Natural Language Processing

AI 3216/UG, AI 5203/PG

Instructor: Nidhi Goyal

Week-2

2.1 Regular Expressions

2.2 Text Normalization

Mahindra™ University
Global Thinkers. Engaged Leaders.

Acknowledgments

These slides are adapted from the book:

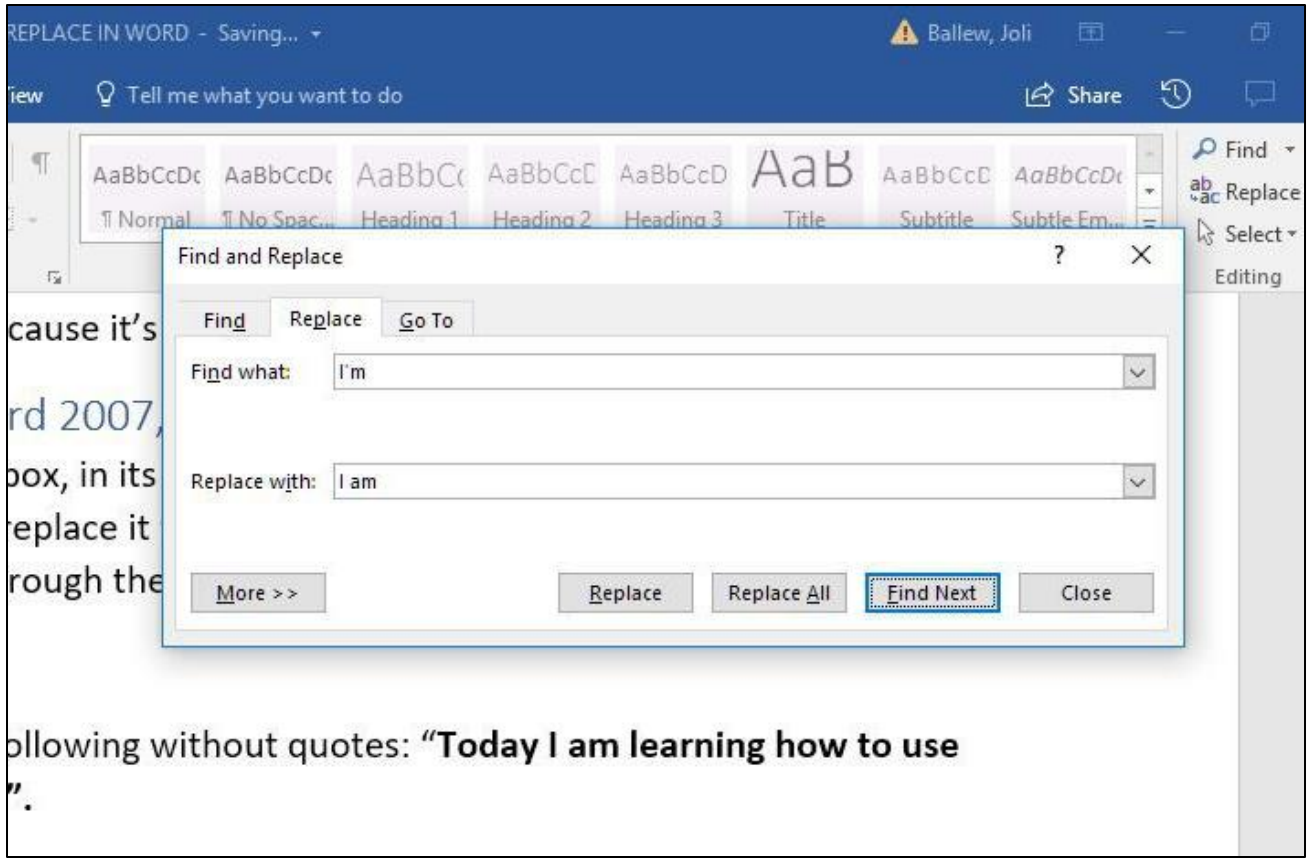
"SPEECH and LANGUAGE PROCESSING: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition"

And

Inspired from standard materials, presentations and resources provided online by verified scholars.

Regular Expressions

- Regular expressions, are sequences of characters that define a search pattern.
- They are used for matching and manipulating text strings based on patterns.



Where to use Regex?

- Data pre-processing
- Rule-based information mining systems
- Pattern matching
- Text feature engineering
- Web scraping
- Data extraction

many more.....

