ITCS 4111/5111: Introduction to NLP

Tokenization: From text to sentences and tokens

Razvan C. Bunescu

Department of Computer Science @ CCI

rbunescu@uncc.edu

Tokenization

- **Tokenization** = segmenting text into words and sentences.
  - A crucial first step in most text processing applications.
  - Recent SoA 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.*

Target ⇒ '*_what_'re_you_?_crazy_?_Sadowsky_._'_I_can_'t_ afford_to_do_that_.*

Tokenization

- John went to San Francisco for an interview at Google. They asked him lots of C++ questions. He’s happy that the interview went well. John’s sister met him at the headquarters and they walked for 2.5 miles to the hotel.
  - For which company did John interview?
  - What topic was he tested on?
  - How is John feeling?
  - Whose sister met him afterwards?
  - How many miles did he walk to the hotel?

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*
  - But: *he’s happy* vs. *the book’s cover* vs. *‘what are you? crazy?’*

Tokenization in IR: From Text to Tokens

- **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 ..."

    3 tokens, 1 type
  - **term** = a (normalized) type that is included in the dictionary.
    - *text* = "to sleep perchance to dream", "US ambassador dreams"
    - *tokens* = to, sleep, perchance, to, dream, US, ambassador, dreams
    - *types* = to, sleep, perchance, dream, US, ambassador, dreams
    - *terms* = sleep, perchance, dream, USA, ambassador (lemmas, norm)

Tokenization in IR: From Text to Tokens

- Split on whitespace and non-alphanumeric?
  - Good as a starting point, but complicated by many tricky cases:
    - Appostrophes are ambiguous:
      - possessive constructions:
        - » the books’s cover => the book s cover
      - contractions:
        - » he’s happy => he is happy
        - » aren’t => are not
      - quotations:
        - » ‘let it be’ => ‘ let it be ‘

Tokenization in IR: From Text to Tokens

- Split on whitespace and non-alphanumeric?
  - Good as a starting point, but complicated by many tricky cases:
    - Whitespaces in proper names or collocations:
      - San Francisco => San_Francisco
        - » how do we determine it should be a single token?
    - Hyphenations:
      - co-education => co-education
      - state-of-the-art => state of the art? state_of_the_art?
      - lowercase, lower-case, lower case => lower_case
      - Hewlett-Packard => Hewlett_Packard? Hewlett Packard?
    - Whitespaces and Hyphenations:
      - San Francisco-Los Angeles => San_Francisco Los_Angeles

Tokenization in IR: From Text to Tokens

- Split on whitespace and non-alphanumeric?
  - Good as a starting point, but complicated by many tricky cases:
    - Whitespaces and Hyphenations:
      - split on hyphens and whitespaces, but use a phrase index.
    - Unusual strings that should be recognized as tokens:
      - C++, C#, B-52, C4.5,M*A*S*H.
    - URLs, IP addresses, email addresses, tracking numbers.
      - exclude numbers, monetary amounts, URLs from indexing?

- **Use same tokenization rules for all documents:**
  - **e.g. training vs. testing.**

Tokenization is Language Dependent

- Need to know the language of document/query:
  - **Language Identification**, based on classifiers trained on short character subsequences as features, is highly effective.
  - French (reduced definite article, postposed clitic pronouns):
    - l'ensemble, un ensemble, donne-moi.
  - German (compund nouns), need *compound splitter*:
    - Computerlinguistik
    - Lebensversicherungsgesellschaftsangestellter
  - East Asian languages, need *word segmenter*:
    - 莎拉波娃现在居住在美国东南部的佛罗里达。
      - Not always guaranteed a unique tokenization
    - Complicated in Japanese, with multiple alphabets intermingled.

Tokenization is Language Dependent

- Need to know the language of document/query:
  - Arabic and Hebrew:
    - Written right to left, but with certain items like numbers written left to right.
    - Words are separated, but letter forms within a word form complex ligatures

استقلت الجزائر في سنة 1962 بعد 132 عاما من الاحتلال الفرنسي.

← → ← → ← start

Algeria achieved its independence in 1962 after 132 years of French occupation.

Language Dependent Processing

- Compound Splitting for German:
  - usually implemented by finding segments that match against dictionary entries.
- Word Segmentation for Chinese:
  - ML sequence tagging models trained on manually segmented text:
    - *Logistic Regression, HMMs, Conditional Random Fields*.
  - Multiple segmentations are possible:

和尚

► **Figure 2.4** Ambiguities in Chinese word segmentation. The two characters can be treated as one word meaning 'monk' or as a sequence of two words meaning 'and' and 'still'.

From Tokens to Terms: Normalization

- **Token Normalization** = reducing multiple tokens to the same canonical term, such that matches occur despite superficial differences.
  1. Create equivalence classes, named after one member of the class:
     - {anti-discriminatory, antidiscriminatory}
     - {U.S.A., USA}
       - but what about C.A.T vs. CAT?
  2. Can complicate later processing tasks if annotation already done on original, unnormalized version of text:
     - Need to maintain positional correspondence between normalized token and its original, unnormalized version

From Tokens to Terms: Normalization

- Accents and diacritics in French:
  - ***résumé*** vs. ***resume.***
- Umlauts in German:
  - ***Tuebingen*** vs. ***Tübingen***
- British vs. American spellings:
  - colour vs. color.
- Multiple formats for dates, times:
  - 09/30/2013 vs. Sep 30, 2013.

