ITCS 6101/8101: Natural Language Processing

Tokenization: Words, Morphemes, Subwords

Razvan C. Bunescu

Department of Computer Science @ CCI

razvan.bunescu@charlotte.edu

Fundamental NLP Tasks Identify **Linguistic Structures**

- **Tokenization**
- **Morphological Analysis**
- Part of Speech Tagging
- Syntactic Parsing
- Word Meaning
- Semantic Parsing
- Anaphora/Coreference Resolution
- Named Entity Linking

Tokenization

- **Tokenization** = segmenting text into words and sentences.
  - A crucial first step in most text processing applications.
  - Language Models use ***subword*** tokenization.

- Whitespace indicative of word boundaries?
  - Yes: English, French, Spanish, …
  - No: Chinese, Japanese, Thai, …

- Whitespace is not enough:
  - ‘What’re you? Crazy?’ said Sadowsky. ‘I can’t afford to do that.’

Whitspace ⇒ ‘*what’re_you?_crazy?*_said_*Sadowsky._‘I_can’t_afford_to_do_that.*

Correct ⇒ ‘*_what_’re_you_?_crazy_?_Sadowsky_._‘_I_can_’t_ afford_to_do_that_.*

Word Segmentation

- In English, characters other than whitespace can be used to separate words:
  - , ; . - : ( )”
- But punctuation often occurs inside words:
  - m.p.h., Ph.D., AT&T, 01/02/06, google.com, 62.5
  - Homework: design regular expressions to match constructions where punctuation does not split:
    - *acronyms*, *dates*, *web addresses*, *numbers*, etc.
    - https://docs.python.org/3/howto/regex.html
- Expansion of clitic constructions:
  - he’s happy ⇒ he is happy
  - Need ambiguity resolution between clitic construction, possessive markers, quotative markers:
    - he’s happy vs. the book’s cover vs. ‘what are you? crazy?’

Tokens, Types, Corpus, Vocabulary

- **Tokenization** = segmenting text into tokens:
  - **token** = a sequence of characters, in a particular document at a particular position.
  - **type** = the class of all tokens that contain the same character sequence.
    - "... to **be** or not to **be** ..."
    - "... so **be** it, he said ..."
  - A *token* is an instance, or occurrence, of a *type*.

- A **corpus** is a collection of text documents:
  1. D_1 = "*Colorless green ideas sleep furiously .*"
  2. D_2 = "*Green makes me sleep , red makes me alert .*"

- The corpus **vocabulary** maps token *types* to their frequency:
  - {*colorless*: 1, *green*: 2, *ideas*: 1, *sleep*: 2, *furiously*: 1, *makes*: 2, …}

Tokenization: Tokens, Types, and Terms

- **Tokenization** = segmenting text into tokens:
  - **token** = a sequence of characters, in a particular document at a particular position.
  - **type** = the class of all tokens that contain the same character sequence.

	Types = |V|	Tokens = N
Shakespeare	31 thousand	884,000
Brown Corpus	38 thousand	1 million
Switchboard conversations	20 thousand	2.4 million
COCA	2 million	440 million
Google N-grams	13+ million	1 trillion

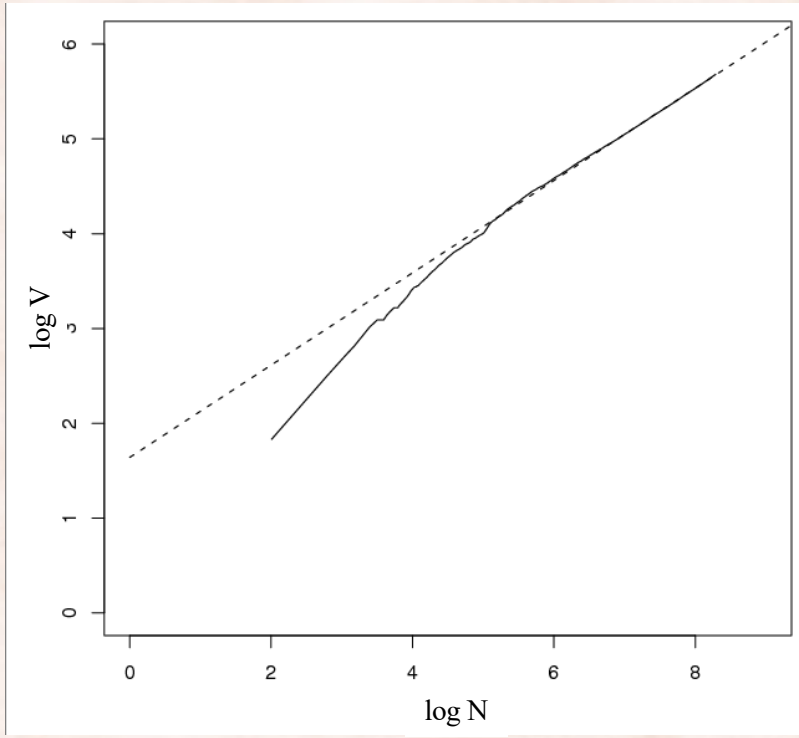
The bigger the corpus, the more word types!