Why?

- Lot of unstructured data
- 1st step is pre-processing
  - Ways to do text pre-processing
    - Regex is one of the tool

Regular Expressions

- Disjunction
  - Negation
  - Pipe |
  - Special characters ? * + .
  - Anchors ^ $

Regular Expressions: Disjunctions

Letters inside square brackets []

Pattern	Matches
`[wW]oodchuck`	Woodchuck, woodchuck
`[1234567890]`	Any digit

Ranges

Pattern	Matches	
`[A-Z]`	An upper case letter	`Drenched Blossoms`
`[a-z]`	A lower case letter	`my beans were impatient`
`[0-9]`	A single digit	`Chapter 1: Down the Rabbit Hole`

Slide Reference: 2_TextProc_Mar_25_2021.pdf (stanford.edu)

Regular Expressions: Negation in Disjunction

Negations [^Ss]

- Carat means negation only when first in []

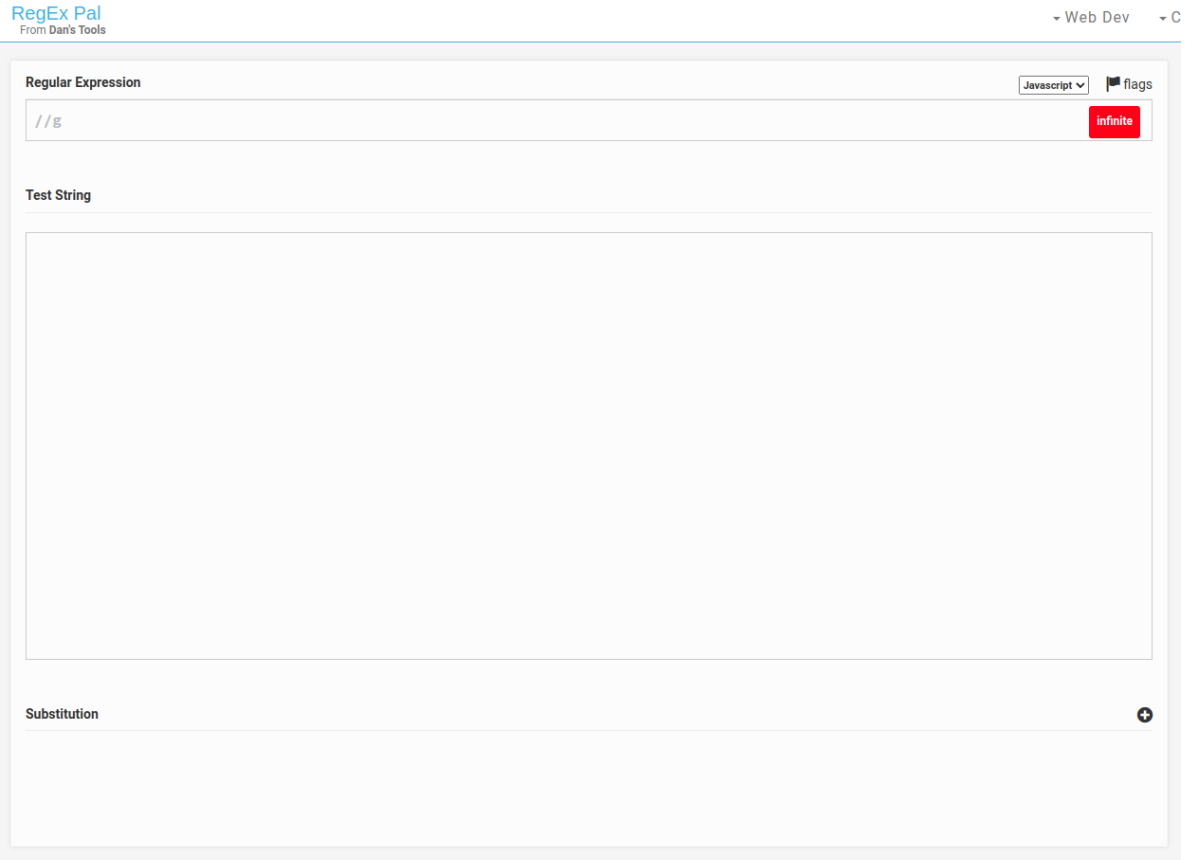
Pattern	Matches	
[^A-Z]	Not an upper case letter	Oyfn pripetchik
[^Ss]	Neither 'S' nor 's'	I have no exquisite reason"
[^e^]	Neither e nor ^	Look here
a^b	The pattern a carat b	Look up a^b now

Slide Reference: 2_TextProc_Mar_25_2021.pdf (stanford.edu)

Try Demo tool!!!

