

# CS 463

# NATURAL LANGUAGE PROCESSING

---

Dr. Saleh Haridy


2023-2024

Week 2

# Regular Expressions, Text Normalization,



# Outline

- ▶ Regular expressions
    - Basic RegExp patterns, Disjunction, grouping and precedence
    - Example using RE python library
  - ▶ Words
    - Lemma, wordform, wordtype, token
  - ▶ Corpora
  - ▶ Text Normalization
    - Word Tokenization
    - Word Normalization, Lemmatization, stemming
  - ▶ Sentence segmentation
- 

# ELIZA chatbot in 1966

- ▶ 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.

- The dialogue above is from ELIZA, an early natural language processing system that could carry on a limited conversation with a user by developed at MIT in 1966.
- ELIZA is a surprisingly 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?**”.

<http://psych.fullerton.edu/mbirnbaum/psychmth.azile/101/>

# Regular Expressions (RE)

- ▶ A language for specifying **text search strings**.
- ▶ RE is an algebraic notation for characterizing a set of strings.
- ▶ Regular expressions can be used to **search** for a **string** in a text or **corpus** of texts such as find strings like \$199 or \$24.99 for extracting tables of prices from a document.
- ▶ the Unix command-line tool **grep** takes a regular expression and returns every line of the input document that matches the expression
- ▶ Python has a built-in library called **re**, which can be used for Regular expressions

[https://www.w3schools.com/python/python\\_regex.asp](https://www.w3schools.com/python/python_regex.asp)

# Basic Regular Expression Patterns

- ▶ The simplest kind of regular expression is a sequence of simple characters.

RE	Example Patterns Matched
/woodchucks/	“interesting links to <u>woodchucks</u> and lemurs”
/a/	“ <u>M</u> ary Ann stopped by Mona’s”
/!/	“You’ve left the burglar behind again!” said Nori

```
import re
txt = "interesting links to woodchucks"
x = re.search("woodchucks", txt)
print(x.group())
```

```
['woodchucks']
```

Python Online IDE:

<https://replit.com/languages/python3>

# Basic Regular Expression Patterns

- ▶ Regular expressions are case sensitive; lower case /s/ is distinct from upper case /S/
- ▶ This means that the pattern /woodchucks/ will not match the string Woodchucks. We can solve this problem with the use of the square braces [ ].
- ▶ The string of characters inside the braces specifies a disjunction (OR) of characters to match, /[wW]/ matches patterns containing either w or W.
- ▶ The regular expression /[1234567890]/ specifies any single digit.

RE	Match	Example Patterns
/[wW]oodchuck/	Woodchuck or woodchuck	<u>“Woodchuck”</u>
/[abc]/	‘a’, ‘b’, or ‘c’	“In uomini, in sold <u>a</u> ti”
/[1234567890]/	any digit	“plenty of <u>7</u> to 5”

# Regular Expression (range)

- ▶ In cases where there is a well-defined sequence associated with a set of characters, the brackets can be used with the dash (–) to specify range any one character in a range. 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
<code>/[A–Z]/</code>	an upper case letter	“we should call it ‘ <u>D</u> renched Blossoms’ ”
<code>/[a–z]/</code>	a lower case letter	“ <u>m</u> y beans were impatient to be hoed!”
<code>/[0–9]/</code>	a single digit	“Chapter <u>1</u> : Down the Rabbit Hole”



# Regular Expression (negation)

- ▶ 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
<code>/[^A-Z]/</code>	not an upper case letter	"O <u>y</u> fn pripetchik"
<code>/[^Ss]/</code>	neither 'S' nor 's'	" <u>I</u> have no exquisite reason for't"
<code>/[^.]/</code>	not a period	" <u>o</u> ur resident Djinn"
<code>/[e^]/</code>	either 'e' or '^'	"look up <u>^</u> now"
<code>/a^b/</code>	the pattern 'a^b'	"look up <u>a^</u> b now"

# Regular Expression (optionality)

- ▶ How can we talk about optional elements, like an optional `s` in `woodchuck` and `woodchucks`? We can't use the square brackets, because while they allow us to say “`s` or `S`”, they don't allow us to say “`s` or nothing”
- ▶ we use the question mark `/?/`, which means “the preceding character or nothing”,

RE	Match	Example Patterns Matched
<code>/woodchucks?/</code>	woodchuck or woodchucks	<u>“woodchuck”</u>
<code>/colou?r/</code>	color or colour	<u>“color”</u>

# Regular Expression (Kleene \*)

- ▶ These words consists of strings with letter b, followed by at least two a's, followed by an exclamation point.
- ▶ The set of operators that allows us to say things like “some Kleene \* number of as” are based on the asterisk or \*,
- ▶ The Kleene star means “**zero or more occurrences** of the immediately previous character or regular expression

baa!  
baaa!  
baaaa!  
baaaaa!