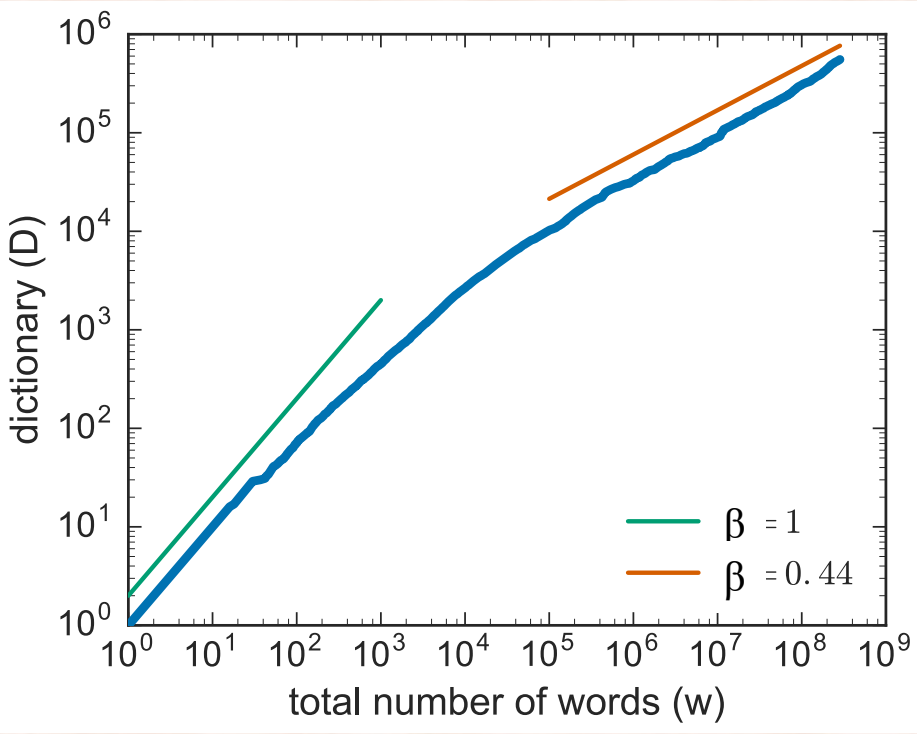
Heap's Law

- Given:
  - |V| is the size of the vocabulary.
  - N is the number of tokens in the collection.
- Then:
  - $|V| = kN^{\beta}$
  - k, β depend on the collection type:
    - typical values: $30 \leq k \leq 100$ and $\beta \approx 0.5$ (square root).
    - in a log-log plot of |V| vs. N, Heaps' law predicts a line with slope of about ½.

Heap's Law Fit to Reuters RCV1 and Guttenberg

- For RCV1, the dashed line $\log_{10} V = 0.49 \log_{10} N + 1.64$ is the best least squares fit.
  - Thus, $V = 10^{1.64} N^{0.49}$ which means $k = 10^{1.64} \approx 44$ and $b = 0.49$.
  - For first 1,000,020 tokens, Heap's Law predicts 38,323 terms vs. actual 38,365 terms.





Zipf's, Heaps' and Taylor's laws are determined by the expansion into the adjacent possible

There are too many words!

$$|V| = kN^{\beta}$$ ← Roughly 0.5

- Function words: *of, the, is, and, una, 是, …*
- Content words: *mango, braise, snowy, feliz, 北京, …*
- Proper names: *Yahoo, Google, Bing, …*

- **Problem**: no matter how big our vocabulary, there will always be words we missed.
  - We will always have unknown words.

- **Solution**:
  - Morphological analysis => use morphemes as tokens instead of words.
    - But morphemes cab be nontrivial to define and identify…
  - **Subword tokenization** => the standard in modern NLP, e.g. LLMs.

Morphology

- **Morphology** = the field of linguistics that studies the internal structure of words.
- **Morpheme** is the smallest linguistic unit that has semantic meaning:
  - **Roots** or **stems**: central morpheme of the word, supplying the main meaning
    - "*carry*", "*depend*", "*Google*", "*lock*"
  - **Affixes**: add additional meaning
    - "*pre*", "*ed*", "*ly*", "*s*"
    - *Inflectional*: grammatical morphemes, often syntactic role like agreement
      - *–ed* past tense on verbs, e.g., *work-ed*
      - *– s*/ – es plural on nouns, e.g., *glass-es*
    - *Derivational*: idiosyncratic in application and meaning, often change grammatical class.
      - *care* (noun)
        - *– full* => *careful* (adjective)
        - *– ly* => *carefully* (adverb)