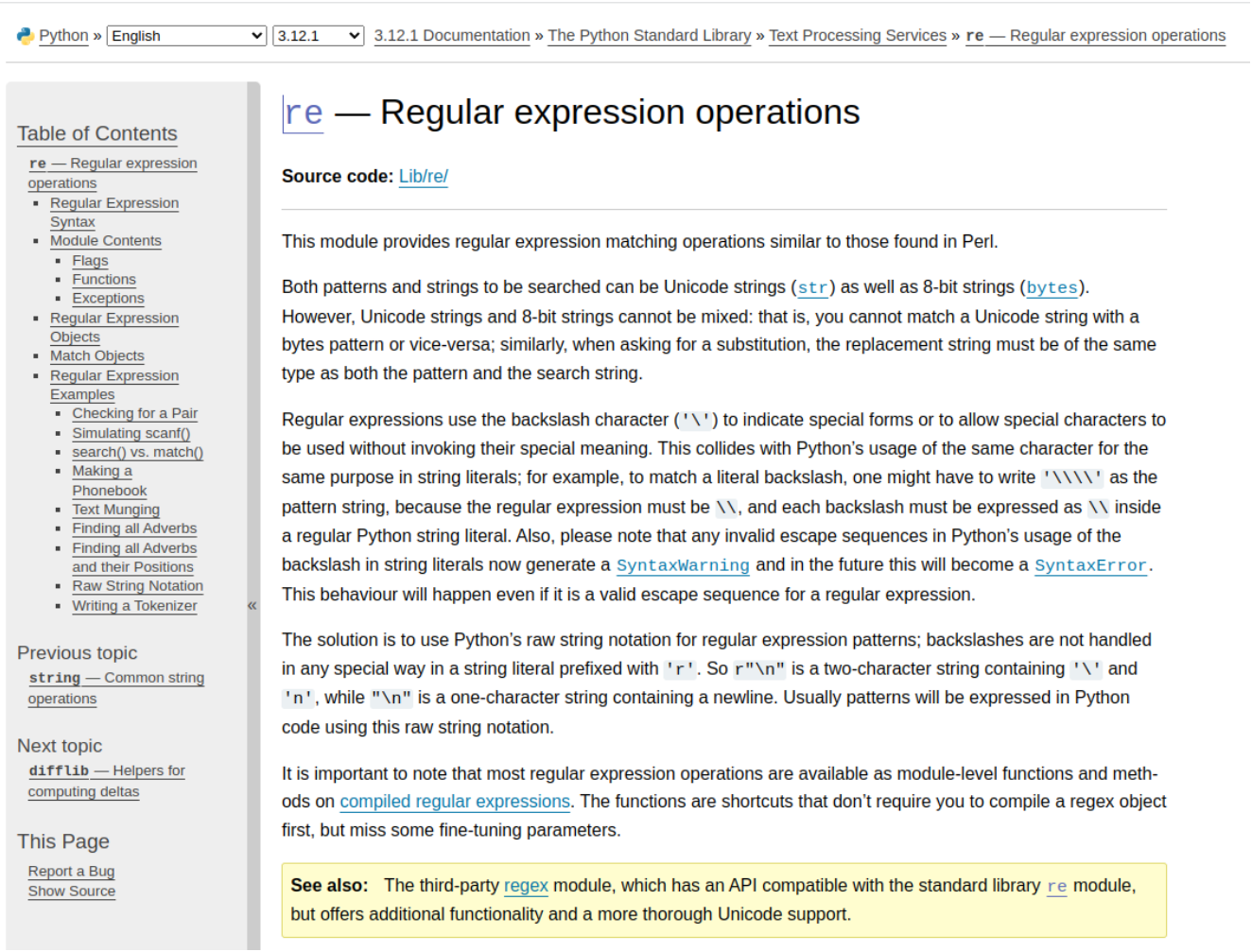
Match the patterns such as

- [Ww]
- [A-Z]
- [a-z]
- [A-Za-z]



[Regex Tester - Javascript, PCRE, PHP (regexpal.com)]

Regex documentation/Python



Python » English 3.12.1 3.12.1 Documentation » The Python Standard Library » Text Processing Services » re — Regular expression operations

