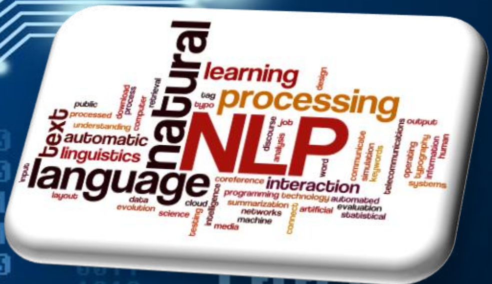
Cairo University
Faculty of Engineering
Computer Engineering Department

Natural Language Processing

Dr. Sandra Wahid



Agenda

- Introduction
- Regular Expressions
- Text Normalization
  - Word Tokenization
  - Word Normalization
  - Sentence Segmentation

What is Natural Language Processing (NLP)?

- A field that deals with Artificial Intelligence (AI) that is concerned with enabling computers to **understand** and **process** human language.

Computers
- Need structured data

Human Speech
- Unstructured
- Ambiguous in nature

Ambiguous 34an feh kalemat keter btdol 3la nfs el m3na, w a2dr awsf nfs el 7aga b toro2 ktera mo5talefa, da gher el ekhtsarat w el acrynoms w keda.

da gher en mmkn a2ol 7aga, ana asdy beha 7aga, wenta tefhamha b 7aga tanya, gher el klam el leh aktur mn m3na, w gher en tre2t el voice tone aslun mmkn t8yr m3na el klam, w 7agat mn keda kter

ELIZA: An Early NLP System

User: I am unhappy.
ELIZA: DO YOU THINK COMING HERE WILL HELP YOU NOT TO BE UNHAPPY
User: I need some help, that much seems certain.
ELIZA: WHAT WOULD IT MEAN TO YOU IF YOU GOT SOME HELP
User: Perhaps I could learn to get along with my mother.
ELIZA: TELL ME MORE ABOUT YOUR FAMILY
User: My mother takes care of me.
ELIZA: WHO ELSE IN YOU FAMILY TAKES CARE OF YOU
User: My father.
ELIZA: YOUR FATHER
User: You are like my father in some ways.

Weizenbaum (1966)

- Simple program that uses **pattern matching** to recognize phrases like “I need X” and translate them into suitable outputs like “What would it mean to you if you got X?”.
- Many people who interacted with ELIZA came to believe that it really **understood** them and their problems.
- One of the most important tools for describing text patterns:

  the **regular expression**.

Regular Expressions

- Formally, a regular expression is an **algebraic notation** for characterizing a set of strings.

- Useful for **searching** in texts, we have
  - A pattern to search for
  - And a corpus of texts to search through.

- Regular expressions are **case sensitive**.

- To test our REs, we will use the online tool RegEx Pal:

  https://www.regexpal.com/ → Can set flag to ignore case

- Note: we may show some regular expressions delimited by slashes but slashes are not part of the regular expressions.

Regular Expressions: Simple text

== find text

String:

Regular Expression

```
/hello/g
```

Test String

```
tasdgfas Hello svavsg hello ghgdfhello
```

Single Character:

Regular Expression

```
/!/g
```

Test String

```
Hi! WE are practicing Regular Expressions ! here.
```

it tries to match the largest sequence

VIP

Regular expressions always match the largest string they can → Greedy

Regular Expressions: Disjunction

- Square Bracket [ ]→match on any one of the characters in the list enclosed by []

Regular Expression

```
/[hH]ello/g
```

Test String

```
Hello! WE are practicing Regular Expressions here!
Say hello to all.
GHello and dfghellofsdhfksj
```

RE	Match	Example Patterns
/[wW]oodchuck/	Woodchuck or woodchuck	"Woodchuck"
/[abc]/	'a', 'b', *or* 'c'	"In uomini, in soldati"
/[1234567890]/	any digit	"plenty of 7 to 5"

Here shows the first match only

Regular Expressions: Range

- Dash –
  - The pattern /[2-5]/ specifies any one of the characters 2, 3, 4, or 5.
  - The pattern /[b-g]/ specifies one of the characters b, c, d, e, f, or g.