Morphology

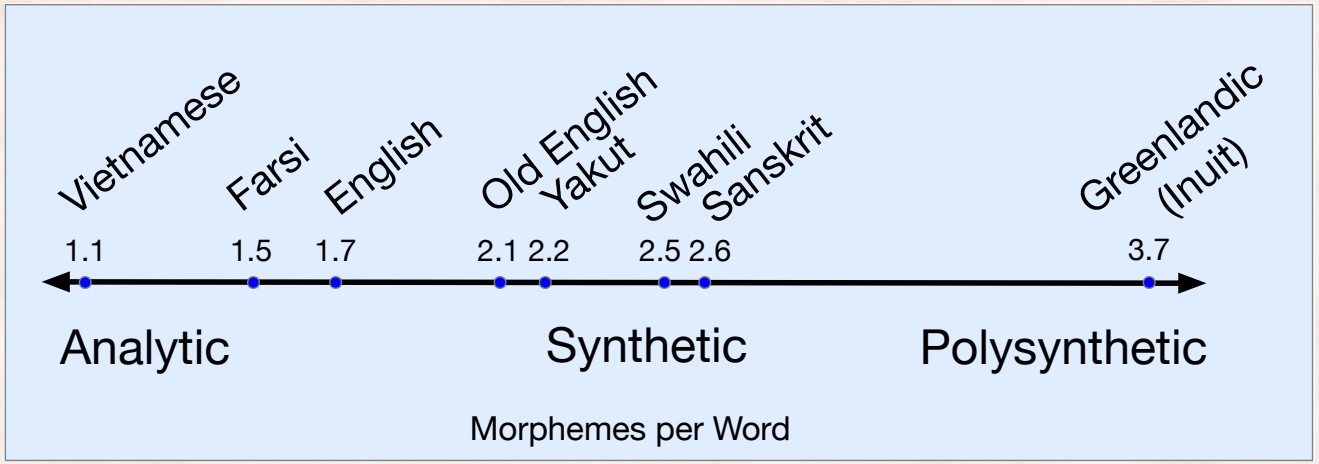
- **Morpheme** is the smallest linguistic unit that has semantic meaning:
  - **Clitics** are morphemes that act syntactically like a word, but are:
    - reduced in form.
    - attached to another word.
  - Clitics in English: `'ve` in `I've` (`'ve` can't appear alone), or `'s` in `the teacher's book`
  - Clitics in French: `l'` in `l'opera`

- Using morphemes as tokens can mitigate the unknown word problem:
  - **Train**: seen words like *worked*, *play*, …
    - Tokenize into morphemes *work*, *-ed*, *play*, …
  - **Test**: encounter unseen word *played*.
    - Tokenize as *play*, *-ed*.
  - But segmenting words into morphemes can be difficult.

Morphological Typology: Morphemes per Word

- ***Few.*** *Cantonese,* spoken in Guangdong, Guangxi, Hong Kong*:*
  - *keoi5 waa6 cyun4 gwok3 zeoi3 daai6 gaan1 uk1 hai6 ni1 gaan1*
  - he say entire country most big building house is this building
    - *He said the biggest house in the country was this one"*

- **Many.** *Koryak*, spoken in Kamchatka peninsula in Russia:
  - *t-ə-nk'e-mejŋ-ə-jetemə-nni-k*
  - 1SG.S-E-midnight-big-E-yurt.cover-E-sew-1SG.S[PFV]
    - *"I sewed a lot of yurt covers in the middle of a night."*



Joseph Greenberg scale (1960)

Morphological Analysis

- **Morphological analysis** = segmenting words into morphemes:
  - carried ⇒ carry + ed (past tense)
  - independently ⇒ in + (depend + ent) + ly
  - Googlers ⇒ (Google + er) + s (plural)
  - unlockable ⇒ un + (lock + able) ? (un + lock) + able ?
- Difficulty of segmenting into morphemes varies across languages:
  - **Agglutinative** languages like Turkish have very clean boundaries between morphemes.
  - **Fusion** languages: a single affix may conflate multiple morphemes.
    - Russian `-om` in `stolom` (table-SG-INSTR- DECL1)
      - » instrumental, singular, and first declension.
    - English `-s` in "`She reads the article`"
      - » Means both "third person" and "present tense"
  - These are tendencies rather than absolutes.

Tokenization Approaches

- **Rule based** (*word* or *morpheme* tokenization):
  - Develop rules based on linguistic knowledge for breaking strings into tokens corresponding to words and punctuation symbols.
    - Usually implemented with a combination of lexicons, regular expressions, and code.
    - Drawback: what to do with new or misspelled words?
  - Examples: **spaCy** and **NLTK** tokenizers.

- **Statistical** (*subword* tokenization):
  - Use statistics over a large corpus of text to learn to break text into ”*common*” **subword** tokens.
    - Advantage: can accommodate unseen words.
  - Examples: **BPE**, **WordPiece**, **SentencePiece**.