RE	Meaning
/a*/	Any string of zero or more a
/aa*/	one a followed by zero or more as
/[ab]*/	zero or more a's or b's
/[9-0][9-0]*/	An integer (a string of digits) is

# Regular Expression (Kleene +)

- ▶ Kleene +, means “one or more occurrences of the immediately preceding character or regular expression”.
- ▶ The expression `/[0-9]+/` is the normal way to specify “a sequence of digits”.
- ▶ One very important special character is the period (`/./`), a wildcard expression that matches any single character

RE	Match	Example Matches
<code>/beg.n/</code>	any character between <i>beg</i> and <i>n</i>	<u>begin</u> , <u>beg'n</u> , <u>begun</u>

# Regular Expression (anchors ^ \$)

- ▶ Anchors are special characters that anchor regular expressions to particular places in a string.
- ▶ the caret ^ has three uses: to match the start of a line, to indicate a negation inside of square brackets, and just to mean a caret

RE	Match
^	start of line
\b	word boundary
\B	non-word boundary

RE	Meaning
^The/	matches the word The only at the start of a line.
\.\$	match a . at the end of a line
/^The dog\.\$/	matches a line that contains only the phrase The dog.
/\bthe\b/	matches the word the but not the word other
/\b99\b	match the string 99 in There are 99 bottles of beer on

# Disjunction “|”, Grouping, and Precedence

- ▶ Suppose we need to search for texts about cats and dogs. We can't use `/[catdog]/`
- ▶ The disjunction operator (the pipe `|`) can be used. The pattern `/cat|dog/` matches either the string `cat` or the string `dog`.

Pattern	Matches
<code>groundhog woodchuck</code>	<code>woodchuck</code>
<code>yours mine</code>	<code>yours</code>
<code>a b c</code>	<code>= [abc]</code>
<code>[gG]roundhog [Ww]oodchuck</code>	<code>Woodchuck</code>

# Substitutions

- ▶ Substitution in Python

- ▶ *re.sub(pattern, repl, string, count=0, flags=0)*

pattern – the pattern which is to be searched and substituted

repl – the string with which the pattern is to be replaced

string – the name of the variable in which the pattern is stored

count – number of characters up to which substitution will be performed

flags – it is used to modify the meaning of the regex pattern

count and flags are optional arguments.

```
import re
sentence1 = "It is raining outside."
re.sub(r"raining", "sunny", sentence1)
```

It is sunny outside.

# Capture Groups

It is often useful to be able to refer to a particular subpart of the string matching the first pattern. For that, we can use "capture groups", a way of storing part of the pattern into a "register" so we can refer to it later in the substitution string.

- Say we want to put angles around all numbers:

*the 35 boxes* → *the <35> boxes*

- Use parens ( ) to "capture" a pattern into a numbered register (1, 2, 3...)
- Use \1 to refer to the contents of the register

s / ( [0-9] + ) / < \1 > /



# Capture groups: multiple registers

```
import re
mon = re.search(r'(\d{4})-(\d{2})-(\d{2})', '2018-09-01')
print(mon.groups())
print(mon.group())
print(mon.group(1))
print(mon.group(2))
print(mon.group(3))
```

('01','09','2018')

01-09-2018

2018

09

01

```
# substitute with group reference
import re
date = r'2018-09-01'
re.sub(r'(\d{4})-(\d{2})-(\d{2})', r'\2/\3/\1/', date)
```

'/09/01/2018'

# But suppose we don't want to capture?

Parentheses have a double function: grouping terms, and capturing

Non-capturing groups: add a ?: after paren:

- ▶ `/(?:some|a few) (people|cats) like some \1/`
- ▶ `matches`
  - `some cats like some cats`
- ▶ `but not`
  - `some cats like some some`

```
sen = 'some cats like some cats'
mon = re.search(r'(?:some|a few) (people|cats) like some \1', sen)
mon.groups()
```