RE	Match	Example Patterns Matched
/[A-Z]/	an upper case letter	"we should call it 'Drenched Blossoms' "
/[a-z]/	a lower case letter	"my beans were impatient to be hoed!"
/[0-9]/	a single digit	"Chapter 1: Down the Rabbit Hole"

Here shows the first match only

what happens for [0-z] ?

Regular Expressions: caret ^

- The square braces can also be used to specify **what a single character cannot be**, by use of the caret ^.
  - If the caret ^ is the first symbol after the open square brace [, the resulting pattern is negated.
  - For example, the pattern /[^a]/ matches any single character (including special characters) except a.
  - This is only true when the caret is the first symbol after the open square brace.
  - If it occurs anywhere else, it usually stands for a caret.

RE	Match (single characters)	Example Patterns Matched
`/[^A-Z]/`	not an upper case letter	"Oyfn pripetchik"
`/[^Ss]/`	neither 'S' nor 's'	"I have no exquisite reason for't"
`/[^.]/`	not a period	"our resident Djinn"
`/[e^]/`	either 'e' or '^'	"look up ^ now"
`/a^b/`	the pattern 'a^b'	"look up a^ b now"

Here shows the first match only

Regular Expressions: Quantifiers

hello
helllo
how to find both?
(helll?o)

RE	Match
?	The question mark indicates ***zero or one* occurrences** of the preceding element. For example, colou?r matches both "color" and "colour".
*	The asterisk indicates ***zero or more* occurrences** of the preceding element. For example, ab*c matches "ac", "abc", "abbc", "abbbc", and so on.
+	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".
{n}	The preceding item is matched **exactly *n* times**.
{min,}	The preceding item is matched ***min* or more times**.
{,max}	The preceding item is matched **up to *max* times**.
{min,max}	The preceding item is matched **at least *min* times, but not more than *max* times**. For example, NLP{3,5} matches “NLPPP”, “NLPPPP”, “NLPPPPP” ,but not “NLP” or “NLPP”

hel{3}o can find?

hel{2,}o ?

Not present in Javascript

Regular Expressions: wildcard and Anchors

- Wildcard: Dot .

it says: at my place there should be something, it may be char, num, special char, even spaces, whatever, but something must exist.

RE	Match	Example Matches
/beg.n/	any character between *beg* and *n*	begin, beg'n, begun

- Anchors:

	RE		Match
	^		start of line
	$		end of line
mafsola	\b		word boundary
	\B	malzo2a	non-word boundary (word)

a "word" for the purposes of a regular expression is defined as any sequence of **digits**, **underscores**, or **letters**

Regular Expression
/^natural$/gm
btbd2 el satr, w btenhy el satr
Test String
natural
natural language processing
natural

Regular Expression
/\bnatural\b/gm
ablha feh ay haga msh word, w b3dha msh word
Test String
lw fe bdayt el satr, byb2a word boundry
natural naturally

Regular Expression
/\Bnatural\B/gm
Test String
34an el bdaya aslun word boundry, fa msh hmatch
natural naturally

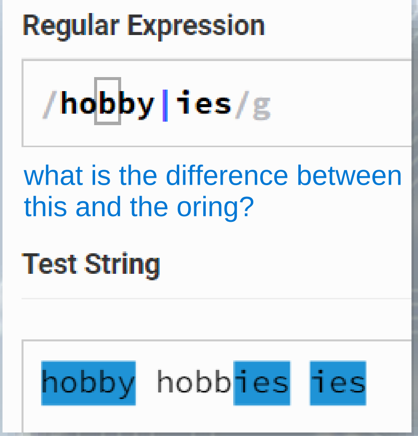
Regular Expression
/natural\B/gm
lazm tenthy b non-word boundry
Test String
hena 34an mafesh space
naturalnaturally

