Text Preprocessing

CS4422/7263 Information Retrieval
Lecture 02

Jiho Noh

Department of Computer Science
Kennesaw State University

CS4422/7263 Spring 2025

KENNESAW STATE UNIVERSITY
COLLEGE OF COMPUTING AND SOFTWARE ENGINEERING

1. **Text Statistics**
   - Zipf's law
   - Heap's law

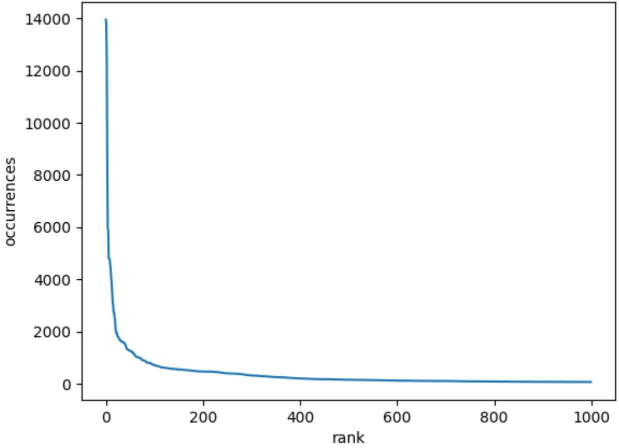
2. **Parsing Documents**
   - Regular expression
   - Parsing structured document
   - tokenization / lemmatization / stemming

Word Frequency

rank	term	freq.	perc.	rank	term	freq.	perc.
1	and	15539	0.031	16	on	2991	0.006
2	the	12164	0.025	17	university	2928	0.006
3	of	9315	0.019	18	contact	2603	0.005
4	to	7990	0.016	19	about	2558	0.005
5	&	6512	0.013	20	search	2430	0.005
6	/	5743	0.012	21	information	2351	0.005
7	for	5333	0.011	22	faculty	2316	0.005
8	in	5178	0.01	23	student	2217	0.004
9	campus	4566	0.009	24	you	2203	0.004
10	ksu	4496	0.009	25	is	2201	0.004
11	a	4314	0.009	26	with	2161	0.004
12	kennesaw	4156	0.008	27	community	2014	0.004
13	students	3361	0.007	28	programs	2013	0.004
14	research	3146	0.006	29	global	1978	0.004
15	state	3065	0.006	30	marietta	1885	0.004

30 most common words from the 1,000 scraped KSU webpages

Word Frequency Plot



Right-skewed plot with a long tail

Word Distribution

- A few words are very common (e.g. ‘the’, ‘and’, ‘of’, etc.)
  - Top 5 most frequent words can account for about 10% of word occurrences
- More than half of the words occur only once ($26,726/42,717 \approx 63\%$)
- In our example, words specific to “KSU” can be found (e.g., ‘kennesaw’, ‘ksu’, ‘university’, ‘campus’, etc.)

Word Frequency (Stopwords Removed)

rank	term	freq.	perc.	rank	term	freq.	perc.
1	campus	4566	0.009	16	marietta	1885	0.004
2	ksu	4496	0.009	17	resources	1873	0.004
3	kennesaw	4156	0.008	18	home	1855	0.004
4	students	3361	0.007	19	staff	1773	0.004
5	research	3146	0.006	20	program	1677	0.003
6	state	3065	0.006	21	diversity	1665	0.003
7	university	2928	0.006	22	ga	1564	0.003
8	contact	2603	0.005	23	©	1445	0.003
9	search	2430	0.005	24	2021	1441	0.003
10	information	2351	0.005	25	college	1354	0.003
11	faculty	2316	0.005	26	online	1346	0.003
12	student	2217	0.004	27	alumni	1308	0.003
13	community	2014	0.004	28	us	1303	0.003
14	programs	2013	0.004	29	safety	1247	0.003
15	global	1978	0.004	30	financial	1169	0.002