From Tokens to Terms: Normalization

- **Case-Folding** = reduce all letters to lower case:
  - change Automobile at beginning of sentences to automobile.
    - how about Ferrari?
  - but may lead to unintended matches:
    - the Fed vs. fed.
    - Bush, Black, General Motors, Associated Press, ...
- **Heuristic** = lowercase only some tokens:
  - words at beginning of sentences.
  - all words in a title where most words are capitalized.
- **Truecasing** = use a classifier to decide when to fold:
  - trained on many heuristic features.

Lemmatization and Stemming

- **Lemmatization** = reduce a word to its base/dictionary form, i.e. its lemma:
  - is, am, are => be
  - car, cars => car

- Lemmatization commonly only collapses the different *inflectional* forms of a lemma:
  - saw => see (if verb), or saw (if noun).

From Tokens to Terms: Stemming

- **Stemming** = reduce *inflectional* and sometimes *derivationally* related forms of a word to a common base form i.e. the *stem*.
  - automate, automates, automatic, automation => automat
  - see, saw => s

- Crude affix chopping that is language dependent:

for example compressed and compression are both accepted as equivalent to compress.	→	for exampl compress and compress ar both accept as equival to compress

Porter's Algorithm

http://www.tartarus.org/˜martin/PorterStemmer/

- The most common stemmer for English:
  - at least as good as other stemming options.
  - 5 phases of word reductions, applied sequentially.
  - conventions for rule selection and application:
    - select the reduction rule that applies to the longest suffix:

Rule			**Example**		
SSES	→	SS	caresses	→	caress
IES	→	I	ponies	→	poni
SS	→	SS	caress	→	caress
S	→		cats	→	cat

    - check the number of syllables, for suffix determination:

$(m > 1)$ EMENT →

would map *replacement* to *replac*, but not *cement* to *c*.

Other Stemming Algorithms

- **Lovins** stemmer, **Paice/Husk** stemmer, **Snowball**:
  - http://www.cs.waikato.ac.nz/~eibe/stemmers/
  - http://www.comp.lancs.ac.uk/computing/research/stemming/
- Stemming is language- and often application-specific:
  - open source and commercial plug-ins.
- Does it improve IR performance?
  - mixed results for English: improves recall, but hurts precision.
    - operative (dentistry) ⇒ oper
  - definitely useful for languages with richer morphology:
    - Spanish, German, Finish (30% gains).

Sentence Segmentation

- Generally based on punctuation marks: **? ! .**
  - Periods are ambiguous, as sentence boundary markers and abbreviation/acronym markers:
    - *Mr., Inc., m.p.h.*
  - Sometimes they mark both:
    - SAN FRANCISCO (MarketWatch) – Technology stocks were mostly in positive territory on Monday, powered by gains in shares of Microsoft Corp. and **IBM Corp.**

- Tokenization approaches:
  - Regular Expressions.
  - Machine Learning (state of the art).

Extracting Linguistic Features with spaCy

```
import spacy

nlp = spacy.load("en_core_web_sm")
doc = nlp("Apple is looking at buying U.K. startup for $11.1 million.")

for token in doc:
    print(token.text, token.lemma_, token.pos_, token.tag_, token.dep_,
          token.shape_, token.is_alpha, token.is_stop)
```

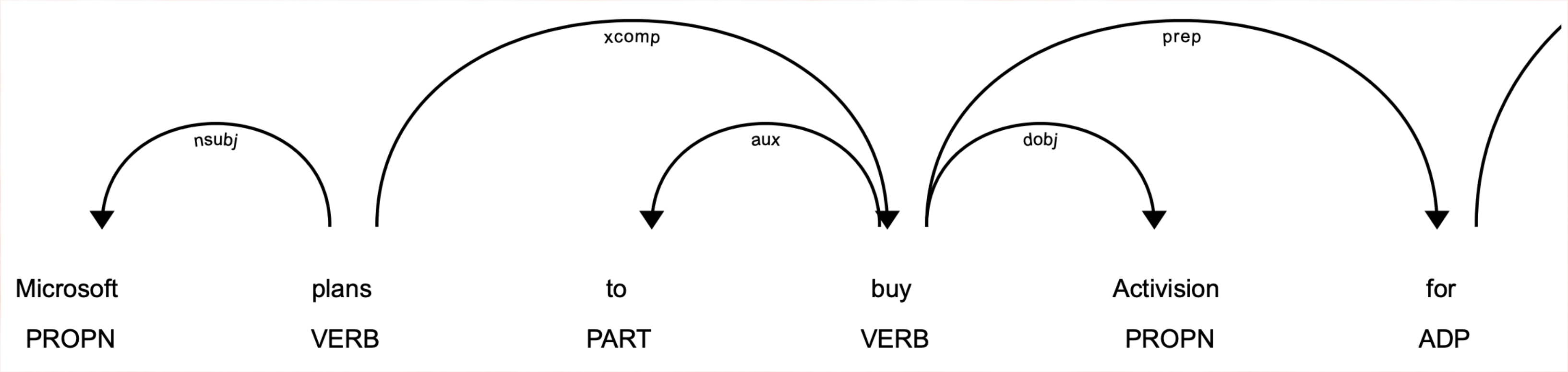
Apple	Apple	PROPN	NNP	nsubj	Xxxxx	True	False
is	be	AUX	VBZ	aux	xx	True	True
looking	look	VERB	VBG	ROOT	xxxx	True	False
at	at	ADP	IN	prep	xx	True	True
buying	buy	VERB	VBG	pcomp	xxxx	True	False
U.K.	U.K.	PROPN	NNP	compound	X.X.	False	False
startup	startup	NOUN	NN	dobj	xxxx	True	False
for	for	ADP	IN	prep	xxx	True	True
$	$	SYM	$	quantmod	$	False	False
11.1	11.1	NUM	CD	compound	dd.d	False	False
million	million	NUM	CD	pobj	xxxx	True	False
.	.	PUNCT	.	punct	.	False	False