Regular Expression
/\Bnatural\B/gm
Test String
naturalnaturally

Regular Expression
/\b99\b/gm
Test String
99 words
99$
$99
299
993
w99

Regular Expressions: Pipe symbol and Grouping

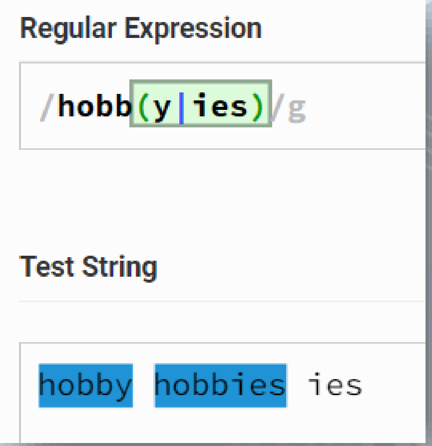
- pipe symbol |: disjunction operator oring



what is the difference between this and the oring?

el ORIng kan single character
lakn hena bn-match el strings nafsha.

- Grouping: Parenthesis ( )



Regular Expressions: Aliases and Backslash

- Aliases: to save typing for common ranges

RE	Expansion	Match	First Matches
\d	[0-9]	any digit	Party␣of␣5
\D	[^0-9]	any non-digit	Blue␣moon
\w	[a-zA-Z0-9_]	any alphanumeric/underscore	Daiyu
\W	[^\w]	a non-alphanumeric	!!!!
\s	[␣\r\t\n\f]	whitespace (space, tab)	
\S	[^\s]	Non-whitespace	in␣Concord

- Backslash: to refer to characters that are special themselves
  - Examples: ., *, [, and \
    - precede them with a backslash, (i.e., \. , *, \[, and \\).

RE	Match	First Patterns Matched
*	an asterisk "*"	"K*A*P*L*A*N"
\.	a period "."	"Dr. Livingston, I presume"
\?	a question mark	"Why don't they come and lend a hand?"
\n	a newline	
\t	a tab	

Regular Expression: Simple Example

- Write a RE to find cases of the English article *the*.

1. **the**→ wrong: misses *The* with capital T
2. **[tT]he**→ wrong: will still incorrectly return texts with *the* embedded in other words (e.g., other or theology).
3. **\b[tT]he\b**→ buggy: since won't treat underscores and numbers as word boundaries and we want to detect sequences as (the_ or the25).
4. **[^a-zA-Z][tT]he[^a-zA-Z]**→ buggy: here we specify that we want instances in which there are no alphabetic letters on either side of *the* but it misses *the* when it begins a line.
5. **(^|[^a-zA-Z])[tT]he([^a-zA-Z]|$)**→ correct: by specifying that before the *the* we require either the beginning-of-line or a non-alphabetic character, and the same at the end of the line. Problems with consecutive *the*

Regular Expressions: Types of Errors

- The process we just went through was based on fixing two kinds of errors:
  - False positives, strings that we incorrectly matched like other or there kelma msh el mfrod tetl3 bs enta tl3tha
  - False negatives, strings that we incorrectly missed, like The.

