

# Basic Text Processing

By J. H. Wang

Feb. 23, 2023

# Outline

- Words
  - Regular expressions
  - Pre-processing
- Language models
  - N-grams
- Syntax
  - POS tagging
  - Parsing

# Basic Text Processing

## Regular Expressions

# Regular expressions

- A formal language for specifying text strings
- How can we search for any of these?
  - woodchuck
  - woodchucks
  - Woodchuck
  - Woodchucks



# Regular Expressions: Disjunctions

- Letters inside square brackets []

Pattern	Matches
<code>[wW]oodchuck</code>	Woodchuck, woodchuck
<code>[1234567890]</code>	Any digit

- Ranges `[A-Z]`

Pattern	Matches	
<code>[A-Z]</code>	An upper case letter	<u>D</u> renched Blossoms
<code>[a-z]</code>	A lower case letter	<u>m</u> y beans were impatient
<code>[0-9]</code>	A single digit	Chapter <u>1</u> : Down the Rabbit Hole

# Regular Expressions: Negation in Disjunction

- Negations `[^Ss]`
  - Carat means negation only when first in []

Pattern	Matches	
<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"
<code>[^e^]</code>	Neither e nor ^	Look h <u>e</u> re
<code>a^b</code>	The pattern a carat b	Look up <u>a^b</u> now

# Regular Expressions: More Disjunction

- Woodchucks is another name for groundhog!
- The pipe | for disjunction

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



# Regular Expressions: ? \* + .

Pattern	Matches	
<code>colou?r</code>	Optional previous char	<u>color</u> <u>colour</u>
<code>oo*h!</code>	0 or more of previous char	<u>oh!</u> <u>ooh!</u> <u>oooh!</u> <u>ooooh!</u>
<code>o+h!</code>	1 or more of previous char	<u>oh!</u> <u>ooh!</u> <u>oooh!</u> <u>ooooh!</u>
<code>baa+</code>		<u>baa</u> <u>baaa</u> <u>baaaa</u> <u>baaaaa</u>
<code>beg.n</code>		<u>begin</u> <u>begun</u> <u>begun</u> <u>beg3n</u>



Stephen C Kleene

Kleene \*, Kleene +



# Regular Expressions: Anchors <sup>^</sup> <sup>\$</sup>

- <sup>^</sup>: start of line
- <sup>\$</sup>: end of line

Pattern	Matches
<sup>^</sup> [A-Z]	<u>P</u> alo Alto
<sup>^</sup> [ ^A-Za-z]	<u>1</u> <u>"Hello"</u>
\. <sup>\$</sup>	The end <u>.</u>
. <sup>\$</sup>	The end <u>?</u> The end <u>!</u>

# Example

- Find me all instances of the word “the” in a text

the

Misses capitalized examples

[tT]he

Incorrectly returns other or theology

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

# Errors

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

## 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 accuracy or precision (minimizing false positives)
  - Increasing coverage or recall (minimizing false negatives).

# Summary

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

# Basic Text Processing

More Regular Expressions:  
Substitutions and ELIZA

# Substitutions

- Substitution in Python and UNIX commands:
- `s/regex1/pattern/`
- **e.g.:**
- `s/colour/color/`

# Capture Groups

- 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

- `/the (.*)er they (.*) , the \1er we \2/`
- Matches
- *the faster they ran, the faster we ran*
- *But not*
- *the faster they ran, the faster we ate*

# 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`

# Lookahead assertions

- `(?= pattern)` is true if pattern matches, but is **zero-width; doesn't advance character pointer**
- `(?! pattern)` true if a pattern does not match
- How to match, at the beginning of a line, any single word that doesn't start with "Volcano":
- `/^(?!Volcano) [A-Za-z]+/`

# 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 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?/

# Basic Text Processing

## Words and Corpora

# 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**: same stem, part of speech, rough word sense
    - **cat** and **cats** = same lemma
  - **Wordform**: the full inflected surface form
    - **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

- **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

# Corpora

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

[Gebru et al \(2020\)](#), [Bender and Friedman \(2018\)](#)

## 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.**

# Basic Text Processing

## Word tokenization

# 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
  - Inspired by Ken Church's UNIX for Poets
  - Given a text file, output the word tokens and their frequencies



# Simple Tokenization in UNIX

- (Inspired by Ken Church's UNIX for Poets.)
- Given a text file, output the word tokens and their frequencies

```
tr -sc 'A-Za-z' '\n' < shakes.txt  
  | sort  
  | uniq -c
```

Change all non-alpha to newlines

Sort in alphabetical order

Merge and count each type

```
1945 A  
 72 AARON      25 Aaron  
19 ABBESS      6 Abate  
 5 ABBOT       1 Abates  
          5 Abbess  
... ..      6 Abbey  
          3 Abbot  
... ..
```

# The first step: tokenizing

```
tr -sc 'A-Za-z' '\n' < shakes.txt | head
```

THE

SONNETS

by

William

Shakespeare

From

fairest

creatures

We

...

