

`the quick brown fox jumped over the lazy dog`

Trigrams

What are the **trigrams** for the following sentence?

`the quick brown fox jumped over the lazy dog`

Trigrams

What are the **trigrams** for the following sentence?

`the quick brown fox jumped over the lazy dog`

Trigrams

What are the **trigrams** for the following sentence?

```
the quick brown fox jumped over the lazy dog
```

Trigrams

What are the **trigrams** for the following sentence?

```
the quick brown fox jumped over the lazy dog
```

Trigrams

What are the **trigrams** for the following sentence?

`the quick brown fox jumped over the lazy dog`

And so on for $n > 3$. Food for thought: how many n-grams does a sentence with m words have?

Trigrams

What are the **trigrams** for the following sentence?

`the quick brown fox jumped over the lazy dog`

And so on for $n > 3$. Food for thought: how many n-grams does a sentence with m words have? $(m - n) + 1$!

N-grams cooked three ways

Our task

We'd like to be able to turn a tokenized sentence (a list of words) into a list of its *n*-word sequences. How should we do this?

A recursive approach

The strategy suggested by our slides animation is straightforward to implement with recursion. Given a list of words `s`:

- `take n` words from `s`
- If the resulting `gram` is long enough, keep (`:`) it
- Move the window one word forward with `tail`
- Rinse and repeat (recursion!) until your list gets too short:

```
ngramsRec :: Int -> [a] -> [[a]]
ngramsRec n s =
  let gram = take n s in
    if length gram == n
      then gram : ngramsRec n (tail s)
      else []
```

Another approach using `tails` from `Data.List`

`Data.List` has many useful functions: `group`, `sort`, and now, **`tails`**:

```
import Data.List
```

```
*W4> let ts = tails [1..4]
*W4> ts
[[1,2,3,4],[2,3,4],[3,4],[4],[]]
--↑ ↑        ↑ ↑      ↑ ↑
```

There's a strategy for bigrams: `take 2` from each tail with `2` to take!

Using `tails` (cont)

Starting by using `map` to `take 2` from each list:

```
*W4> let shrunk = map (take 2) ts
*W4> shrunk
[[1,2],[2,3],[3,4],[4],[]]
```

Keeping just the ones with `2` elements, using `filter`:

```
*W4> filter (\xs -> length xs == 2) shrunk
[[1,2],[2,3],[3,4]]
```

That was easy! And what's more, nothing about how this works is specific to bigrams: just replace `2` above with the desired `n`!

Putting it together

Starting with a list like `[1,2,3,4]`:

Putting it together

Starting with a list like `[1,2,3,4]`:

1. Get the `tails`: `[[1,2,3,4],[2,3,4],[3,4],[4],[]]`

Putting it together

Starting with a list like `[1,2,3,4]`:

1. Get the `tails`: `[[1,2,3,4],[2,3,4],[3,4],[4],[]]`
2. `take 2` from each, with `map`: `[[1,2],[2,3],[3,4],[4],[]]`

Putting it together

Starting with a list like `[1,2,3,4]`:

1. Get the `tails`: `[[1,2,3,4],[2,3,4],[3,4],[4],[]]`
2. `take 2` from each, with `map`: `[[1,2],[2,3],[3,4],[4],[]]`
3. Get rid of the ones that're too short: `[[1,2],[2,3],[3,4]]`

Putting it together

Starting with a list like `[1,2,3,4]`:

1. Get the `tails`: `[[1,2,3,4],[2,3,4],[3,4],[4],[]]`
2. `take 2` from each, with `map`: `[[1,2],[2,3],[3,4],[4],[]]`
3. Get rid of the ones that're too short: `[[1,2],[2,3],[3,4]]`

And there are your bigrams! Again, there's nothing special about `take`-ing **`2`** here. So this strategy will work for any *n*.