Table of Contents

re — Regular expression operations
- Regular Expression Syntax
- Module Contents
  - Flags
  - Functions
  - Exceptions
- Regular Expression Objects
- Match Objects
- Regular Expression Examples
  - Checking for a Pair
  - Simulating scanf()
  - search() vs. match()
  - Making a Phonebook
  - Text Munging
  - Finding all Adverbs
  - Finding all Adverbs and their Positions
  - Raw String Notation
  - Writing a Tokenizer

Previous topic

string — Common string operations

Next topic

difflib — Helpers for computing deltas

This Page

Report a Bug
Show Source

re — Regular expression operations

Source code: Lib/re/

This module provides regular expression matching operations similar to those found in Perl.

Both patterns and strings to be searched can be Unicode strings (`str`) as well as 8-bit strings (`bytes`). However, Unicode strings and 8-bit strings cannot be mixed: that is, you cannot match a Unicode string with a bytes pattern or vice-versa; similarly, when asking for a substitution, the replacement string must be of the same type as both the pattern and the search string.

Regular expressions use the backslash character (`'\'`) to indicate special forms or to allow special characters to be used without invoking their special meaning. This collides with Python's usage of the same character for the same purpose in string literals; for example, to match a literal backslash, one might have to write `'\\\\'` as the pattern string, because the regular expression must be `\\`, and each backslash must be expressed as `\\` inside a regular Python string literal. Also, please note that any invalid escape sequences in Python's usage of the backslash in string literals now generate a `SyntaxWarning` and in the future this will become a `SyntaxError`. This behaviour will happen even if it is a valid escape sequence for a regular expression.

The solution is to use Python's raw string notation for regular expression patterns; backslashes are not handled in any special way in a string literal prefixed with `'r'`. So `r"\n"` is a two-character string containing `'\'` and `'n'`, while `"\n"` is a one-character string containing a newline. Usually patterns will be expressed in Python code using this raw string notation.

It is important to note that most regular expression operations are available as module-level functions and methods on compiled regular expressions. The functions are shortcuts that don't require you to compile a regex object first, but miss some fine-tuning parameters.

See also: The third-party regex module, which has an API compatible with the standard library `re` module, but offers additional functionality and a more thorough Unicode support.

https://docs.python.org/3/library/re.html#module-re

Regular Expressions: More Disjunction (pipe |)

Pattern	Matches
`groundhog|woodchuck`	`woodchuck`
`yours|mine`	`yours`
`a|b|c`	= [abc]
`[gG]roundhog|[Ww]oodchuck`	`Woodchuck`

Slide Reference: 2_TextProc_Mar_25_2021.pdf (stanford.edu)

Regular Expressions: ? , Kleen operators(*+),.

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

Slide Reference: 2_TextProc_Mar_25_2021.pdf (stanford.edu)

Regular Expressions: Anchors ^ $

Pattern	Matches
^[A-Z]	Palo Alto
^[^A-Za-z]	1 "Hello"
\.$	The end.
.$	The end? The end!

Slide Reference: 2_TextProc_Mar_25_2021.pdf (stanford.edu)

Some examples

Find me all instances of the word "the" in a text:

- the Misses capitalized examples
- [tT]he Incorrectly returns other or theology

Sol: `[^a-zA-Z][tT]he[^a-zA-Z]`

Slide Reference: 2_TextProc_Mar_25_2021.pdf (stanford.edu)