kelma el mfrod tetl3 bs enta mtl3thash

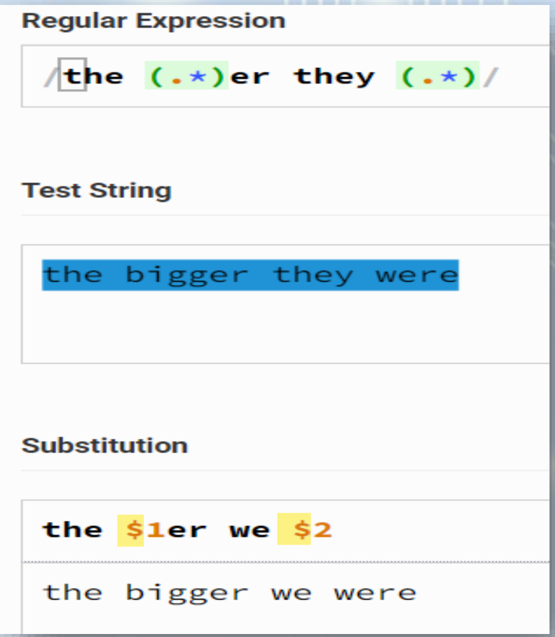
- Reducing the overall error rate for an application thus involves two antagonistic efforts:
  - Increasing **precision** (minimizing false positives) # True Positive / (# TP + #FP)
  - Increasing **recall** (minimizing false negatives)

$$R = \frac{\#\text{ True Positive}}{\#\text{ TP} + \#\text{FN}}$$

	Act 1	Act 0
Pred 1	TP	FP
Pred 0	FN	TN

Regular Expressions: Substitution

- (abc): capture group using parenthesis
- (?:non-captured group): regex engine will not number this group
- \N: backreference to group #N
  - In Javascript use $ instead of \
- Example: the (.*)er they (.*), the \1er we \2



Regular Expressions: Lookaround Assertions

- For performing matches based on information that **follows** or **precedes** a pattern, without the information within the lookaround assertion forming part of the returned text → do not consume characters in the string, but only assert whether a match is possible or not (**zero-length** assertions).

- Types of lookaround assertion:
  - *Positive Lookahead* **(?=f)** : Asserts that what immediately **follows** the current position in the string is f
    - a(?=b) will match a in abc but will not match a in acb or bac
  - *Negative Lookahead* **(?!f)** : Asserts that what immediately **follows** the current position in the string is **not** f
    - a(?!b) will match a in acb but will not match a in abc
  - *Positive Lookbehind* **(?<=f)** : Asserts that what immediately **precedes** the current position in the string is f
    - (?<=y)z will match z in xyz but will not match z in zyx
  - *Negative Lookbehind* **(?<!f)** : Asserts that what immediately **precedes** the current position in the string is **not** f
    - (?<!y)z will match z in zyx but will not match z in xyz

Text Normalization

- Before almost any natural language processing of a text, the text has to be normalized. At least **three tasks** are commonly applied as part of any normalization process:

1. Tokenizing (segmenting) words
2. Normalizing word formats
3. Segmenting sentences

Word Tokenization

- Word Tokenization: it is the task of segmenting running text into tokens: words.
  - These tokens may include numbers, punctuation or not depending on the application.
  - A tokenizer can also be used to expand **clitic** contractions that are marked by apostrophes: converting *what're* to the two tokens *what are.* A clitic is a part of a word that can't stand on its own and can only occur when it is attached to another word.
  - Tokenization algorithms may also tokenize **multiword** expressions like New York or rock 'n' roll as a single token.

- In practice, since tokenization needs to be run before any other language processing, it needs to be **very fast**.

- Word tokenization is **more complex in languages** like written Chinese, Japanese, and Thai, which do not use spaces to mark potential word-boundaries.

Word Tokenization

Why??

Unstructured data and natural language text → chunks of information that can be considered as **discrete** elements.

Unstructured string (text document) →a **numerical** data structure suitable for machine learning.

Tokens can also be used directly by a computer to trigger useful actions and responses,
or they might be used in a machine learning pipeline as features that trigger more complex decisions or behavior.

Very simple using split

Using regular expressions

Using algorithms

Word Tokenization: Split

- Separating text at each blank space.