Translating to code

```haskell
ngrams :: Int -> [a] -> [[a]]
ngrams n l = let ts     = tails l
                 shrunk = map (take n) ts in
  filter (\xs -> length xs == n) shrunk
--       aka `(== n) . length`
```

```
*W4> ngrams 3 (words "I rode to Ralph's to buy bread")
[["I","rode","to"],["rode","to","Ralph's"],
["to","Ralph's","to"],["Ralph's","to","buy"],
["to","buy","bread"]]
```

Note that instead of using `filter`, we could `take ((length l - n) + 1) shrunk`!

Another neat library function: `zip`

```
*W4> :t zip
zip :: [a] -> [b] -> [(a, b)]
```

```
*W4> zip [1..5] ['a'..'d']
[(1,'a'),(2,'b'),(3,'c'),(4,'d')] -- 5 gets tossed out
```

```
*W4> zip [1..5] (tail [1..5])
[(1,2),(2,3),(3,4),(4,5)] -- voila!
```

`zip`-grams

```haskell
bigrams :: [a] -> [(a, a)]      -- pairs of two words
bigrams xs = zip xs (tail xs)   -- rather than lists
```

```haskell
trigrams :: [a] -> [(a, a, a)] -- triples
trigrams xs = zip3 xs (tail xs) (tail (tail xs))
-- Data.List offers up to zip7!
```

So there you have it, three ways to cook *n*-grams:

- Recursively, using `tails`, and using `zip`
- You’re free to use whichever you want
- All are pretty efficient (though the `zip` way seems zippiest)

map trigrams then concat

Our text is split into tokenized, cleaned sentences (tokened). Thus, whichever *n*-gram approach we take, we need to use map:

```
*W4> head (map trigrams tokened) -- 3grams for sent #1
[("the","fulton","county"),("fulton","county","grand"),
("county","grand","jury"),("grand","jury","said"),...]
```

We get one of these lists for each sentence. If we just want a single list with all trigrams, we can concat all these lists:

```
*W4> concat [[(1,2,3), (2,3,4)], [(6,7,8), (7,8,9)]]
[(1,2,3),(2,3,4),(6,7,8),(7,8,9)]
```

concatMap (cont)

map-ing and then concat-ing is so common in Haskell that there is a dedicated library function that does both at once:

```
*W4> concatMap trigrams [[1..4],[6..9]]
[(1,2,3),(2,3,4),(6,7,8),(7,8,9)]
```

Food for thought: sentence breaks

Since we `concat` was there any point in segmenting into sentences?

Food for thought: sentence breaks

Since we `concat` was there any point in segmenting into sentences?

- **Yes!** This way, *n*-grams never cross sentence boundaries.
- There is a real sense in which `"the party"` is a bigram in a way `"party it"` **isn't** in `"I went to the party. It was fun."`

[2] These also serve a more technical purpose. See Jurafsky & Martin's Exercise 3.5.

Food for thought: sentence breaks

Since we `concat` was there any point in segmenting into sentences?

- **Yes!** This way, *n*-grams never cross sentence boundaries.
- There is a real sense in which `"the party"` is a bigram in a way `"party it"` **isn't** in `"I went to the party. It was fun."`

Still, we may not want to toss out sentence breaks when we get our *n*-grams. To accomplish this, we can add start and stop symbols:[2]

```
tokenize' xs = "<s>" : tokenize xs ++ ["</s>"]
```

```
*W4> tokenize' "I am cool"
["<s>","I","am","cool","</s>"]
*W4> bigrams (tokenize' "I am cool")
[("<s>","I"),("I","am"),("am","cool"),("cool","</s>")]
```

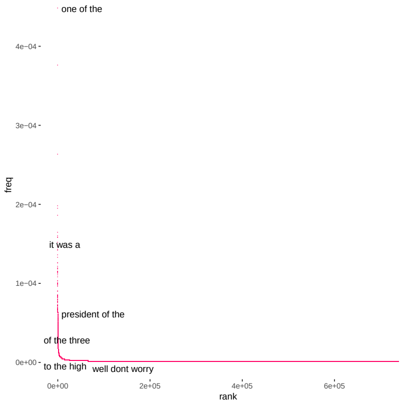
[2] These also serve a more technical purpose. See Jurafsky & Martin's Exercise 3.5.