Natural Language Processing

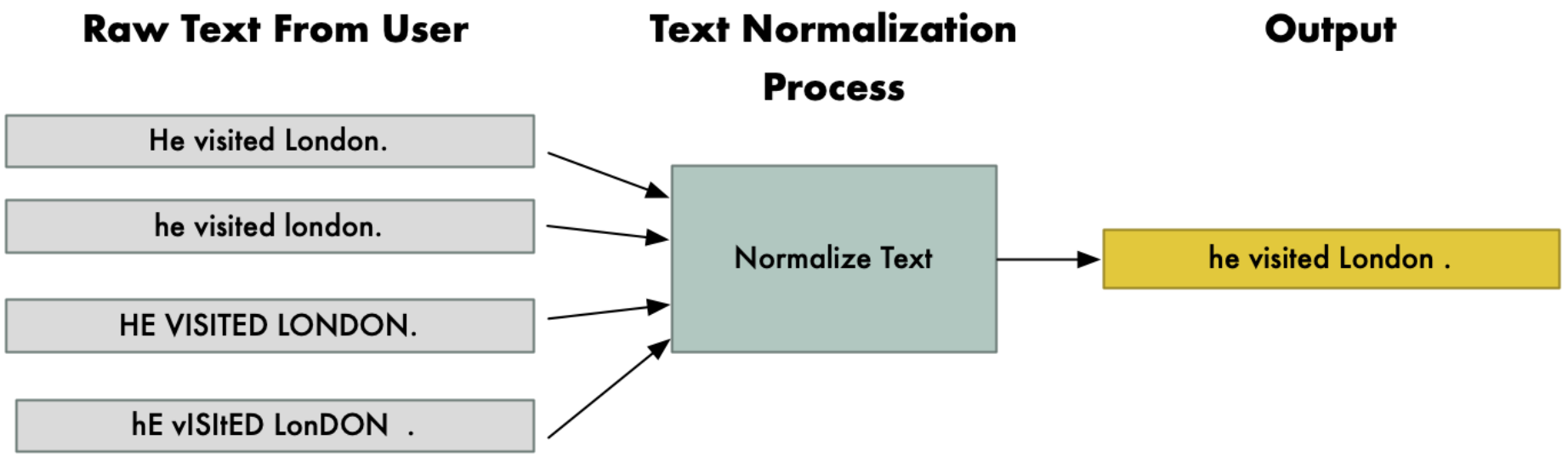
AI 3216/UG, AI 5203/PG

Week-2

2.1 Regular Expressions

2.2 Text Normalization

Text Normalization



Slide Reference: 2_TextProc_Mar_25_2021.pdf (stanford.edu)

Real world issues in text that need Normalization

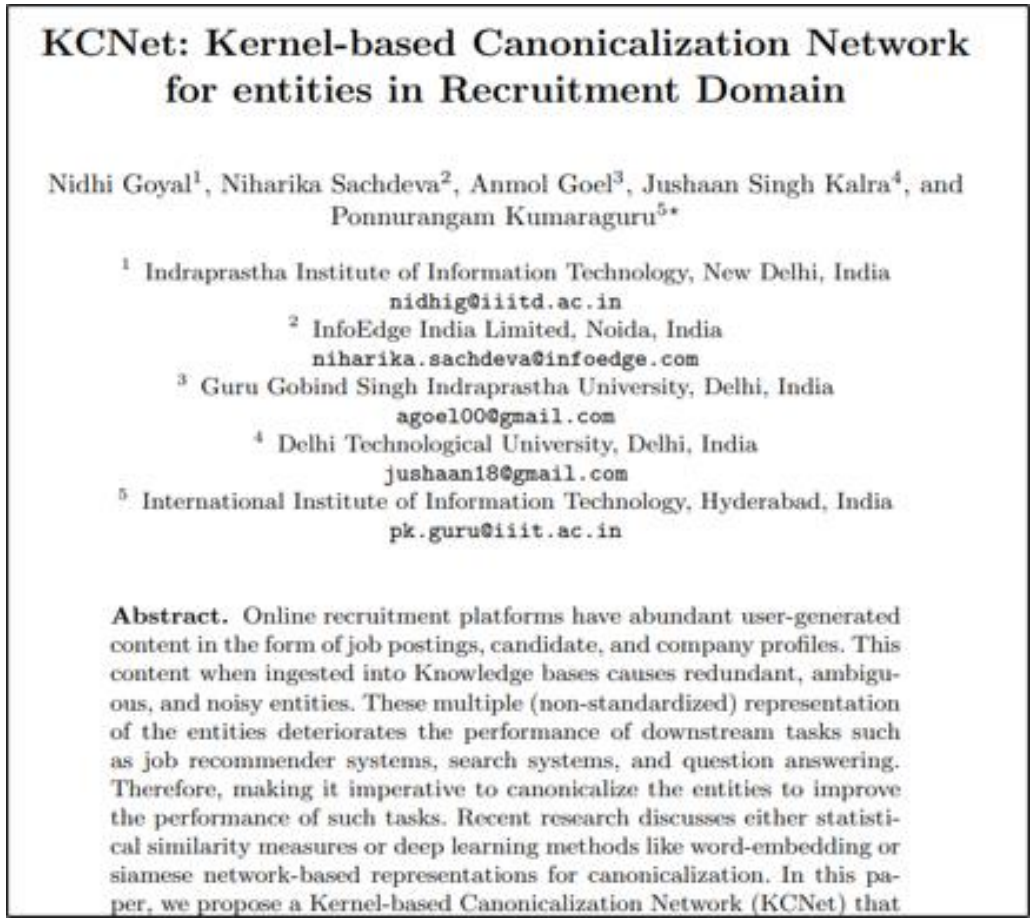
Industry examples:

Recruitment Domain

E-commerce

and many more.....