30 most common words from the 1,000 scraped KSU webpages

Zipf's law

- **rank (r)** is the numerical position of a word in the list sorted by decreasing **word frequency** (f)
- *Zipf (1949)* "discovered" empirical evidence such that:

$$f \cdot r = k,$$

  where k is a constant value characterizing the distribution
- If we define the probability of a word of rank r in a corpus (P_r) as the occurrence proportion where N is the total number of word occurrences,

$$P_r = \frac{f}{N} = \frac{k}{Nr} = \frac{A}{r},$$

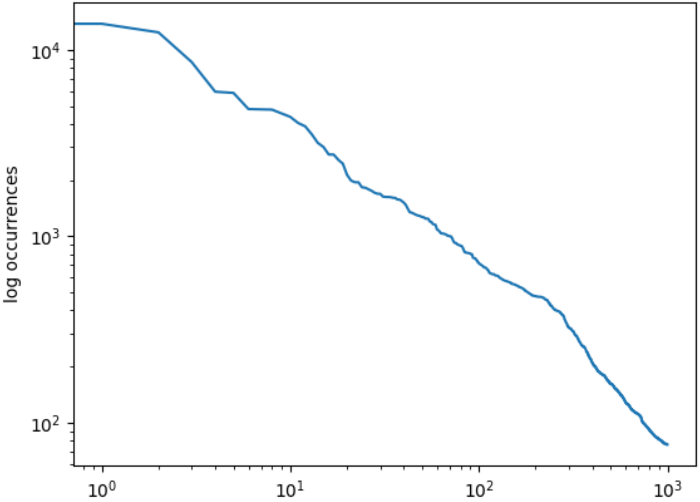
  where k, N, A are all constants which depend on the corpus word distribution.

Zipf's law

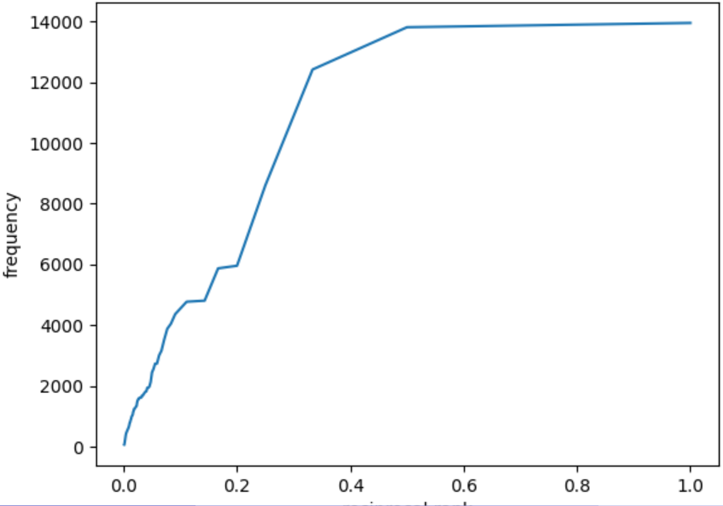
- Zipf's law is an empricial law which can be generalized to the power law ($y = kx^c$)
- Specifically, Zipf's law is a power law with $c = -1$
- On a *log-log plot*, power laws become linear functions with the slope c

$$\log(y) = \log(kx^c) = \log(k) + c\log(x)$$

Word Frequency Plot (log-log)



Word Frequency Plot (reciprocal rank)



Explanations for Zipf's law

- **George Kingsley Zipf** said,
  - Principle of least effort: balance between speacker's desire for small vocabulary and hearer's desire for a large one
- **Herbert Simon** said,
  - Rich get richer: similar distributions as seen in physics and sociology.

Stopwords

- *function words* with high frequency (e.g., 'of', 'and', 'the') takes a large proportaion of index terms. We call these words **stopwords**.
- We can eliminate stopwords to reduce inverted-index storage costs and retrieval computation.