# The second step: sorting

```
tr -sc 'A-Za-z' '\n' < shakes.txt | sort | head
```

A

A

A

A

A

A

A

A

A

...

# More counting

- Merging upper and lower case

```
tr 'A-Z' 'a-z' < shakes.txt | tr -sc 'A-Za-z' '\n' | sort | uniq -c
```

- Sorting the counts

```
tr 'A-Z' 'a-z' < shakes.txt | tr -sc 'A-Za-z' '\n' | sort | uniq -c | sort -n -r
```

```
23243 the
22225 i
18618 and
16339 to
15687 of
12780 a
12163 you
10839 my
10005 in
8954 d
```

What happened here?

# 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](#), [rock 'n' roll](#)

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

# Tokenization in languages without spaces

Many languages (like Chinese, Japanese, Thai) don't use spaces to separate words!

How do we decide where the token boundaries should be?

# Word tokenization in Chinese

- Chinese words are composed of characters called "**hanzi**" (or sometimes just "**zi**")
- Each one represents a meaning unit called a morpheme
- Each word has on average 2.4 of them
- But deciding what counts as a word is complex and not agreed upon



# How to do word tokenization in Chinese?

- 姚明進入總決賽 “Yao Ming reaches the finals”

# How to do word tokenization in Chinese?

- 姚明進入總決賽 “Yao Ming reaches the finals”

- 3 words?

- 姚明      進入      總決賽

- YaoMing reaches finals

# How to do word tokenization in Chinese?

- 姚明進入總決賽 “Yao Ming reaches the finals”

- 3 words?

- 姚明 進入 總決賽

- YaoMing reaches finals

- 5 words?

- 姚 明 進入 總 決賽

- Yao Ming reaches overall finals

# How to do word tokenization in Chinese?

- 姚明進入總決賽 “Yao Ming reaches the finals”

- 3 words?

- 姚明 進入 總決賽

- YaoMing reaches finals

- 5 words?

- 姚 明 進入 總 決賽

- Yao Ming reaches overall finals

- 7 characters? (don't use words at all):

- 姚 明 進 入 總 決 賽

- Yao Ming enter enter overall decision game

# Word tokenization / segmentation

So in Chinese it's common to just treat each character (zi) as a token

- So the **segmentation** step is very simple

In other languages (like Thai and Japanese), more complex word segmentation is required

- The standard algorithms are neural sequence models trained by supervised machine learning

# Basic Text Processing

## Byte Pair Encoding

# 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,..., a, b, c, d....}

- 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) Addendum

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 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 \_

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, er

## vocabulary

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

# 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.

- 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*

# Basic Text Processing

## Word Normalization and Other Issues

# Word Normalization

- Putting words/tokens in a standard format
  - U.S.A. or USA
  - uhhuh or uh-huh
  - Fed or fed
  - am, is, be, are

# Case folding

- Applications like IR: reduce all letters to lower case
  - Since users tend to use lower case
  - Possible exception: upper case in mid-sentence?
    - e.g., ***General Motors***
    - ***Fed*** vs. ***fed***
    - ***SAIL*** vs. ***sail***
- For sentiment analysis, MT, Information extraction
  - Case is helpful (***US*** versus ***us*** is important)

# Lemmatization

Represent all words as their lemma, their shared root  
= dictionary headword form:

- *am, are, is* → *be*
- *car, cars, car's, cars'* → *car*
- Spanish *quiero* ('I want'), *quieres* ('you want')  
→ *querer* 'want'
- *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
- **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*
- Parse Spanish *amaren* ('if in the future they would love') into morpheme *amar* 'to love', and the morphological features *3PL* and *future subjunctive*

# 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)

SSSES → SS (e.g., grasses → grass)

# Dealing with complex morphology is necessary for many languages

- e.g., the Turkish word:
- **Uygarlastiramadiklarimizdanmissinizcasina**
- `(behaving) as if you are among those whom we could not civilize'
- **Uygar** `civilized' + **las** `become'
  - + **tir** `cause' + **ama** `not able'
  - + **dik** `past' + **lar** `plural'
  - + **imiz** `p1pl' + **dan** `abl'
  - + **mis** `past' + **siniz** `2pl' + **casina** `as if'

# Sentence Segmentation

!, ? mostly unambiguous but **period** “.” is very ambiguous

- Sentence boundary
- 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.

# Thanks for Your Attention!