Research- Real-world/Industry use-case



KCNet: Kernel-based Canonicalization Network for entities in Recruitment Domain

Nidhi Goyal[1], Niharika Sachdeva[2], Anmol Goel[3], Jushaan Singh Kalra[4], and Ponnurangam Kumaraguru[5*]

[1] Indraprastha Institute of Information Technology, New Delhi, India
nidhig@iiitd.ac.in
[2] InfoEdge India Limited, Noida, India
niharika.sachdeva@infoedge.com
[3] Guru Gobind Singh Indraprastha University, Delhi, India
agoel00@gmail.com
[4] Delhi Technological University, Delhi, India
jushaan18@gmail.com
[5] International Institute of Information Technology, Hyderabad, India
pk.guru@iiit.ac.in

Abstract. Online recruitment platforms have abundant user-generated content in the form of job postings, candidate, and company profiles. This content when ingested into Knowledge bases causes redundant, ambiguous, and noisy entities. These multiple (non-standardized) representation of the entities deteriorates the performance of downstream tasks such as job recommender systems, search systems, and question answering. Therefore, making it imperative to canonicalize the entities to improve the performance of such tasks. Recent research discusses either statistical similarity measures or deep learning methods like word-embedding or siamese network-based representations for canonicalization. In this paper, we propose a Kernel-based Canonicalization Network (KCNet) that

https://cdn.iiit.ac.in/cdn/precog.iiit.ac.in/pubs/2021_July_KCNet-slides.pdf

Basic Normalization steps:

1. Segmenting/tokenizing words in running text
2. Normalizing word formats
3. Segmenting sentences in running text

Slide Reference: 2_TextProc_Mar_25_2021.pdf (stanford.edu)

Tokenization

Input: Mahindra university department

Tokens:

Mahindra

University

Department

A token is a sequence of characters in a document

What are valid tokens?

Hewlett-Packard Company

Are these two tokens "Hewlett" or "Packard" or one token?

Mahindra university -> 1 token or two?

State-of-the-art-> how many tokens?

Language issues-> Left-right or right-left (For example: Arabic)

Simple Code Example

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

Source: https://web.stanford.edu/~jurafsky/slp3/2.pdf

Complexity in Word tokenization

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

Another Solution-

Byte-pair encoding – [Read the example from book]

Byte pair encoding 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 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