NLTK English Stopwords

i me my myself we our ours ourselves you your yours yourself yourselves he him his himself she her hers herself it its itself they them their theirs themselves what which who whom this that these those am is are was were be been being have has had having do does did doing a an the and but if or because as until while of at by for with about against between into through during before after above below to from up down in out on off over under again further then once here there when where why how all any both each few more most other some such no nor not only own same so than too very s t can will just don should now

Stopwords

- Traditional Approach
  - Remove stopwords, especially for information retrieval processes
- Issues with Stopwords; Removing stopwords can
  - loose the context of a free flowing text
  - damage important terms such as "to be or not to be"
- Modern Approach
  - Especially with deep learning NLP methods, do not remove stopwords

Vocabulary Size

- Vocabulary size means the number of *unique* words
- The rate of vocabulary growth diminishes as the number of documents in a corpus increases
- This pattern can be formulated and used for determining the size of the inverted index that will scale with the size of the corpus

1. **Text Statistics**
   - Zipf's law
   - Heap's law
2. **Parsing Documents**
   - Regular expression
   - Parsing structured document
   - tokenization / lemmatization / stemming

Heap's Law

$$V = Kn^{\beta},$$

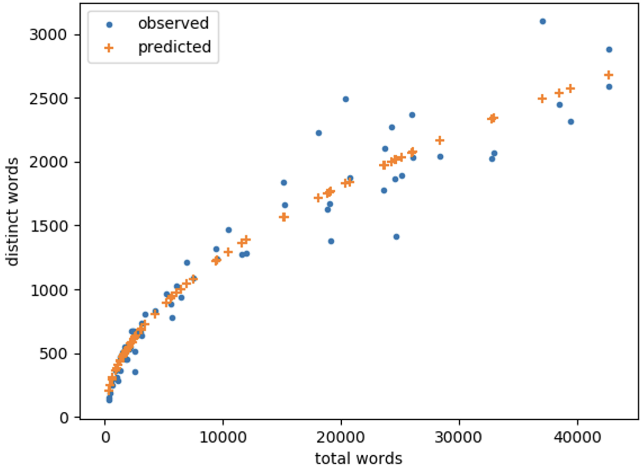
where V is the vocabulary size. n is the length of the corpus in words. K is a constant in $\mathbb{R}$ and $0 < \beta < 1$.

Typically, we use following constant for the English vocabulary

- $K \approx$ 10-100
- $\beta \approx$ 0.4-0.6

Heap's Law — King James Bible

Book	n	V
Genesis	38520	2448
Exodus	32767	2024
Leviticus	24621	1412
...		
III John	295	155
Jude	609	295
Revelation	12003	1283



Heap's Law — Exercise

We want to estimate the size of the vocabulary for a corpus of 1,000,000 words. However, we only know statistics computed on smaller corpora sizes:

- For 100,000 words, there are 50,000 unique words
- For 500,000 words, there are 150,000 unique words

Estimate the vocabulary size for the 1,000,000 words corpus.

1. **Text Statistics**
   - Zipf's law
   - Heap's law

2. **Parsing Documents**
   - Regular expression
   - Parsing structured document
   - tokenization / lemmatization / stemming

Regular Expression

```
^(\+\d{1,2}\s)?\(?\d{3}\)?[\s.-]\d{3}[\s.-]\d{4}$
```

- Also called *regex* or *regexp*
- A concise and flexible means for matching string patterns
- Mandatory skill for data processing with textual inputs such as NLP

Regular Expression
Brief History

- Developed in theoretical computer science and formal language theory in 1951
- Became popular from 1968 for pattern matching in a text editor and a lexical analysis in a compiler
- In the 1980s, more complicated regexes arose in Perl
- Today, most of modern programming languages support the regex search
- “Old school” language, but fast/compact/efficient way to express string patterns
- http://xkcd.com/208/

Regular Expression — Meta-characters

Each character in a regular expression is either understood to be a meta-character with its special meaning, or a regular character with its literal meaning.

Meta-characters

`.`   `+`   `?`   `*`   `^`   `$`   `[...]`   `-`   `[^...]`   `|`   `()`   `{m,n}`

Regular Expression — Meta-characters

Meta-characters

`.   +   ?   *   ^   $   [...]   -   [^...]   |   ()   {m,n}`

. (dot)

- The dot sign matches any single character.
- For example, `a.b` matches any string that contains an 'a', then any other character and then 'b'
- [.] matches literally a dot

Regular Expression — Meta-characters

Meta-characters

`.   +   ?   *   ^   $   [...]   -   [^...]   |   ()   {m,n}`

+

- The plus sign indicates one or more occurrences of the preceding element.
- For example, `ab+c` matches 'abc', 'abbc', 'abbbc', and so on, but not 'ac'.