SpaCy Visualizers

- Displaying syntactic dependences:

```
import spacy
from spacy import displacy

nlp = spacy.load("en_core_web_sm")
doc = nlp("Microsoft plans to buy Activision for $69 billion.")
displacy.render(doc, style = "dep")
```



SpaCy Visualizers

- Display named entities:

```
import spacy
from spacy import displacy

nlp = spacy.load("en_core_web_sm")
doc = nlp("Microsoft plans to buy Activision for $69 billion.")
displacy.render(doc, style = "ent")
```

Microsoft **ORG** plans to buy Activision for $69 billion **MONEY** .

- For more options and saving formats, see documentation:
  - https://spacy.io/usage/visualizers

Tokenization and Sentence Segmentation

```
import spacy

nlp = spacy.load("en_core_web_sm")
doc = nlp("Apple is looking at buying U.K. startup for $1 billion. "
          "The deal is unlikely to go through.")

for token in doc:
    print(token, end = ' ')
print()
```

Apple is looking at buying U.K. startup for $ 1 billion . The deal is unlikely to go through .

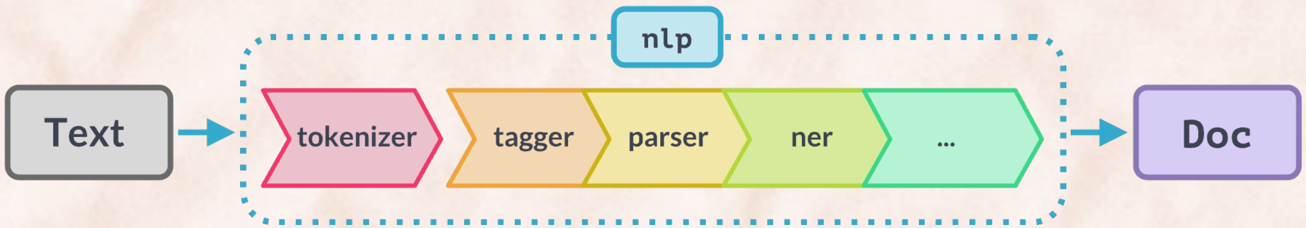
```
for sent in doc.sents:
    for token in sent:
        print(token, end = ' ')
    print()
```

Apple is looking at buying U.K. startup for $ 1 billion .
The deal is unlikely to go through .

Tokenization and Sentence Segmentation

https://spacy.io/usage/processing-pipelines

- By default, spaCy's nlp() function runs an entire linguistic pipeline:



NAME	COMPONENT	CREATES	DESCRIPTION
tokenizer PROCESSING PIPELINE	`Tokenizer`	`Doc`	Segment text into tokens.
tagger	`Tagger`	`Token.tag`	Assign part-of-speech tags.
parser	`DependencyParser`	`Token.head`, `Token.dep`, `Doc.sents`, `Doc.noun_chunks`	Assign dependency labels.
ner	`EntityRecognizer`	`Doc.ents`, `Token.ent_iob`, `Token.ent_type`	Detect and label named entities.
lemmatizer	`Lemmatizer`	`Token.lemma`	Assign base forms.
textcat	`TextCategorizer`	`Doc.cats`	Assign document labels.
custom	custom components	`Doc._.xxx`, `Token._.xxx`, `Span._.xxx`	Assign custom attributes, methods or properties.

- But this is inefficient if we only need to tokenize …

Tokenization in spaCy

https://spacy.io/api/tokenizer
https://spacy.io/usage/linguistic-features#tokenization

- Run only the pipeline component(s) that are needed. Two options:
  1. **Call the component directly.**
  2. Use the default pipeline but disable components that are not needed.

```
from spacy.lang.en import English

nlp = English()

tokenizer = nlp.tokenizer
tokens = tokenizer("U.S. economy is healing, but there’s a long way to go. "
                   "The spread of Covid-19 led to surge in orders for factory robots")
for token in tokens:
    print(token, end = ' ')
print()
```

U.S. economy is healing , but there ’s a long way to go . The spread of Covid-19 led to surge in orders for factory robots .

Tokenization in spaCy

https://spacy.io/usage/processing-pipelines

- Run only the pipeline component(s) that are needed. Two options:
  1. Call the component directly.
  2. **Use the default pipeline but disable components that are not needed.**

```
import spacy
nlp = spacy.load("en_core_web_sm", exclude = ['tagger, ner, parser'])

doc = nlp("U.S. economy is healing, but there's a long way to go. "
          "The spread of Covid-19 led to surge in orders for factory robots.")

for token in doc:
  print(token.text, end = ' ')
print()
```

U.S. economy is healing , but there ’s a long way to go . The spread of Covid-19 led to surge in orders for factory robots .