https://web.stanford.edu/~jurafsky/slp3/2.pdf

Implementation

https://huggingface.co/learn/nlp-course/en/chapter6/5

https://github.com/SumanthRH/tokenization

Other tokenizers

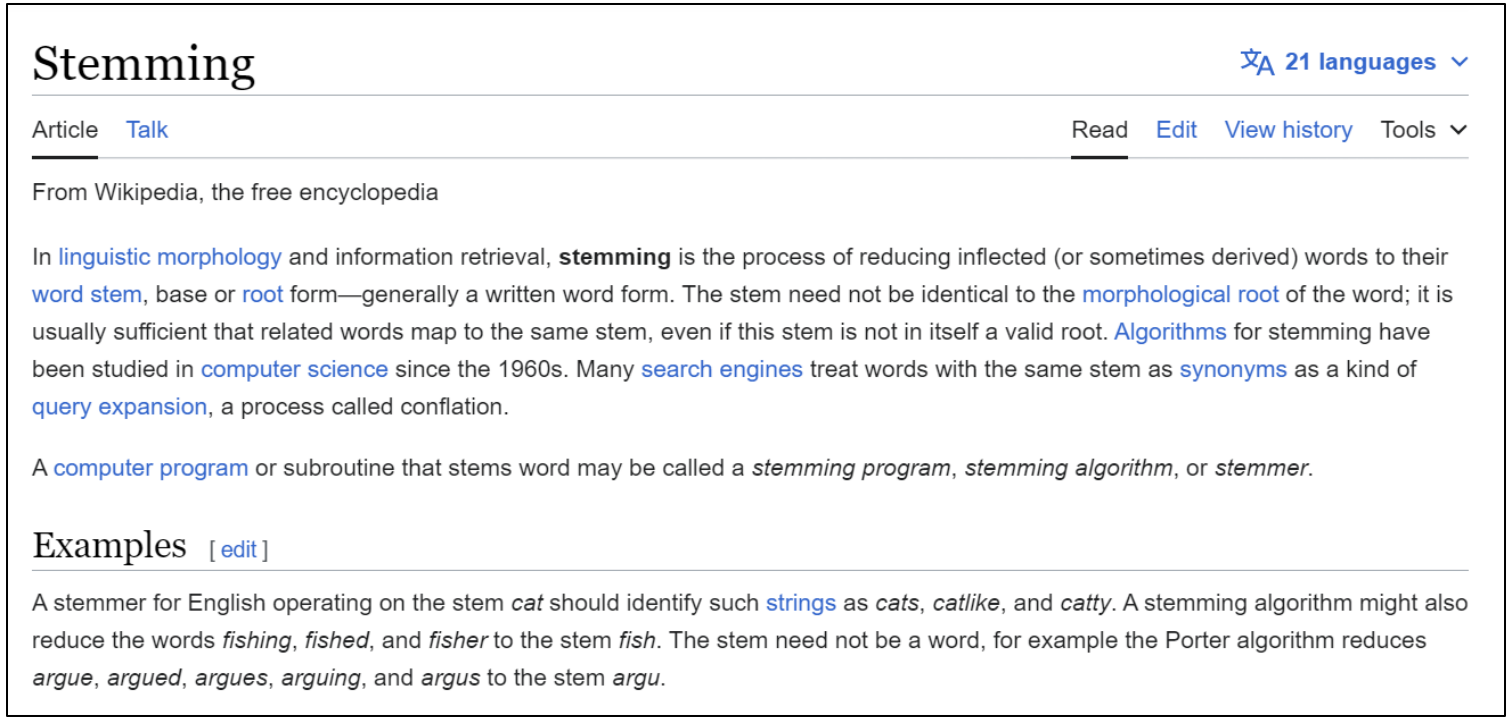
- Word piece tokenizers
- Sentence piece tokenizers

Stemming in NLP

affect	amus	close
affect	amuse	close
affectation	amused	closed
affected	amusement	closely
affecting	amusements	closing
affection	amusing	grate
affections		grate
affects		grateful
		gratefully

Source: https://www.projectpro.io/article/stemming-in-nlp/780

Stemming



Stemming

21 languages

Article Talk Read Edit View history Tools

From Wikipedia, the free encyclopedia

In linguistic morphology and information retrieval, **stemming** is the process of reducing inflected (or sometimes derived) words to their word stem, base or root form—generally a written word form. The stem need not be identical to the morphological root of the word; it is usually sufficient that related words map to the same stem, even if this stem is not in itself a valid root. Algorithms for stemming have been studied in computer science since the 1960s. Many search engines treat words with the same stem as synonyms as a kind of query expansion, a process called conflation.

A computer program or subroutine that stems word may be called a *stemming program*, *stemming algorithm*, or *stemmer*.

Examples [edit]

A stemmer for English operating on the stem *cat* should identify such strings as *cats*, *catlike*, and *catty*. A stemming algorithm might also reduce the words *fishing*, *fished*, and *fisher* to the stem *fish*. The stem need not be a word, for example the Porter algorithm reduces *argue*, *argued*, *argues*, *arguing*, and *argus* to the stem *argu*.

Source: https://en.wikipedia.org/wiki/Stemming#

Stemming

Stemming suggests crude affix chopping

- language dependent
- automation, automatic, automate ---> (automat)

Stemming programs are called as Stemmers or Stemming algorithms

Porter Stemming Algorithm (tartarus.org)

The Porter Stemmer (Porter, 1980)

- Common Algorithm for English language
- A simple rule-based algorithm for stemming
- An example of a HEURISTIC method
- Based on rules like:
  - ATIONAL -> ATE (e.g., rel**ational** -> rel**ate**)
- The algorithm consists of 7 sets of rules, applied in order