`('cats',)`

# More Operators

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

**Figure 2.8** Aliases for common sets of characters.

RE	Match
*	zero or more occurrences of the previous char or expression
+	one or more occurrences of the previous char or expression
?	exactly zero or one occurrence of the previous char or expression
{n}	n occurrences of the previous char or expression
{n,m}	from n to m occurrences of the previous char or expression
{n,}	at least n occurrences of the previous char or expression
{,m}	up to m occurrences of the previous char or expression

**Figure 2.9** Regular expression operators for counting.

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	

**Figure 2.10** Some characters that need to be backslashed.

# Example

- ▶ Assume we want to find all instances of the word “the” in a text.

the

- ▶ This will miss the word (The)

- capitalized examples

[tT]he

- But it Incorrectly returns other or theology

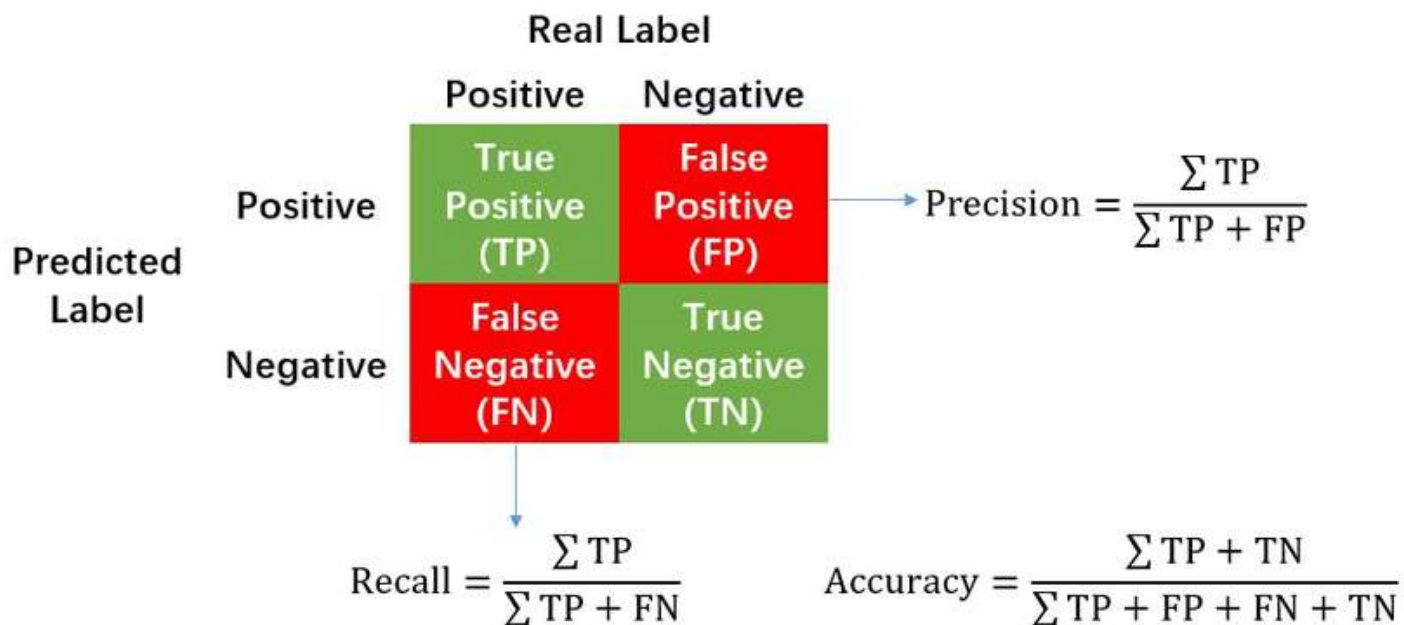
[^a-zA-Z][tT]he[^a-zA-Z]

# Errors

- ▶ The process we just went through in previous example was based on **fixing two kinds of errors**:
  1. Matching strings that we should not have matched (**there, then, other**)  
**False positives (Type I errors)**
  2. Not matching things that we should have matched (The)  
**False negatives (Type II errors)**

# Errors cont.

- ▶ In NLP we are always dealing with these kinds of errors.
- ▶ Reducing the error rate for an application often involves two antagonistic efforts:
  - **Increasing precision** (minimizing false positives)
  - **Increasing recall** (minimizing false negatives).