Sentence Segmentation in spaCy

https://spacy.io/api/sentencizer

- Run only the pipeline component(s) that are needed:
  - But spaCy by default uses the parser for sentence segmentation!
    - Use a rule-based (but not as accurate) Sentencizer.

```
from spacy.lang.en import English
nlp = English()

nlp.add_pipe("sentencizer")

doc = nlp("U.S. economy is healing, but there's a long way to go. "
          "The spread of Covid-19 led to surge in orders for factory robots.")

# Print tokens, one sentence per line.
for sent in doc.sents:
  for token in sent:
    print (token, end = ' ')
  print()
```

U.S. economy is healing , but there 's a long way to go .
The spread of Covid-19 led to surge in orders for factory robots .

Tokenization in spaCy

https://spacy.io/usage/linguistic-features#tokenization

```python
def tokenizer_pseudo_code(
    special_cases,
    prefix_search,
    suffix_search,
    infix_finditer,
    token_match,
    url_match
):
    tokens = []
    for substring in text.split():
        suffixes = []
        while substring:
            while prefix_search(substring) or suffix_search(substring):
                if token_match(substring):
                    tokens.append(substring)
                    substring = ""
                    break
                if substring in special_cases:
                    tokens.extend(special_cases[substring])
                    substring = ""
                    break
                if prefix_search(substring):
                    split = prefix_search(substring).end()
                    tokens.append(substring[:split])
                    substring = substring[split:]
                    if substring in special_cases:
                        continue
                if suffix_search(substring):
                    split = suffix_search(substring).start()
                    suffixes.append(substring[split:])
                    substring = substring[:split]
            if token_match(substring):
                tokens.append(substring)
                substring = ""
            elif url_match(substring):
                tokens.append(substring)
                substring = ""
            elif substring in special_cases:
                tokens.extend(special_cases[substring])
                substring = ""
            elif list(infix_finditer(substring)):
                infixes = infix_finditer(substring)
                offset = 0
                for match in infixes:
                    tokens.append(substring[offset : match.start()])
                    tokens.append(substring[match.start() : match.end()])
                    offset = match.end()
                if substring[offset:]:
                    tokens.append(substring[offset:])
                substring = ""
            elif substring:
                tokens.append(substring)
                substring = ""
        tokens.extend(reversed(suffixes))
    return tokens
```

Tokenization in NLTK

https://www.nltk.org/book/ch03.html

Default word tokenizer in NLTK:

```
>>> import nltk, re, pprint
>>> from nltk import word_tokenize
>>> from urllib import request
>>> url = http://www.gutenberg.org/files/2554/2554-0.txt
>>> response = request.urlopen(url)
>>> raw = response.read().decode('utf8')
>>> tokens = word_tokenize(raw)
>>> len(tokens)
254354
 >>> tokens[:10]
['The', 'Project', 'Gutenberg', 'EBook', 'of', 'Crime', 'and', 'Punishment', ',', 'by']
```

Custom tokenization through regular expressions in NLTK:

```
>>> text = 'That U.S.A. poster-print costs $12.40...'
>>> pattern = r'''(?x)    # set flag to allow verbose regexps
...     ([A-Z]\.)+        # abbreviations, e.g. U.S.A.
...   | \w+(-\w+)*        # words with optional internal hyphens
...   | \$?\d+(\.\d+)?%?  # currency and percentages, e.g. $12.40, 82%
...   | \.\.\.            # ellipsis
...   | [][.,;"'?():-_`]  # these are separate tokens; includes ], [
... '''
>>> nltk.regexp_tokenize(text, pattern)
['That', 'U.S.A.', 'poster-print', 'costs', '$12.40', '...']
```

Statistical Properties of Text

Statistical Properties of Text

- **Zipf's Law** models the distribution of terms 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 and properties of compression algorithms.