- Example in Python using **split()**:

```
text=r'Natural language processing (NLP) is one of the most exciting aspects of machine learning and artificial intelligence.'
text.split()
```

```
['Natural',
 'language',
 'processing',
 '(NLP)',
 'is',
 'one',
 'of',
 'the',
 'most',
 'exciting',
 'aspects',
 'of',
 'machine',
 'learning',
 'and',
 'artificial',
 'intelligence.']
```

Word Tokenization: Regular Expressions

- Example of a basic regular expression that can be used to tokenize with the **nltk.regexp** tokenize function of the Python-based Natural Language Toolkit (NLTK):

VERBOSE flag: allows the user to write regular expressions that can look nicer and are more readable since whitespace within the pattern is ignored.

```
import nltk
import re

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)
```

```
[('', '', ''),
 ('A.', '', ''),
 ('', '-print', ''),
 ('', '', ''),
 ('', '', '.40'),
 ('', '', '')]
```

What's wrong ???

```
import nltk
import re

text = 'That U.S.A.poster-print costs$12.40...' OR
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', '...']
```

Change Capturing to Non-Capturing Groups using ?:

Word Tokenization: Algorithms

- Most tokenization schemes have two parts:
  - **token learner**: takes a raw training corpus and induces a vocabulary (a set of tokens).
  - **token segmenter/parser**: takes a raw test sentence and segments it into the tokens in the vocabulary.

- Three algorithms are widely used:
  1. Byte-pair encoding (Sennrich et al., 2016)
  2. Unigram language modeling (Kudo, 2018)
  3. WordPiece (Schuster and Nakajima, 2012)

Word Tokenization: Byte-pair Encoding

- The BPE token learner begins with a vocabulary that is just the **set of all individual characters**.

- It then examines the training corpus, chooses the **two symbols that are most frequently adjacent** (say '*A*', '*B*'), adds a new merged symbol '*AB*' to the vocabulary, and replaces every adjacent '*A*' '*B*' in the corpus with the new '*AB*'.

- It continues to count and merge, creating new longer and longer character strings, **until *k* merges** have been done creating *k* novel tokens; *k* is a parameter of the algorithm.

- The resulting vocabulary consists of the original set of characters plus *k* new symbols.

Word Tokenization: Byte-pair Encoding

- The algorithm is usually **run inside words**: the input corpus is first white-space-separated to give a set of strings, each corresponding to the characters of a word, plus a special end-of-word symbol.

- Example: input corpus of 18 word tokens with counts for each word (the word *low* appears 5 times, the word *newer* 6 times, and so on), which would have a starting vocabulary of 11 letters:

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 _	

Word Tokenization: Byte-pair Encoding

- The BPE algorithm first count all pairs of adjacent symbols: the most frequent is the pair *e r* because it occurs in *newer* (frequency of 6) and *wider* (frequency of 3) for a total of 9 occurrences. We then merge these symbols, treating *er* as one symbol, and count again:

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 _	

- Now the most frequent pair is *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 _	

Word Tokenization: Byte-pair Encoding

- Next *n e* (total count of 8) get merged 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 _	

- If we continue, 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_

Word Tokenization: Byte-pair Encoding

- The token learner part of the BPE, figure adapted from (Bostrom and Durrett, 2020).

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

BPE was originally a **data compression** algorithm that is used to find the best way to represent data by identifying the common byte pairs. We now use it in NLP to find the best representation of text using the **smallest number of tokens**.

Word Tokenization: Byte-pair Encoding

- Once we've learned our vocabulary, the token parser is used to tokenize a test sentence.

- The frequencies in the test data don't play a role, just the frequencies in the training data since the token parser just runs on the test data the merges we have learned from the training data, **greedily, in the order we learned them**.

- First, we segment each test sentence word into characters. Then, we apply the rules in order:
  - replace every instance of *e r* in the test corpus with *er*
  - replace every instance of *er* - in the test corpus with *er-* , and so on.

- If the test corpus contained the **previously known** word *n e w e r* -, it would be tokenized as a full word.

- If the test corpus contained **a new (unknown)** word like *l o w e r -,* it would be merged into the two tokens *low er-*.

- Of course in real algorithms BPE is run with **many thousands of merges** on a **very large input corpus**. The result is that most words will be represented as full symbols, and only the very rare words (and unknown words) will have to be represented by their parts.