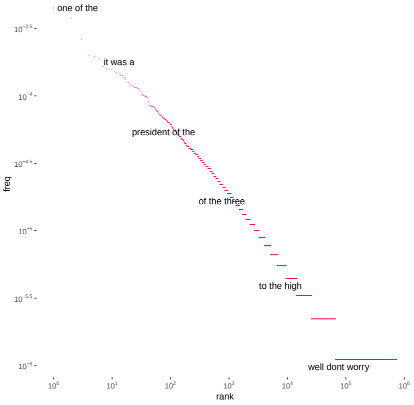
n-gram frequency

We can sort our *n*-grams into types, and count how many instances of each type are in our text, exactly like we did with words!

```haskell
main = do
  raw <- readFile "files/brown_notags.txt"
  let sents = getSentences raw
  let tokened = map (tokenize' . clean') sents
  let grams = concatMap bigrams tokened
  let types = getTypes grams
  let withCount = sort (map addCount types)
  let withFs = relFreqs (length tokened) withCount
  writeFile "out/bigrams.txt" (unlines (map show withFs))
```

Zipfian trigram frequencies in the Brown corpus





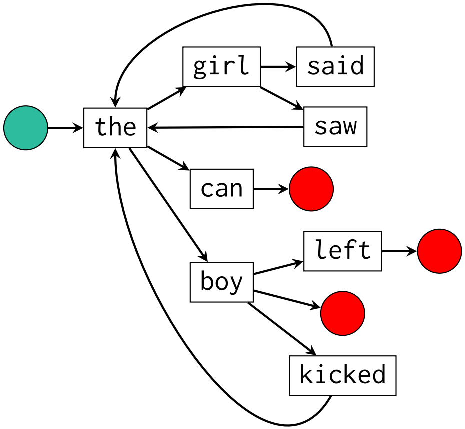
Generative *n*-gram models

A very simple “corpus” and its bigrams

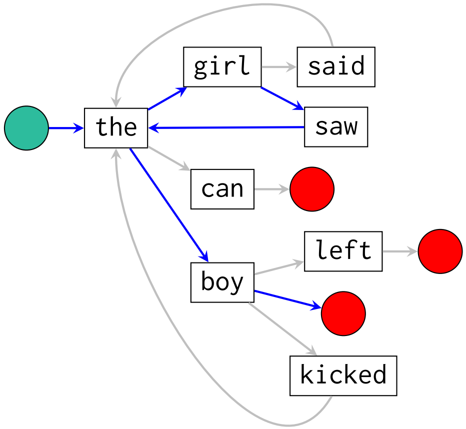
```
corpus = [ "the girl saw the boy"
         , "the boy kicked the can"
         , "the girl said the boy left"]
```

```
*W4> concatMap (bigrams . tokenize') corpus
[("<s>","the"),("the","girl"),("girl","saw"),
("saw","the"), ("the","boy"),("boy","</s>"),
--
("<s>","the"),("the","boy"),("boy","kicked"),
("kicked","the"),("the","can"),("can","</s>"),
--
("<s>","the"),("the","girl"),("girl","said"),
("said","the"),("the","boy"),("boy","left"),
("left","</s>")]
```

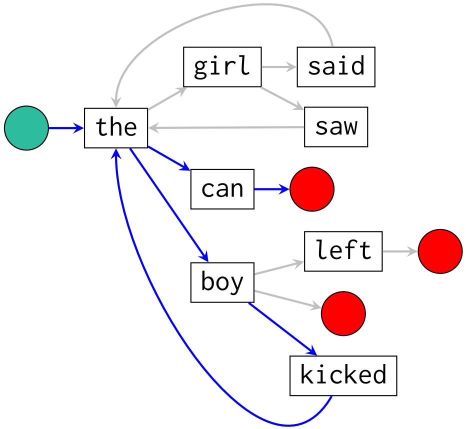
Visualizing bigrams as a diagram (directed graph)