Regular Expression — Meta-characters

Meta-characters

`.   +   ?   *   ^   $   [...]   -   [^...]   |   ()   {m,n}`

?

- The question mark indicates zero or one occurrences of the preceding element.
- For example, `colou?r` matches both 'color' and 'colour'.

Regular Expression — Meta-characters

Meta-characters

`.   +   ?   *   ^   $   [...]   -   [^...]   |   ()   {m,n}`

*****

- The asterisk indicates zero or more occurrences of the preceding element.
- For example, `ab*c` matches ‘ac’, ‘abc’, ‘abbc’, ‘abbbc’, and so on.

Regular Expression — Meta-characters

Meta-characters

`.   +   ?   *   ^   $   [...]   -   [^...]   |   ()   {m,n}`

^

- The caret matches the starting position within the string or a line.
- For example, `^The` matches any line that starts with the characters 'The'.

Regular Expression — Meta-characters

Meta-characters

`.   +   ?   *   ^   $   [...]   -   [^...]   |   ()   {m,n}`

$

- The dollar sign matches the ending position of the string.
- For example, `^The.*[.!]$` matches any line that starts with the characters 'The' and ends with either period or exclamation mark.

Regular Expression — Meta-characters

Meta-characters

`.   +   ?   *   ^   $   [...]   -   [^...]   |   ()   {m,n}`

[...]

- A bracket expression matches a single character that is contained within the brackets.
- For example, `[abc]` matches a signle character either 'a', 'b', or 'c'. `[a-z]` specifies a range which matches any lowercase letter from 'a' to 'z'. `[a-zA-Z0-9]` indicates any *alphanumeric* character.

Regular Expression — Meta-characters

Meta-characters

`.   +   ?   *   ^   $   [...]   -   [^...]   |   ()   {m,n}`

[^...]

- Matches a single character that is not contained within the brackets.
- For example, `[^a-zA-Z0-9]` indicates any character that is **not in** the *alphanumeric* characters.

Regular Expression — Meta-characters

Meta-characters

`.   +   ?   *   ^   $   [...]   -   [^...]   |   ()   {m,n}`

| (boolean OR)

- A vertical bar separates alternatives.
- For example, `gray|grey` can match "gray" or "grey".

Regular Expression — Meta-characters

Meta-characters

```
.   +   ?   *   ^   $   [...]   -   [^...]   |   ()   {m,n}
```

( )

- Parenthesis defines a marked subexpression. The string matched within the parentheses can be recalled later.
- For example

```
>>> import re
>>> str = "30062-1234"
>>> m = re.match(r"^(\d{5})(?:[-\s](\d{4}))?$", str)
>>> m.group(0)
'30062-1234'
```

Regular Expression — Meta-characters

Meta-characters

`.   +   ?   *   ^   $   [...]   -   [^...]   |   ()   {m,n}`

{m,n}

- Matches the preceding element at least *m* and not more than *n* times.
- For example, `a{3,5}` matches only "aaa", "aaaa", and "aaaaa".

Regular Expression — Examples

Hexadecimal color values

```
/^#?([A-Fa-f0-9]{6}|[A-Fa-f0-9]{3})$/
```

Email addresses

```
/^([a-z0-9_.-]+)@([\da-z.-]+)\.([a-z.]{2,6})$/
```

In Python,

```
>>> email_pattern = r"^([a-z0-9_.-]+)@([\da-z.-]+)\.([a-z.]{2,6})$"
>>> str1 = "john.doe@gmail.com"
>>> m = re.match(email_pattern, str1)
>>> m.group(1)
'john.doe'
>>> m.group(2)
'gmail'
>>> m.group(3)
'com'
```

1. **Text Statistics**
   - Zipf's law
   - Heap's law

2. **Parsing Documents**
   - Regular expression
   - Parsing structured document
   - tokenization / lemmatization / stemming

Parsing Structured Documents

Markup languages

A **markup language** is a system for annotating a document in a way that is visually distinguishable from the content.