# Simple Application: ELIZA

- ▶ Early NLP system that imitated a Rogerian psychotherapist Joseph Weizenbaum, 1966.
- ▶ Uses pattern matching to match, e.g.,:
  - “I need X”and translates them into, e.g.
  - “What would it mean to you if you got X?”

# Simple Application: ELIZA

Men are all alike.

IN WHAT WAY

They're always bugging us about something or other. CAN YOU THINK OF A SPECIFIC EXAMPLE

Well, my boyfriend made me come here.

YOUR BOYFRIEND MADE YOU COME HERE

He says I'm depressed much of the time.

I AM SORRY TO HEAR YOU ARE DEPRESSED





# How ELIZA (chatbot) works

- ▶ `s/. * I'M (depressed|sad) . */I AM SORRY TO HEAR YOU ARE \1 /`
- ▶ `s/. * I AM (depressed|sad) . */WHY DO YOU THINK YOU ARE \1 /`
- ▶ `s/. * all . */IN WHAT WAY?/`
- ▶ `s/. * always . */CAN YOU THINK OF A SPECIFIC EXAMPLE?/`

```
Q=r"Hi, I'M sad"
```

```
A=re.sub(r". * I'M (depressed|sad)",r'I AM SORRY TO HEAR YOU ARE \1',Q)
```

```
'I AM SORRY TO HEAR YOU ARE sad'
```

# Regular expressions Summary

- ▶ Regular expressions play a surprisingly large role
  - Sophisticated sequences of regular expressions are often the first model for any text processing text
- ▶ For hard tasks, we use machine learning classifiers
  - But regular expressions are still used for pre-processing, or as features in the classifiers
  - Can be very useful in capturing generalizations

# How many words in a sentence?

- ▶ "I do uh main– mainly business data processing"
  - Fragments, filled pauses
- ▶ "Seuss's **cat** in the hat is different from other **cats**!"
  - **Lemma**: Two words are the same lemma if they have the same stem, the same part of speech, the same sense
    - **cat** and **cats** = same lemma
  - **Wordform**: the exact surface form of the word, with all its inflections or endings
    - **cat** and **cats** = different wordforms

# How many words in a sentence?

they lay back on the San Francisco grass and looked at the stars and their

- It depends how you count. We could count word types, the number of unique words that occur in the sentence
- By that count we only count "the" once, even though it appears twice.
- ▶ **Type**: an element of the vocabulary.
- ▶ **Token**: an instance of that type in running text.
- ▶ How many?
  - 15 tokens (or 14)
  - 13 types (or 12) (or 11?)

# How many words in a **corpus**?

$N$  = number of tokens

$V$  = vocabulary = set of types,  $|V|$  is size of vocabulary

Heaps Law = Herdan's Law =  $|V| = kN^\beta$  where often  $.67 < \beta < .75$   
i.e., vocabulary size grows with  $>$  square root of the number of word tokens

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

**Types** are the number of distinct words in a corpus;

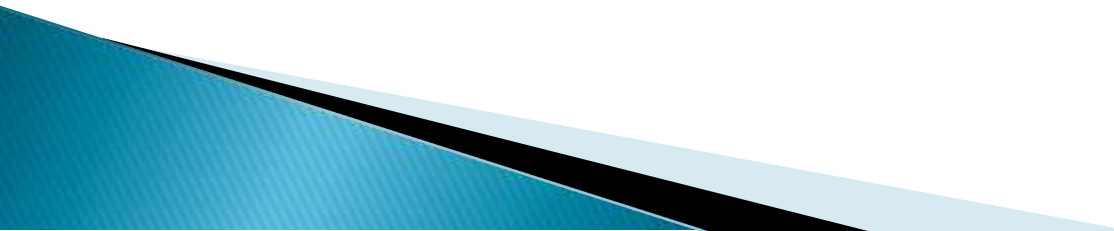
**Tokens** are the total number  $N$  of running words

# Corpora (text datasets)

A text corpus is a large body of text. Many corpora are designed to contain a careful balance of material in one or more genres.

Words don't appear out of nowhere!