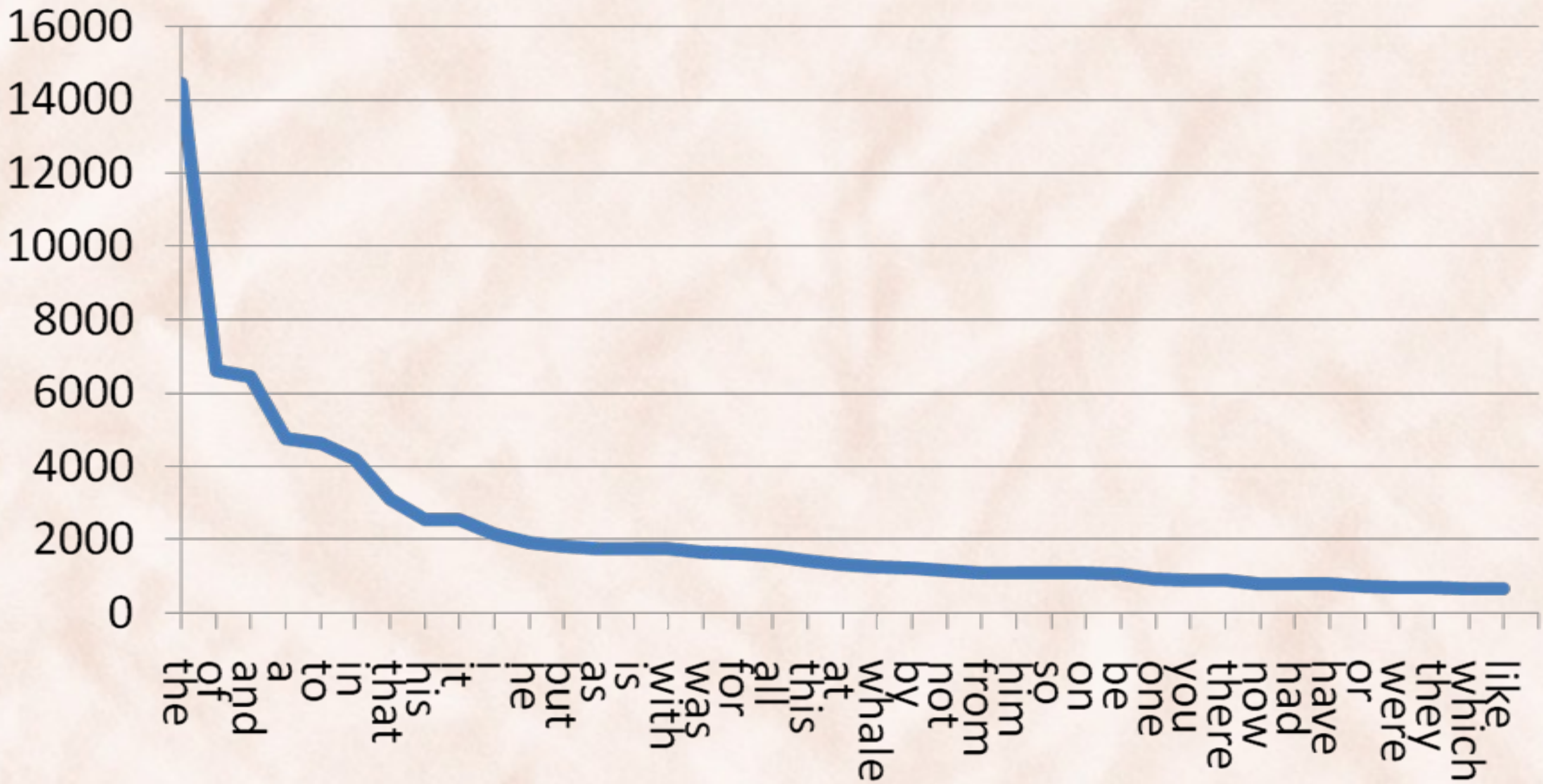
- **Heap's Law** models the number of words in the vocabulary as a function of the corpus size:
  - What is the number of unique words appearing in a corpus of size N words?
  - This determines how the size of the inverted index in IR will scale with the size of the corpus .

Word Distribution

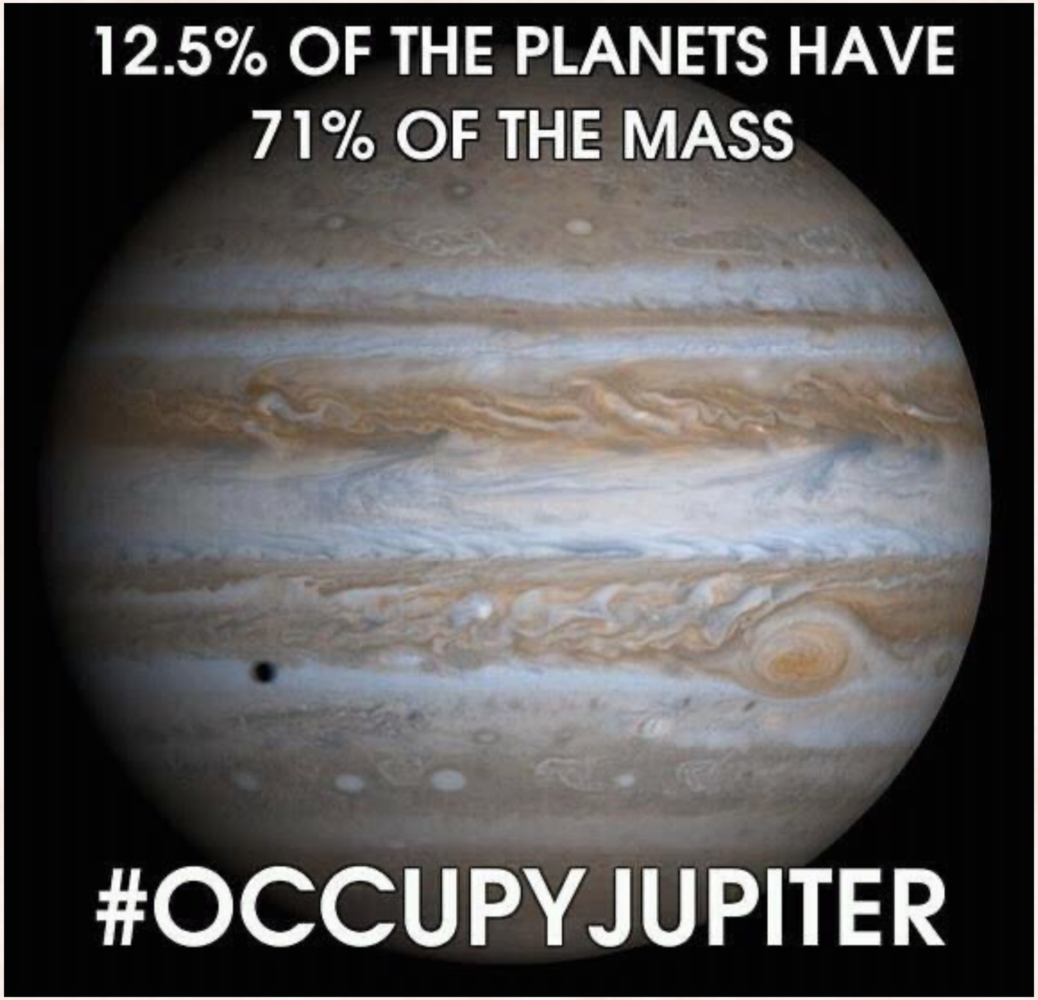
- **A few words are very common**:
  - The 2 most frequent words (e.g. “the”, “of”) can account for about 10% of word occurrences.
- **Most words are very rare**:
  - Half the words in a corpus appear only once, called *hapax legomena* (Greek for “read only once”)
- A “*heavy tailed*” or “*long tailed*”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