Subword Tokenization

- NLP algorithms often learn some facts about language from a **training** corpus and then use these facts to make decisions about a separate **test** corpus.
  - The vocabulary of tokens V is built from the **training** corpus.
  - What to do if the test **corpus** contains a token that is not in V?
    - Training corpus contains low, new, newer, but not lower.
    - If the word lower appears in the test corpus, the NLP system will *not know what to do with it*.
      - But we've seen new and newer! If we had segmented newer as new + er, the NLP system could have learned that any <adj> + er means a stronger version of <adj>.
      - This is how we can make (some) sense of Jabberwocky.
        - » https://en.wikipedia.org/wiki/Jabberwocky

Word segmentation: Subwords

- Use the data to tell us how to tokenize:
  - Instead of manually designed rules.
  - Instead of training on manually tokenized examples.

- Called **Subword tokenization**:
  - Because tokens are often parts of words.
  - Tokens end up including common morphemes, like *-est* or *-er*.
    - A morpheme is the smallest meaning-bearing unit of a language;
      - *unlikeliest* has morphemes *un-*, *likely*, and *-est*.

Subword Tokenization

- Three common algorithms:
  1. **Byte-Pair Encoding (BPE)** (Sennrich et al., 2016)
  2. **Unigram language modeling tokenization** (Kudo, 2018)
  3. **WordPiece** (Schuster and Nakajima, 2012)

- All have 2 parts:
  1. A token **learner** that takes a raw training corpus and induces a vocabulary, e.g. a set of tokens.
  2. A token **segmenter** that takes a raw test sentence and tokenizes it according to that vocabulary.

Byte Pair Encoding (BPE)

Let vocabulary be the set of all individual characters

= {A, B, C, D, … , a, b, c, d, …}

- Repeat:
  - Choose the two symbols that are most frequently adjacent in training corpus (say ‘A’, ‘B’),
  - Add a new merged symbol ‘AB’ to the vocabulary
  - Replace every adjacent ’A’ ’B’ in corpus with ‘AB’.
- Until *k* merges have been done.

BPE token learner algorithm

function BYTE-PAIR ENCODING(strings C, number of merges k) **returns** vocab V

$V \leftarrow$ all unique characters in C # initial set of tokens is characters
for $i = 1$ **to** k **do** # merge tokens til k times
  $t_L, t_R \leftarrow$ Most frequent pair of adjacent tokens in C
  $t_{NEW} \leftarrow t_L + t_R$ # make new token by concatenating
  $V \leftarrow V + t_{NEW}$ # update the vocabulary
  Replace each occurrence of t_L, t_R in C with t_{NEW} # and update the corpus
return V

Byte Pair Encoding (BPE)

- Most subword algorithms are run inside white-space separated tokens.
- First add a special end-of-word symbol '__' after each word in the corpus:
  - Alternatively, attach symbol '__' at the beginning of each word
- Next, separate into characters, create an initial vocabulary from all characters.

BPE token learner

Original (very fascinating🙄) corpus:

low low low low low lowest lowest newer newer newer newer newer newer wider wider wider new new

Add end-of-word tokens and segment:

corpus	**vocabulary**
5 l o w _	_, d, e, i, l, n, o, r, s, t, w
2 l o w e s t _	
6 n e w e r _	
3 w i d e r _	
2 n e w _	

BPE token learner

corpus

5 l o w _
2 l o w e s t _
6 n e w e r _
3 w i d e r _
2 n e w _

vocabulary

_, d, e, i, l, n, o, r, s, t, w

R_1 = merge e r to er

corpus

5 l o w _
2 l o w e s t _
6 n e w er _
3 w i d er _
2 n e w _

vocabulary

_, d, e, i, l, n, o, r, s, t, w, er

Byte Pair Encoding (BPE)

corpus	vocabulary
5 l o w _	_, d, e, i, l, n, o, r, s, t, w, er
2 l o w e s t _	
6 n e w er _	
3 w i d er _	
2 n e w _	

R_2 = merge er _ to er_

corpus	vocabulary
5 l o w _	_, d, e, i, l, n, o, r, s, t, w, er, er_
2 l o w e s t _	
6 n e w er_	
3 w i d er_	
2 n e w _	

Byte Pair Encoding (BPE)

corpus	vocabulary
5 l o w _	_, d, e, i, l, n, o, r, s, t, w, er, er_
2 l o w e s t _	
6 n e w er_	
3 w i d er_	
2 n e w _	

R_3 = merge n e to ne

corpus	vocabulary
5 l o w _	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne
2 l o w e s t _	
6 ne w er_	
3 w i d er_	
2 ne w _	

Byte Pair Encoding (BPE)

So far:

R_1 = merge e r to er

R_2 = merge er _ to er_

R_3 = merge n e to ne

The next merges are:

	Merge	Current Vocabulary
R_4 =	(ne, w)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new
R_5 =	(l, o)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo
R_6 =	(lo, w)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo, low
R_7 =	(new, er_)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo, low, newer_
R_8 =	(low, _)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo, low, newer_, low_

Using BPE on a new text

- On the test corpus, run each merge learned from the training data:
  - Greedily, **in the order they were added** to vocabulary.
    - test frequencies don't play a role.
      - So, merge every e r to er, then merge er _ to er_, etc.