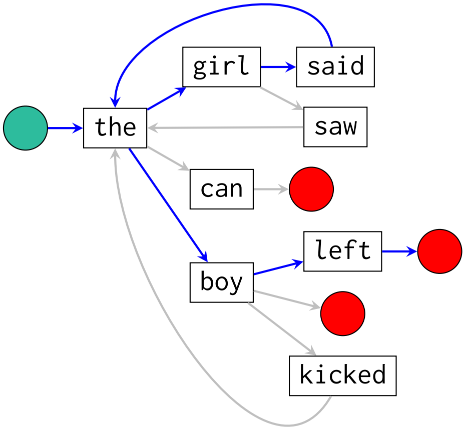
Visualizing bigrams as a diagram (directed graph)



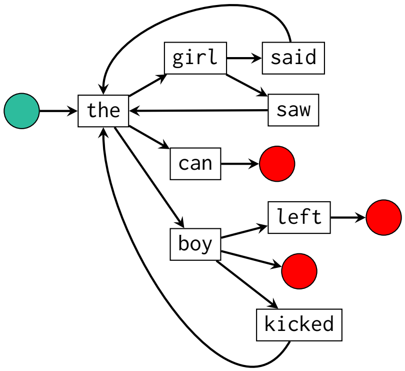
Visualizing bigrams as a diagram (directed graph)



Visualizing bigrams as a diagram (directed graph)



Generativity and creativity: walking new paths



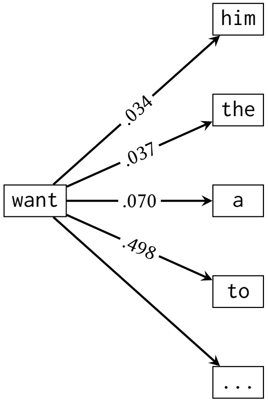
The good

```
<s> the boy left </s>
<s> the boy kicked the boy </s>
<s> the girl saw the can </s>
<s> the girl said the girl said ...
```

The not so good

```
<s> the boy </s>
<s> the girl said the can </s>
<s> the boy kicked the boy left </s>
```

Transitions may be labeled with **likelihoods** or **weights**



Small portion of weighted bigram model, Brown corpus

Probabilistic *n*-gram generation

Generate text by *probabilistically* walking through the graph: at any fork in the road, pick a path by (in effect) rolling a weighted dice.[3]

```
*W4> Control.Monad.Random.fromList [("H",0.1), ("T",0.9)]
"T" -- probably :)
```

The probability of a complete path through the graph is the product of all the transition probabilities along its path:

- $P(X_1 \dots X_n) = \prod_{k=1}^{n} P(X_k \mid X_{k-1})$ (bigrams)
- $P(X_1 \dots X_n) = \prod_{k=1}^{n} P(X_k \mid X_{k-2} X_{k-1})$ (trigrams)

[3] If you don't use weights, the paths at a fork may be taken with equal likelihood.

Demo… drawbacks?

Demo… drawbacks?

Some obvious issues with these models:

- Unigram models are pretty worthless for doing anything other than generating word salad with relatively common words
- Bigram models are goldfish-brained; any single transition is plausible enough, but generally goes weird places fast
- Trigram models do better (though not amazingly so). But they're easily cornered, getting stuck in a sparse region of the graph, where they can only reproduce training text verbatim.
    - Zipf's law: most of your types (words or *n*-grams) are **rare**

In general, higher-order gram models work better, but they require **incredible** amounts of training data. Otherwise most grammatical *n*-grams simply will simply never be observed.

The case for structure

Probably the biggest issue from a linguist's perspective is that *n*-gram models make no provisions for structure:

(4) I met the chef **who** Mary said you wanted to meet _.

By the time (e.g.) a trigram model generates *you*, it's forgotten there was a *who* to begin with, leading to things like:

(5) I met the chef **who** Mary said you have my sleeping bag.

Long-distance dependencies are tricky for *n*-grams, in general.

The case for structure (cont)

Or consider subject-verb agreement. What matters for *was* vs. *were* in the following example is a choice made 9 words back!

(6) [The woman/women who noticed Taylor sitting forlorn alone on the dock] was/were at first quite puzzled.

The way English actually works, of course, is that the verb agrees with its **subject** (in square brackets).

- Subjecthood is structural, but *n*-grams don't see structure.