Word Distribution

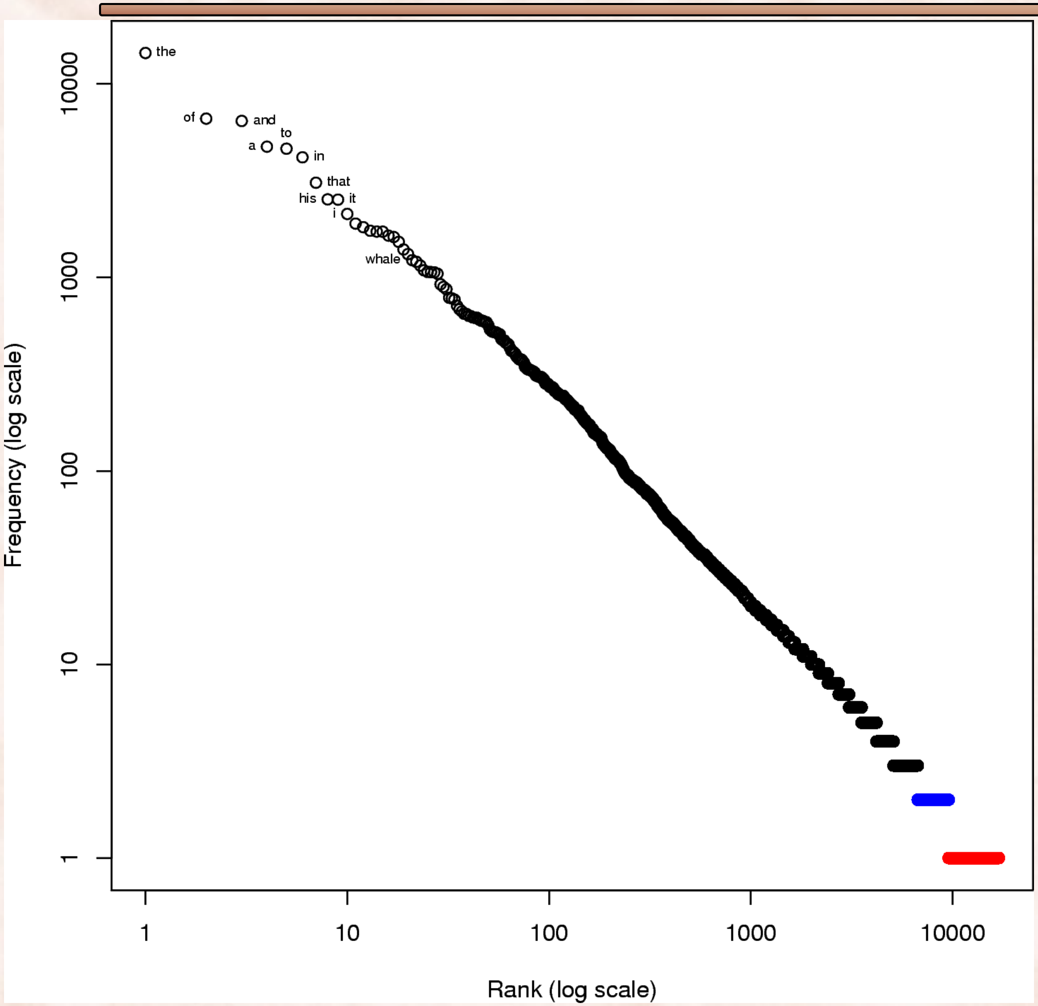
Frequency vs. rank for all words in Moby Dick.



Zipf's Law



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*:

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

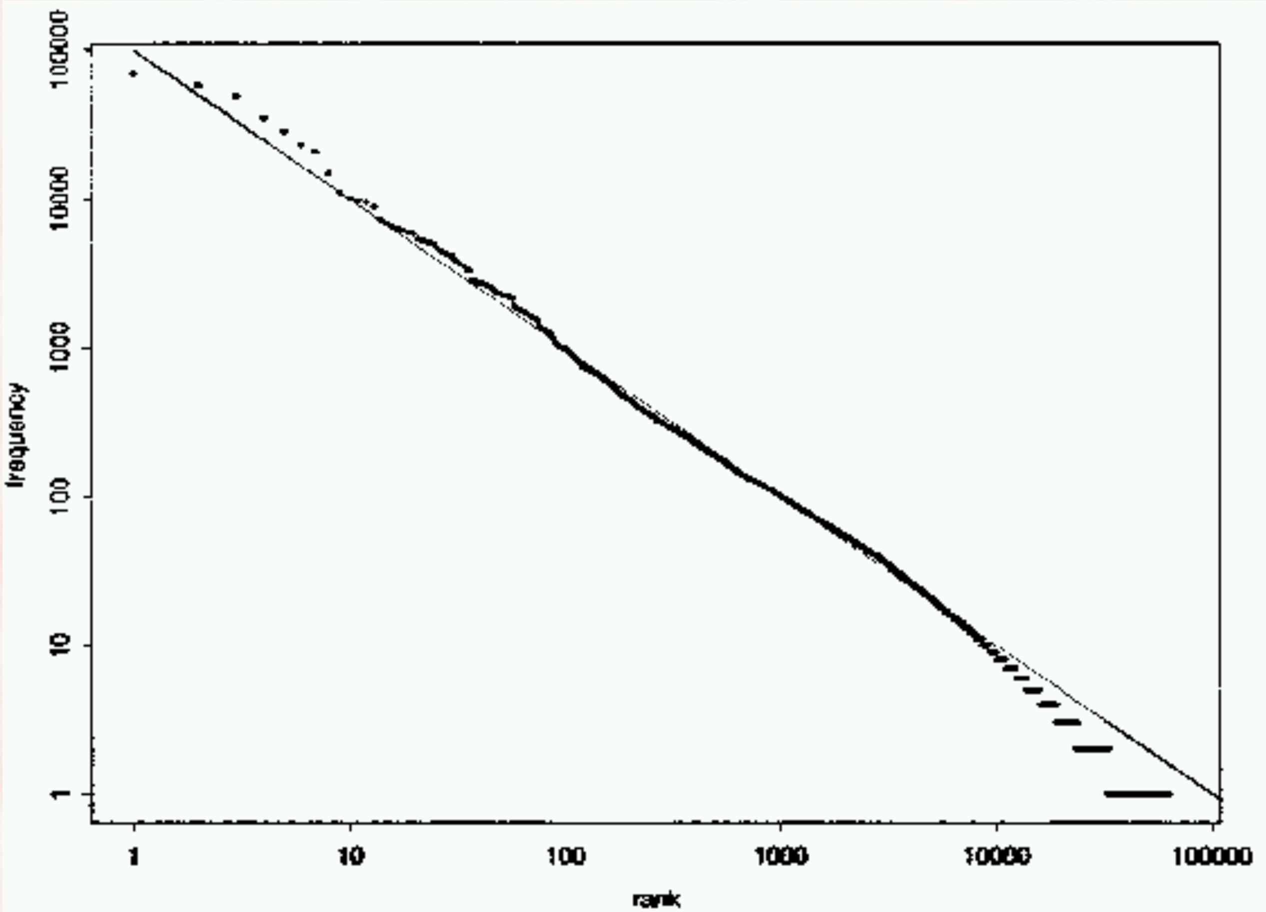
  - c is a normalization constant that depends on the corpus.