- V = {_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo, low, newer_, low_}

  - Test set "n e w e r _" would be tokenized as a full word.
  - Test set "l o w e r _" would be two tokens: "low" + "er_ ":
    - “lower” was never seen in the training corpus.
    - However, we’ve seen “low” and “er”.
      - The *meaning* of “low” + er” can be derived from the *meaning* of its components.

OpenAI's tokenizer

https://github.com/openai/tiktoken

- **tiktoken** is a fast open source BPE tokenizer released by OpenAI's and used with its models.

```
import tiktoken

# To get the tokenizer corresponding to a specific model in the OpenAI API:
enc = tiktoken.encoding_for_model("gpt-4")

tokens = [enc.decode_single_token_bytes(token) for token in
                    enc.encode(" soooo much rrrracing in Kannapolis this Summer!")]
# To translate to the standard representation (utf-8), you can use token.decode('utf-8').
utf8_tokens = [token.decode('utf-8') for token in tokens]
print(utf8_tokens)
```

```
['so', 'ooo', ' much', ' r', 'rr', 'r', 'acing', ' in', ' Kann', 'apolis', ' this', ' Summer', '!']
```

- More details in the jupyter-notebook …

WordPiece Tokenizer

- Used by BERT, DistilBERT, and Electra.
- Greedy procedure like BPE.
  - BPE chooses to merge the **most frequent** symbol pair.
  - WordPiece merges the pair that **maximizes the likelihood** of the training data once added to the vocabulary.
    - If A and B are a candidate pair, their score is given by:

$$\frac{P(AB)}{P(A)P(B)}$$

how is this related to pmi(A,B)?

    - Choose to merge the pair with the highest score.
    - This can be shown to maximize the likelihood of the data.

https://huggingface.co/docs/transformers/tokenizer_summary#wordpiece
https://ai.googleblog.com/2021/12/a-fast-wordpiece-tokenization-system.html
https://www.tensorflow.org/text/guide/subwords_tokenizer#applying_wordpiece

Statistical Properties of Text

- **Heap's Law** models the number of words in the vocabulary as a function of the corpus size:
  - What is the number of unique words appearing in a corpus of size N words?
  - This determines how the size of the inverted index in IR will scale with the size of the corpus.

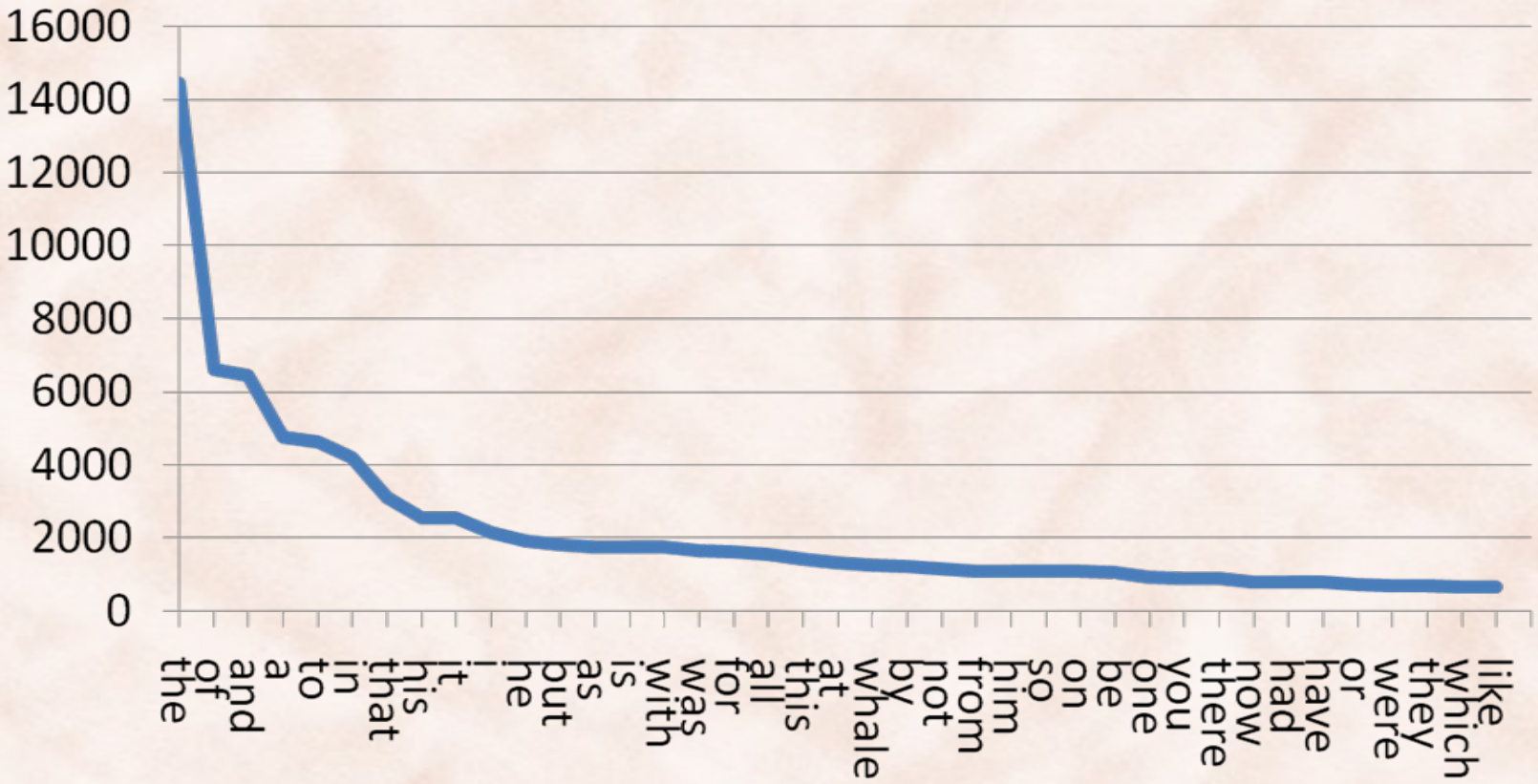
- **Zipf's Law** models the distribution of terms/types in a corpus:
  - How many times does the k^{th} most frequent word appears in a corpus of size N words?
  - Important for determining index terms for IR (search engines) and properties of compression algorithms.