A text is produced by

- a specific writer(s),
  - at a specific time,
  - in a specific variety,
  - of a specific language,
  - for a specific function.
- 

# Corpora vary along dimension like

- **Language:** 7097 languages in the world
- **Variety**, like African American Language varieties.
  - AAE Twitter posts might include forms like "*iont*" (*I don't*)
- **Code switching**, e.g., Spanish/English, Hindi/English:

S/E: Por primera vez veo a @username actually being hateful! It was beautiful:)

*[For the first time I get to see @username actually being hateful! it was beautiful:)]*

H/E: dost tha or ra- hega ... dont worry ... but dherya rakhe

*["he was and will remain a friend ... don't worry ... but have faith"]*

- **Genre:** newswire, fiction, scientific articles, Wikipedia
- **Author Demographics:** writer's age, gender, ethnicity, SES

# Corpus datasheets

## **Motivation:**

- Why was the corpus collected?
- By whom?
- Who funded it?

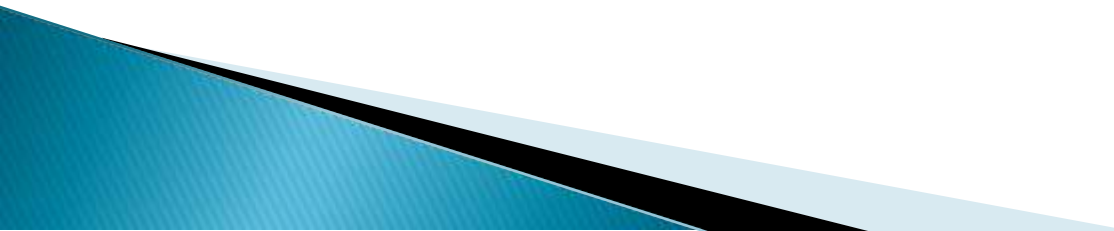
**Situation:** In what situation was the text written?

**Collection process:** If it is a subsample how was it sampled? Was there consent? Pre-processing?

- ▶ + **Annotation process, language variety, demographics, etc.**



# Text Normalization

- ▶ Every NLP task requires text normalization:
    1. Tokenizing (segmenting) words
    2. Normalizing word formats
    3. Segmenting sentences
- 

# Space-based tokenization

- ▶ A very simple way to tokenize
  - For languages that use space characters between words
    - Arabic, Cyrillic, Greek, Latin, etc., based writing systems
  - Segment off a token between instances of spaces
- ▶ Unix tools for space-based tokenization
  - The "tr" command in Unix
  - Given a text file, output the word tokens and their frequencies
- ▶ Python tools for space-based tokenization
- ▶ *split* method for string
- ▶ *findall* method in re library
- ▶ ...

# Simple Tokenization in Python

```
text = """ Tokenization is a common task a data scientist comes  
across when working with text data. It consists of splitting an  
entire text into small units, also known as tokens. Most Natural  
Language Processing (NLP) projects have tokenization as the first  
step because it's the foundation for developing good models and  
helps better understand the text we have."""
```

```
# use split  
text.split()
```

```
#tokenize the text with regular expressions  
token=re.findall('\w+', text)  
token[:10]
```

```
['Tokenization', 'is', 'a',  
'common',  
'task', 'a',  
'data',  
'scientist',  
'comes',  
'across',  
'when',  
'working',
```

# Simple Tokenization in Python

```
# convert to lowercase
for word in token:
    words.append(word.lower())
words[:10]
```

```
# download stopwords module
import nltk
nltk.download("stopwords")
sw=nltk.corpus.stopwords.words('english')
sw[:5]
```

```
# get the list without stop words
words_ne=[]
for word in words:
    if word not in sw:
        words_ne.append(word)
words_ne=sorted(words_ne)
words_ne[:5]
```

```
['across', 'also', 'better', 'comes', 'common']
```

# Issues in Tokenization

- ▶ Can't just blindly remove punctuation:
  - [m.p.h.](#), [Ph.D.](#), [AT&T](#), [cap'n](#)
  - prices ([\\$45.55](#))
  - dates ([01/02/06](#))
  - URLs (<http://www.stanford.edu>)
  - hashtags ([#nlproc](#))
  - email addresses ([someone@cs.colorado.edu](mailto:someone@cs.colorado.edu))
- ▶ Clitic: a word that doesn't stand on its own
  - "are" in [we're](#), French "je" in [j'ai](#), "le" in [l'honneur](#)
- ▶ When should multiword expressions (MWE) be words?
  - [New York](#), [Kingdom of Saud Arabia](#)

# Tokenization in NLTK

Bird, Loper and Klein (2009), *Natural Language Processing with Python*. O'Reilly

```
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', '...']
```

The special (?x) "verbose flag" tells Python to strip out the embedded whitespace and comments.