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

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



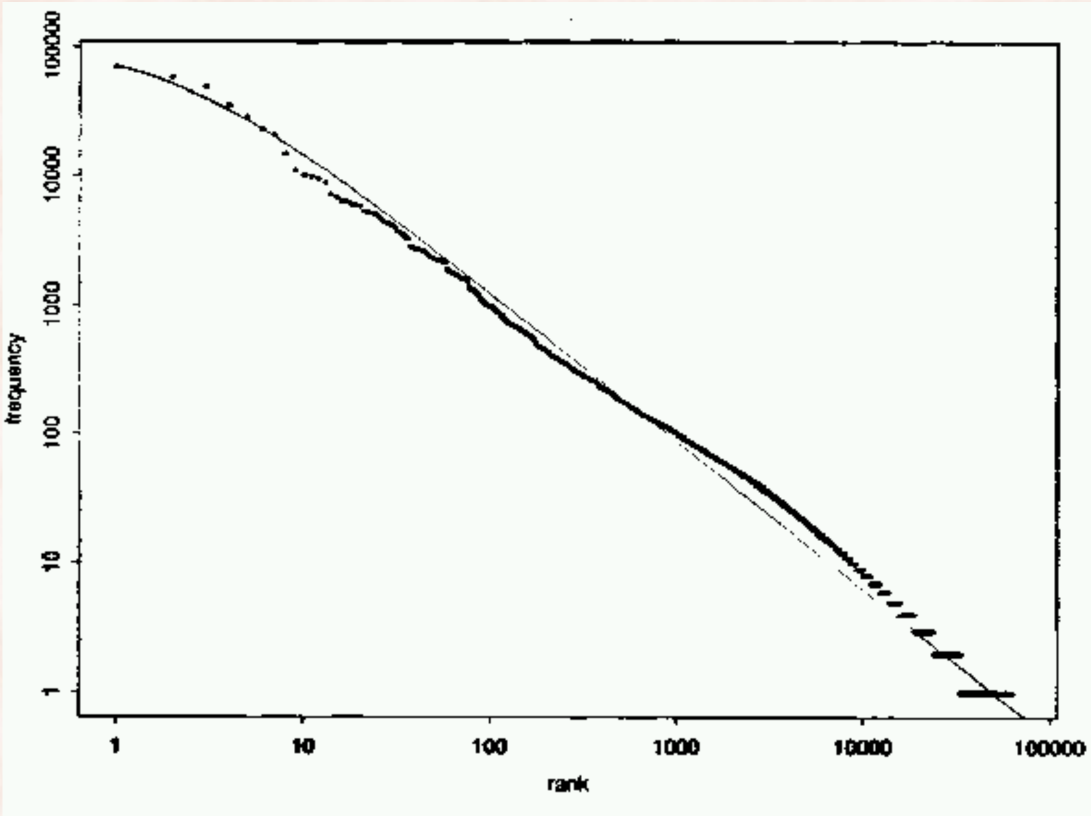
Mandelbrot's Distribution

- 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

Vocabulary vs. Collection Size

- How big is the term vocabulary?
  - That is, how many distinct words are there?
- Can we assume an upper bound?
  - Not really upper-bounded due to proper names, typos, etc.
- In practice, the vocabulary will keep growing with the collection size.

Heap's Law

- Given:
  - M is the size of the vocabulary.
  - T is the number of tokens in the collection.
- Then:
  - $M = kT^b$
  - k, b depend on the collection type:
    - typical values: $30 \leq k \leq 100$ and $b \approx 0.5$ (square root).
    - in a log-log plot of M vs. T, Heaps' law predicts a line with slope of about ½.

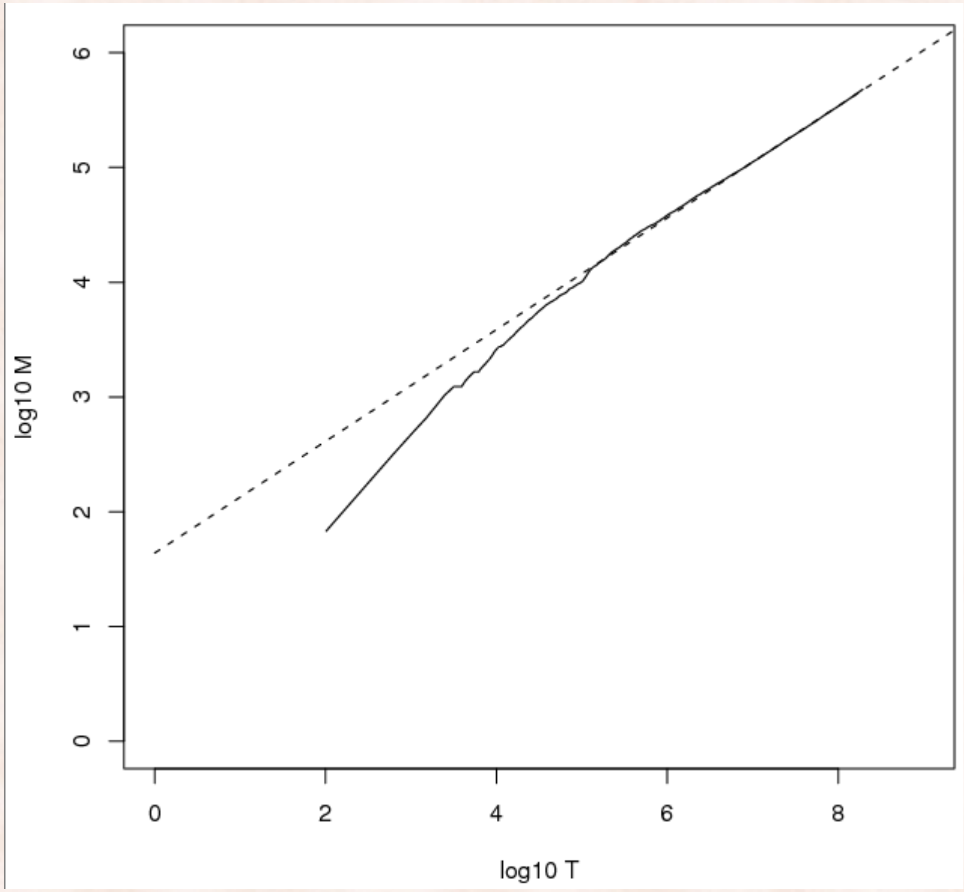
Heap's Law Fit to Reuters RCV1

- For RCV1, the dashed line $\log_{10}M = 0.49 \log_{10}T + 1.64$ is the best least squares fit.

- Thus, $M = 10^{1.64}T^{0.49}$ so $k = 10^{1.64} \approx 44$ and $b = 0.49$.

- For first 1,000,020 tokens:
  - Law predicts 38,323 terms;
  - Actually, 38,365 terms.

  $\Rightarrow$ Good empirical fit for RCV1!



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.

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.
- Can include 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.
- So first add a special end-of-word symbol '__' before whitespace in training corpus:
  - Homework exercise:
    - Design a RE and write Python code to do this substitution.
- Next, separate into letters.

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		**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 _	

Merge e r to er

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 _	

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 _`	

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

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, er_

Merge n e to ne

corpus

5 l o w _
2 l o w e s t _
6 ne w er_
3 w i d er_
2 ne w _

vocabulary

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

Byte Pair Encoding (BPE)

The next merges are:

Merge	**Current Vocabulary**
(ne, w)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new
(l, o)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo
(lo, w)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo, low
(new, er_)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo, low, newer_
(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.

WordPiece Tokenizer

- Used by for 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

Recommended Readings

- Section 2.2, 2.3, and 2.4 in J & M.
- HuggingFace summary of tokenization techniques.