- **type setting**: TeX, troff, LaTeX
- **markup meta-languages**: SGML, XML, HTML
- **lightweight**: Markdown, MediaWiki markup language

Markup Language Examples

LaTeX example

```
\begin{itemize}
  \item \textbf{type setting}: TeX, troff, LaTeX
  \item \textbf{markup meta-languages}: SGML, XML, HTML
  \item \textbf{lightweight}: Markdown, MediaWiki markup language
\end{itemize}
```

HTML example

```
<ul id="lst_markup_languages">
  <li><b>type setting</b>: TeX, troff, LaTeX</li>
  <li><b>markup meta-language</b>: SGML, XML, HTML</li>
  <li><b>lightweight</b>: Markdown, MediaWiki markup language</li>
</ul>
```

XML vs. HTML

Both of XML and HTML are the offsprings of Standard Generalized Markup Language (SGML)

XML	HTML
- extensible set of tags	- fixed set of tags
- content oriented	- presentation oriented
- standard data infrastructure	- no data validation capability
- allows multiple output forms	- single presentation

XML Elements

- An XML element is made up of a start tag, an end tag, and data in between.

```
<autho>Jiho Noh and Technoblade</author>
```

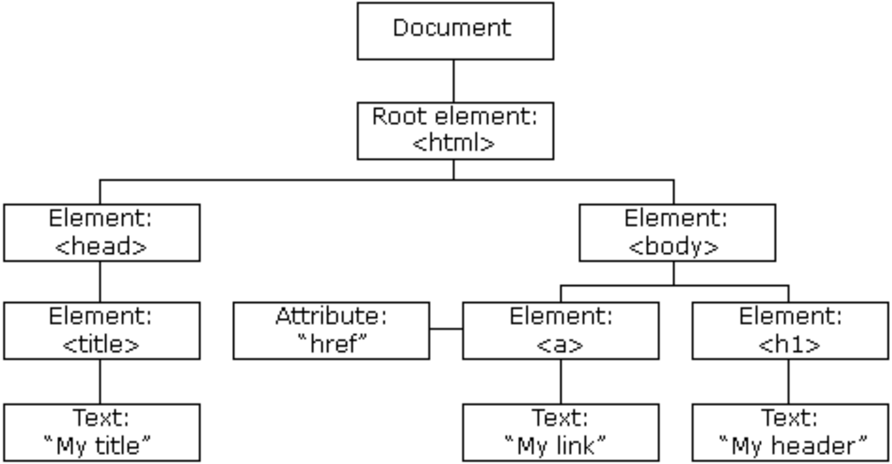
- XML can abbreviate empty elements.

```
<img src="..."/>
<br/>
```

- An attribute is a name-value pair separated by =.

```
<address zip="12345">1234 Hamilton Rd.</address>
```

XML Document Tree



XML DOM Model

- **Document Node** - Complete XML document structure is a document node.
- **Element Node** - Every XML element is an element node. This is also the only type of node that can have attributes.
- **Attribute Node** - Each attribute is considered an attribute node. It contains information about an element node, but is not actually considered to be children of the element.
- **Text Node** - The document texts are considered as text node. It can consist of more information or just white space.

XPATH

- An XML query language to search for features in XML (also HTML) documents
- XPATH describes paths to elements
    - Look like UNIX path description with tags instead of directories and files
    - Simple path descriptors are sequences of tags seperated by slashes (/)

XPATH — Selecting Nodes

Expression	Description
nodename	Selects all nodes with the name “nodename”
/	Beginning single slash selects from the root node
//	Double slash selects nodes in the document from the current node that match the selection no matter where they are
.	Selects the current node in context
..	Selects the parent of the current node

examples

```
/html/body/div[@id="introduction"]/div[3]/article
//p[@class="book_description"]
//div[@id="music"]/../p[0]
```

XPATH — Wildcards

Wildcard	Description
*	Matches any element node
@*	Matches any attribute node
node()	Matches any node of any kind

examples

```
//*
/bookstore/*
//book[@*]    <!-- select all books which have at
                   least one attribute of any kind -->
```