# Another option for text tokenization

Instead of

- white-space segmentation
- single-character segmentation

Use the data to tell us how to tokenize.

**Subword tokenization** (because tokens can be parts of words as well as whole words)



# Subword tokenization

- ▶ Three common algorithms:
  - **Byte–Pair Encoding (BPE)** (Sennrich et al., 2016)
  - **Unigram language modeling tokenization** (Kudo, 2018)
  - **WordPiece** (Schuster and Nakajima, 2012)
- ▶ All have 2 parts:
  - A token **learner** that takes a raw training corpus and induces a vocabulary (a set of tokens).
  - A token **segmenter** that takes a raw test sentence and tokenizes it according to that vocabulary



# Byte Pair Encoding (BPE) token learner

Let vocabulary be the set of all individual characters

$$= \{A, B, C, D, \dots, a, b, c, d, \dots\}$$

- ▶ Repeat:
  - Choose the two symbols that are most frequently adjacent in the training corpus (say 'A', 'B')
  - Add a new merged symbol 'AB' to the vocabulary
  - Replace every adjacent 'A' 'B' in the 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  $C$

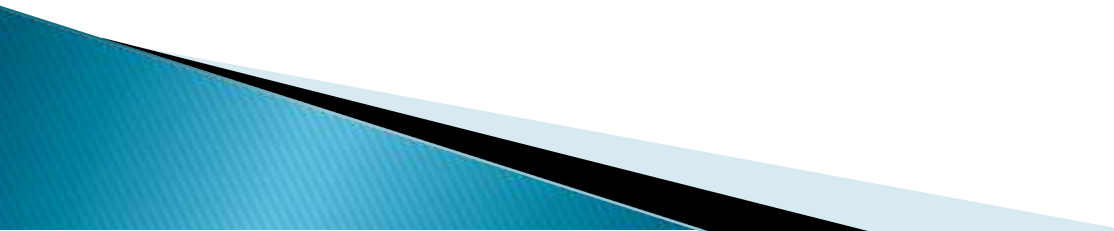
$t_{NEW} \leftarrow t_L + t_R$                               # make new token by concatenating

$V \leftarrow V + t_{NEW}$                               # 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 space-separated tokens.
  - ▶ So we commonly first add a special end-of-word symbol '\_\_\_' before space in training corpus
  - ▶ Next, separate into letters.
- 

# BPE token learner

Original (very fascinating😊) corpus:

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

Add end-of-word tokens, resulting in this vocabulary:

**vocabulary**

—, d, e, i, l, n, o, r, s, t, 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

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

# BPE

## 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, e r

Merge **er \_** to **er\_**

## 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, e r, e r\_

# 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

# 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—



# BPE token **segmenter** algorithm

On the test data, run each merge learned from the training data:

- Greedily
- In the order we learned them
- (test frequencies don't play a role)

So: merge every **e r** to **er**, then merge **er \_** to **er\_**, etc.

Test = ' **newer-**      **low er-** '

▶ Result:

- 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\_"

# Properties of BPE tokens

Usually include frequent words

And frequent subwords

- Which are often morphemes like *-est* or *-er*

A **morpheme** is the smallest meaning-bearing unit of a language

- *unlikeliest* has 3 morphemes *un-*, *likely*, and *-est*

Implementation of BPE

<https://huggingface.co/course/chapter6/5?fw=pt>

# Word Normalization

- ▶ Putting words/tokens in a standard format
  - U.S.A. or USA or US → USA
  - uhhuh or uh-huh
  - Fed or fed
  - am, is, be, are → be
  - KSA, Saudi Arabia, Kingdom of Saudi Arabia > Saudia
- For information retrieval (IR) or information extraction about the US, we might want to see information from documents whether they mention the US or the USA.