Word Distribution

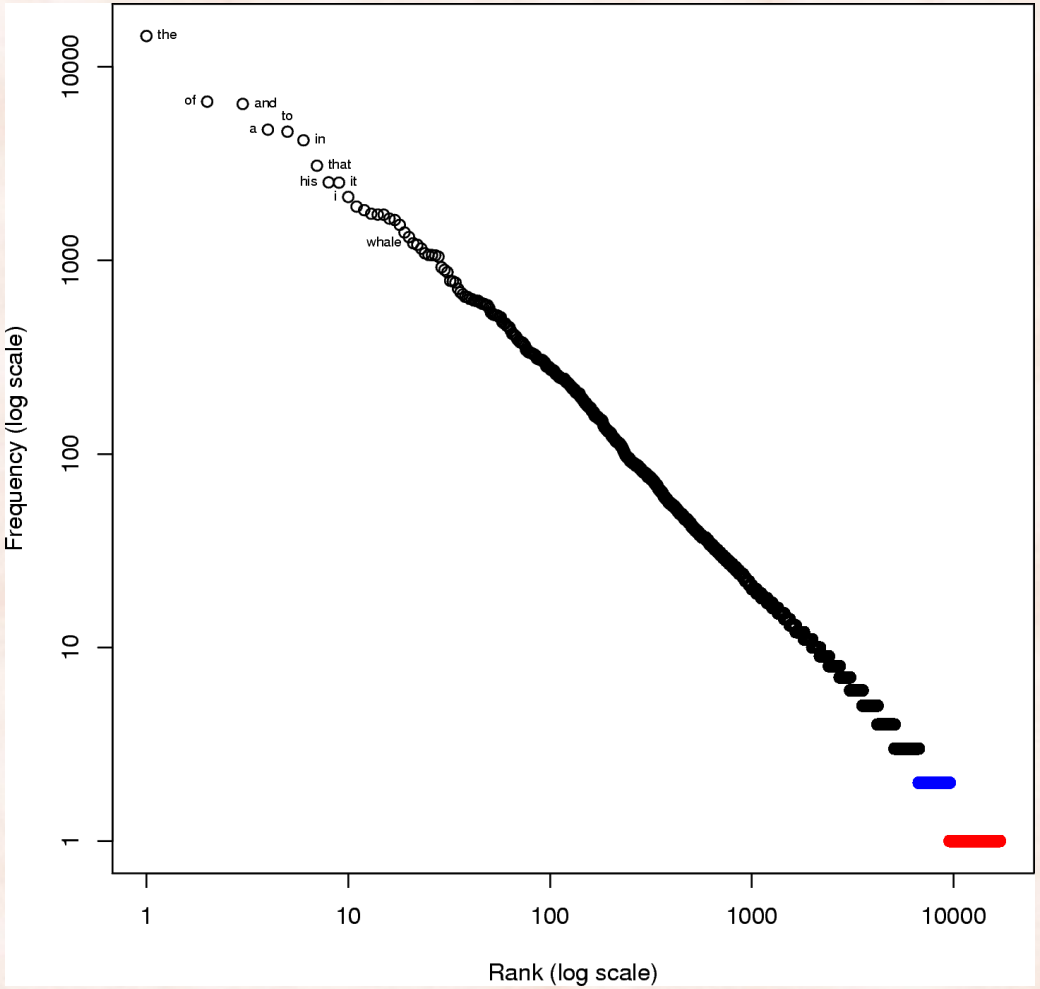
- **A few words are very common**:
  - The 2 most frequent words (e.g. “the”, “of”) can account for about 10% of word occurrences.
- **Most words are very rare**:
  - Half the words in a corpus appear only once, called *hapax legomena* (Greek for “read only once”)
- A “*heavy tailed*” or “*long tailed*” distribution:
  - Since more of the probability mass is in the “tail” compared to an exponential distribution.