Word Tokenization: Byte-pair Encoding

- **Pros:**

  - It is considered a **subword** tokenization algorithm that retains the semantic features of a token without demanding a very large vocabulary.
    - Unlike character-based models (tokens are characters): where we risk losing the semantic features of the word.
    - Unlike word-based tokenization: where we need a very large vocabulary to encompass all the possible variations of every word.

  - It ensures that most common words are represented as single tokens while rare words are broken down into two or more subword tokens allowing **out-of-vocabulary (OOV)** words to be represented.

Word Tokenization: Byte-pair Encoding

- **Cons:**
  - It is **greedy**: it tries to find the best pair at every iteration, which means it is not very efficient.
  - The generated tokens are dependent on the **number of iterations**, therefore we may have different tokens (and vocab size) and thus different representations based on how long we keep iterating.

Word Normalization

- **Word normalization** is the task of putting words/tokens in a **standard format**, choosing a single normal form for words with multiple forms.
  - Examples: *USA* and *US* or *uh-huh* and *uhhuh*.

- **Case folding** is another kind of normalization: mapping everything to **lower case**.
  - Example: *NLP*, *Nlp* and *nlp* are all represented identically.
  - Is done based on the application: e.g.: *US* is a country while *us* is a pronoun.

- **Lemmatization** is the task of determining that two words have the same **root**, despite their surface differences. For example:
  - the words *(am, are, is)* have the shared lemma *be.*
  - the words *(dinner, dinners)* both have the lemma *dinner.*

Word Normalization

- **How is lemmatization done?**
  - The most sophisticated methods for lemmatization involve complete **morphological parsing** of the word.

- **Morphology** is the study of the way words are **built up from smaller meaning-bearing units** called morphemes.

- Two broad classes of morphemes:
  - **stems**—the central morpheme of the word, supplying the **main meaning.**
  - **affixes**—*prefixes* and *suffixes*: **additional** letters that adhere to stems.

What is the difference between Stemming and Lemmatization?

Lemmatization algorithms can be complex. Stemming is simpler as it mainly consists of chopping off word-final stemming affixes → it is the naive version of morphological analysis.

Stemming →looks at the form of the word. Lemmatization →looks at the meaning of the word.

Word Normalization

Why word normalization??

- reduce text randomness
- bringing it closer to a predefined “standard”

→This helps us to reduce the amount of different information that the computer has to deal with, and therefore improves efficiency.

→The goal of normalization techniques like stemming and lemmatization is to reduce inflectional forms and sometimes derivationally related forms of a word to a common base form.

Word Normalization: The Porter Stemmer

- One of the most widely used stemming algorithms.

- The algorithm is based on **series of rewrite rules** run in series, as a cascade, in which the output of each pass is fed as input to the next pass.

- Sample rules:

ATIONAL	$\rightarrow$	ATE	(e.g., relational $\rightarrow$ relate)
ING	$\rightarrow$	ϵ	if stem contains vowel (e.g., motoring $\rightarrow$ motor)
SSES	$\rightarrow$	SS	(e.g., grasses $\rightarrow$ grass)

- Simple stemmers can be useful in cases where we need to **collapse** across different variants of the same lemma. Nonetheless, they **commit errors.**
  - Examples: organization → organ, policy → police

Sentence Segmentation

- It is another important step in text processing.

- The most useful cues for segmenting a text into sentences are **punctuation**, like periods, question marks, and exclamation points.

- Question marks and exclamation points are **relatively unambiguous** markers of sentence boundaries. Periods, on the other hand, are **more ambiguous:**
  - A marker of abbreviations like Mr. or Inc.
  - Even more complex case in which the period marked both an abbreviation and the sentence boundary.

- In general, sentence tokenization methods work by first deciding (based on rules or machine learning) whether a period is part of the word or is a sentence-boundary marker. An abbreviation dictionary can help.


Thank You