# Case folding

- ▶ Applications like IR: reduce all letters to lower case
  - Since users tend to use lower case
- ▶ For sentiment analysis, MT, Information extraction
  - Case is helpful (*US* versus *us* is important)
- ▶ For many natural language processing situations we also want two morphologically different forms of a word to behave similarly

# Lemmatization

- Lemmatization is the task of determining that two words have the same root, despite their surface differences.
- ▶ Represent all words as their **lemma**, their shared root
- ▶ = dictionary headword form:
  - *am, are, is* → *be*
  - *car, cars, car's, cars'* → *car*
  - Arabic: كُتِبَ - يَكْتُبُ - كَتَبُوا - مَكْتُبُهُ - كِتَابٌ - كَاتِبٌ
  - *He is reading detective stories*
    - → *He be read detective story*

# Lemmatization is done by Morphological Parsing

## ▶ Morphemes:

- The small meaningful units that make up words consists of two classes:
- **Stems**: The core meaning-bearing units
- **Affixes**: Parts that adhere to stems, often with grammatical functions

## ▶ Morphological Parsers:

- Parse *cats* into two morphemes *cat* and *s*
- In Arabic : وسـيـكـتـبـون (ون - يـكـتـب - وس)
  - Stem: يـكـتـب
  - Lemma: كـتـب

Lemmatization algorithms can be complex and we use stemming.

# Stemming

- ▶ Reduce terms to stems, chopping off affixes crudely

This was not the map we found in Billy Bones's chest, but an accurate copy, complete in all things-names and heights and soundings-with the single exception of the red crosses and the written notes.



Thi wa not the map we found in Billi Bone s chest but an accur copi complet in all thing name and height and sound with the singl except of the red cross and the written note .

# Porter Stemmer

- ▶ Based on a series of rewrite rules run in series
  - A cascade, in which output of each pass fed to next pass
- ▶ Some sample rules:

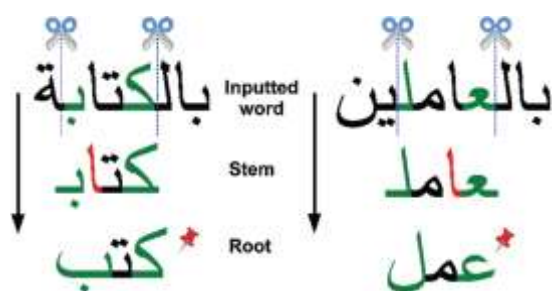
ATIONAL → ATE (e.g., relational → relate)

ING →  $\epsilon$  if stem contains vowel (e.g., motoring → motor)

SSES → SS (e.g., grasses → grass)



Dealing with complex morphology is necessary for Arabic languages



Prefix	Attached word	Example
ين	singular female	تلعين
ان	male dual	يلعبان
و	plural male,	ينمو
ه	missing singular	ضربته
ك	addresser singular	ضربك
ا	male dual	أكلا
ي	singular female	أكلتي
ن	plural	أكلن
ت	singular female	أكلت
ات	female plural	لاعبات
ون	absent male plural	يلعبون
وا	absent male plural	أكلوا
تم	addresser male plural	أكلتم
هم	absent male plural	ضربهم
كم	addresser male plural	ضربكم

# Porter Stemmer in NLTK

```
nltk.download('book')
```

```
from nltk import word_tokenize
raw = """DENNIS: Listen, strange women lying in
ponds distributing swords is no basis for a
system of government. Supreme executive power
derives from
a mandate from the masses, not from some
farfetched aquatic ceremony."""
tokens = word_tokenize(raw)
porter = nltk.PorterStemmer()
[porters.stem(t) for t in tokens]
```

```
['denni',  
'.',  
'listen',  
'.',  
'strang',  
'women',  
'lie', 'in',  
'pond',  
'distribut']
```

# Sentence Segmentation

- ▶ The most useful cues for segmenting a text into sentences are punctuation, like periods, question marks, and exclamation points.
- ▶ .,!, ? mostly unambiguous
- ▶ but **period** “.” is very ambiguous (U.S.A), 12.34
  - Between sentence boundary marker
  - Abbreviations like Inc. or Dr.
  - Numbers like .02% or 4.3
- ▶ Common algorithm: Tokenize first: use rules or ML to classify a period as either (a) part of the word or (b) a sentence–boundary.
  - An abbreviation dictionary can help
- ▶ Sentence segmentation can then often be done by rules based on this tokenization.

# Sentence Segmentation using NLTK

```
text = """ My name is Ahmed. I'm thirty 30.0 years old.  
I live in Riyadh, K.S.A """  
sents = nltk.sent_tokenize(text)  
sents
```

```
[' My name is Ahmed.', "I'm thirty 30.0 years old.", 'I  
live in Ryiad, K.S.A']
```