Word Distribution

Frequency vs. rank for all words in Moby Dick.



Word Distribution (Log Scale)



Moby Dick:
- 44% *hapax legomena*
- 17% *dis legomena*

"Honorificabilitudinitatibus":
- Shakespeare's *hapax legomenon*
- longest word with alternating vowels and consonants

Zipf's Law

- Rank all the words in the vocabulary by their frequency, in decreasing order.
  - Let $r(w)$ be the rank of word w.
  - Let $f(w)$ be the frequency of word w.

- Zipf (1949) postulated that frequency and rank are related by a *power law*:

  - c is a normalization constant that depends on the corpus.

$$f(w) = \frac{c}{r(w)}$$

Zipf's Law

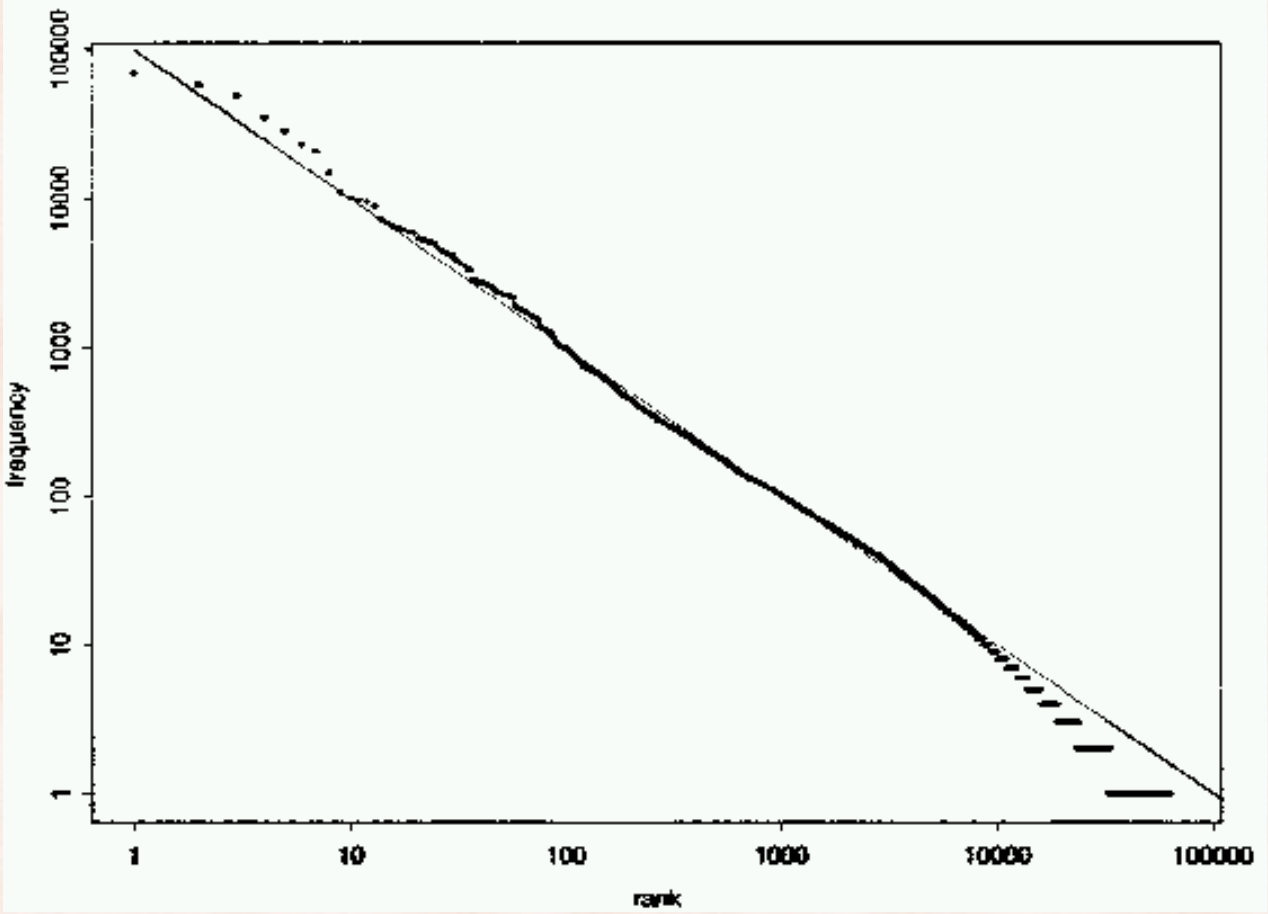
- If the most frequent term (the) occurs f_1 times:
  - Then the second most frequent term (of) occurs $f_1 / 2$ times.
  - The third most frequent term (and) occurs $f_1 / 3$ times, …

- **Power Laws**: $y = cx^k$
  - Zipf's Law is a power law with $k = -1$.
  - Linear relationship between log(y) and log(x):
    - $\log(y) = \log c + k \log(x)$
    - on a log scale, power laws give a straight line with slope k.