Demo — Python lxml package

Download HTML source from a URL and extract all href links

```
import requests
from lxml import etree, html

url = 'https://jiho.us/index.html'
resp = requests.get(url, 'html.parser')
tree = html.fromstring(resp.content)
print(etree.tostring(tree, encoding='unicode',
                     pretty_print=True))
etree.XPath('//a/text()')(tree)
etree.XPath('//a/@href')(tree)
```

link: lxml.de

Demo — Python BeautifulSoup library

Download HTML source from a URL and extract all href links

```
from bs4 import BeautifulSoup as BS
import re

soup = BS(resp.content)
for link in soup.findAll('a',
               attrs={'href': re.compile('^http[s]?://')}):
    print(link.get('href'))
```

link: BeautifulSoup4

1. **Text Statistics**
   - Zipf's law
   - Heap's law

2. **Parsing Documents**
   - Regular expression
   - Parsing structured document
   - tokenization / lemmatization / stemming

Text Normalization

One of the first steps in the transformation process from natural language to numeric features is **text normalization**, which includes *tokenization, lemmatization, and stemming*.

- **Tokenization**: Sequence segmentation in running text
- **Lemmatization**: Reducing inflections or variant forms of words to base form (i.e., lemma)
- **Stemming**: Reducing terms to their stems for information retrieval needs

Word Count

How many words?

Korean popular music, or "K-pop," became one of South Korea's most visible cultural exports.

- **Type**: an element of the vocabulary
- **Token**: an instance of that type in running text
- N = number of tokens
- V = vocabulary (set of types)
- $|V|$ is the size of the vocabulary

Tokenization

Identifying word boundaries is where the process of tokenization comes in.
Let's give the most naive definition of tokens as the alphabet chunks separated by non-alphabet characters

Demo — Tokenize text using UNIX commands

```
tr -sc 'a-zA-Z' '\n' < the-book.txt
    | tr -s 'A-Z' 'a-z'
    | sort
    | uniq -c
    | sort -nr
    | less
```

Challenges in Tokenization

- apostrophes (possession, contraction, etc.): `lawyer's, isn't, I'm`
- multi-word terms: `state-of-the-art, South Korea`
- abbreviations: `e.g., m.p.h.`
- Chinese, Japanese do not use whitespaces at all !!.

There's no standardized rules

Word Normalization

- Applications like IR: reduce all letters to lower case
  - Most of the words are in lower case
  - In most cases, capitalized word has the same meaning of the lower cased one
- In machine learning applications: case is helpful
  - **US** is different from **us**. Even **They** and **they** have diffrent contextual meanings.

Lemmatization

In morphology, **Lemma** is the **canonical form, dictionary form, or citation form** of a set of words (headword).

variant form	base forms
breaking, broke, broken, breaks	break
car, cars, car's, cars'	car
am, are, is	be

Lemmatization example using NLTK with WordNet PoS

```
>>> s  = "A group of students is studying " \
         "literature studies in a study room."

>>> lemmatiser.lemmatize(s)
"A group of student be study literature study in a study room ."
```

Stemming

Stemming is removing the affixes from a word and reduce it to its root word.
In morphology, an **affix** is an additional element placed at the beginning or end of a root, **stem**, or a word, to modify its meaning.

Stemming example

```
['broken', 'lemmatization', 'beautiful', 'traditional', 'plotted']]
to
['broken', 'lemmat', 'beauti', 'tradit', 'plot']
```

Lemmatization vs. Stemming — Difference?

Stemming

- a crude heuristic process that chops off the ends of words
- aiming to remove derivational affixes

Lemmatization

- finds the base form more properly by using available vocabulary and morphological analysis of words
- returns the dictionary form of a word, which is known as the *lemma*

Word Normalization?

- Tokenization/lemmatization/stemming are destructive processes.
- Language dependent
- These techniques are designed with *recall* in mind, such as in search engines.
- If the goal of your application is *precision*, then you may not need these techniques.

Text Preprocessing

An example pipeline of Text Preprocessing

1. Lower casing
2. Removal of Punctuations
3. Removal of Stopwords
4. Removal of Frequent words
5. Removal of Rare words
6. Stemming
7. Lemmatization
8. Removal of emojis
9. Removal of emoticons
10. Conversion of emoticons to words

Summary

- Tokenization/lemmatization/stemming are destructive processes.
- Language dependent
- These techniques are designed with recall in mind, such as in search engines.
- If the goal of your application is precision, then you may not need these techniques.

questions? discussion?