- Zipf is quite accurate, except for very high and low rank.

Zipf's Law Fit to Brown Corpus

$$f(w) = \frac{100000}{r(w)}$$



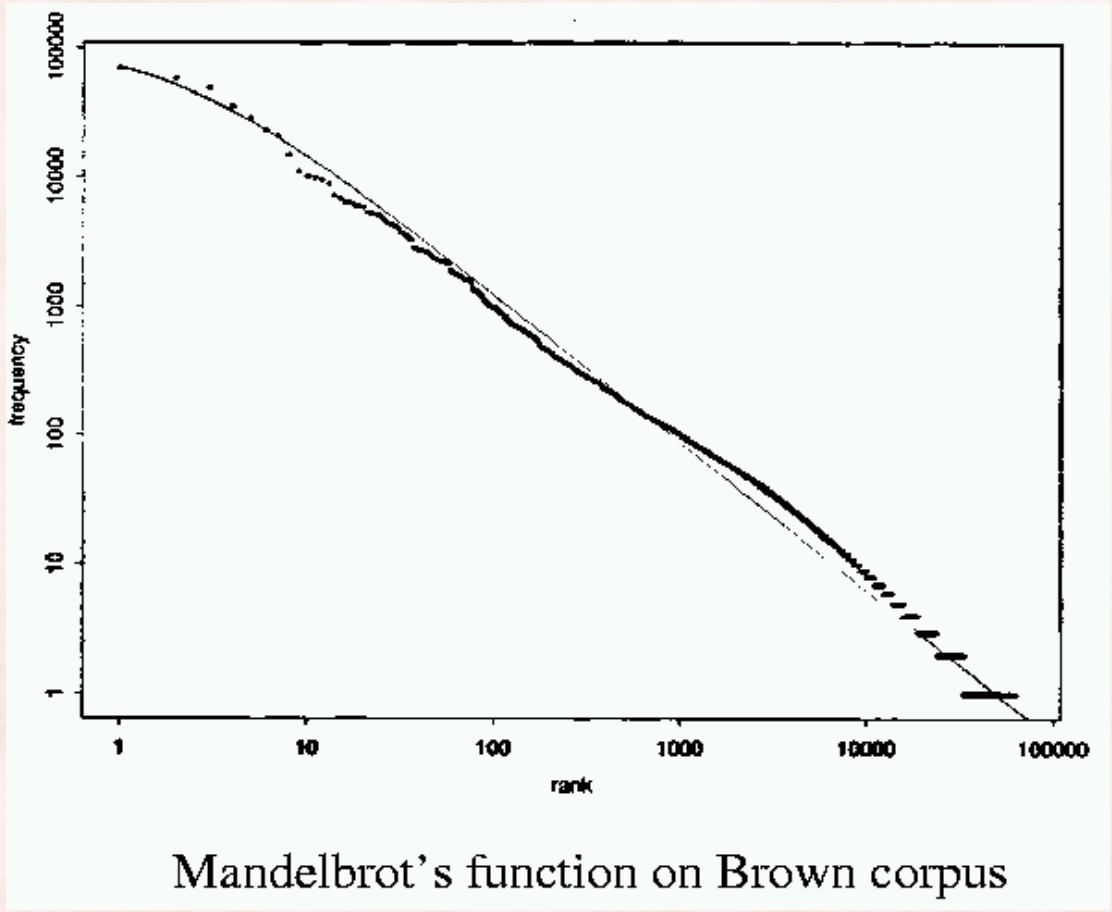
Mandelbrot's Distribution

- The following more general form gives a bit better fit:

$$f = c/(r+\rho)^K$$

- When fit to Brown corpus:
  - $c = 105.4$
  - $K = 1.15$
  - $\rho = 100$

Mandelbrot's Law Fit to Brown Corpus



Mandelbrot's function on Brown corpus

Explanations

- **Zipf's Law**:
  - Zipf's explanation was his "principle of least effort":
    - Balance between speaker's desire for a small vocabulary and hearer's desire for a large one.
  - Herbert Simon's explanation is "rich get richer."
  - Li (1992) shows that just random typing of letters including a space will generate "words" with a Zipfian distribution.

- **Heaps' Law**:
  - Can be derived from Zipf's law by assuming documents are generated by randomly sampling words from a Zipfian distribution.

Supplementary Readings

- Section 2.1 to 2.6 in the textbook.
- HuggingFace summary of tokenization techniques.