The Porter Stemmer: definitions

- Definitions:
  - **CONSONANTS**: a letter other than A, E, I, O, U, and Y preceded by consonant
  - **VOWEL**: any other letter (if the letter is not a consonant)
- With this definition, all words are of the form: $(C)(VC)^m(V)$
  - C: string of one or more consonants (con+)
  - V: string of one or more vowels
  - m: measure of word or word part which is represented in form of VC
- E.g.
  - **Troubles**
  - C $(VC)^m$ V

Measure of the word

- M=0 TREE, BY, TR
- M=1 TROUBLE, OATS, TREES, IVY
- M=2 TROUBLES, PRIVATE, OATEN

The Porter Stemmer: Rule format

- The rules are of the form:

(condition) S1 -> S2 where S1 and S2 are suffixes

- If the rule (m>1) EMENT->
  - In this S1 is EMENT and S2 is NULL
  - So, this would map REPLACEMENT with REPLAC

Conditions

m	The measure of the stem
***S**	The stem ends with S
v	The stem contains a vowel
***d**	The stem ends with a double consonant (TT,SS)
***o**	The stem ends in CV C (second C not W, X, or Y) Ex: WIL, HOP

The condition may also contains expressions with and, or, or not
Example ((m>1) and (*s or*t)) -tests for a stem with m>1 ending in s or t

The Porter Stemmer: Step 1

- SSES -> SS
  - caresses -> caress
- IES -> I
  - ponies -> poni
  - ties -> ti
- SS -> SS
  - caress -> caress
- S -> €
  - cats -> cat

The Porter Stemmer: Step 2a (past tense, progressive)

- (m>1) EED -> EE
  - **Condition verified**: agreed -> agree
  - **Condition not verified**: feed -> feed
- (*V*) ED -> €
  - **Condition verified**: plastered -> plaster
  - **Condition not verified**: bled -> bled
- (*V*) ING -> €
  - **Condition verified:** motoring -> motor
  - **Condition not verified**: sing -> sing

The Porter Stemmer: Step 2b (cleanup)

- (These rules are ran if second or third rule in 2a apply)
- AT -> ATE
  - Conflat(ed) -> conflate
- BL -> BLE
  - Troubl(ing) - > trouble
- (*d & ! (*L or *S or *Z)) -> single letter
  - **Condition verified**: hopp(ing) -> hop, tann(ed) -> tan
  - **Condition not verified**: fall(ing) -> fall
- (m=1 & *o) -> E
  - **Condition verified**: fil(ing) -> file
  - Condition not verified: fail -> fail

The Porter Stemmer: step 3 and 4

- Step 3: Y elimination (*V*) Y -> I
  - **Condition verified**: happy -> happi
  - **Condition not verified**: sky -> sky
- Step 4: Derivational Morphology, I
  - (m>0) ATIONAL -> ATE
    - Relational -> relate
  - (m>0) IZATION -> IZE
    - Generalization -> generalize
  - (m>0) BILITI -> BLE
    - Sensibiliti -> sensible

Porter Stemmer Step 5 and Step 6

- Derivational Morphology II
  - (m>0) ICATE-> IC
    - Triplicate-> Triplic
  - (m>0) FUL -> €
    - hopeful-> hope
  - (m>0) NESS-> €
    - goodness->good
- Derivational Morphology III
  - (m>0) ANCE-> €
    - allowance-> allow
  - (m>0) ENT -> €
    - dependent-> depend
  - (m>0) IVE-> €
    - effective->effect

The porter stemmer Step 7 (cleanup)

- Step 7a
  - (m>1) E -> €
    - Probate -> probat
  - (m=1 & !*o) NESS -> €
    - Goodness -> good
- Step 7 b
  - (m>1 & *d & *L) -> single letter
    - **Condition verified**: controll -> control
    - **Condition not verified:** roll -> roll

Lemmatization

Lemmatization

18 languages

Article Talk Read Edit View history Tools

From Wikipedia, the free encyclopedia

Lemmatization (or less commonly **lemmatisation**) in linguistics is the process of grouping together the inflected forms of a word so they can be analysed as a single item, identified by the word's lemma, or dictionary form.[1]

In computational linguistics, lemmatization is the algorithmic process of determining the lemma of a word based on its intended meaning. Unlike stemming, lemmatization depends on correctly identifying the intended part of speech and meaning of a word in a sentence, as well as within the larger context surrounding that sentence, such as neighbouring sentences or even an entire document. As a result, developing efficient lemmatization algorithms is an open area of research.[2][3][4]

Description [edit]

In many languages, words appear in several *inflected* forms. For example, in English, the verb 'to walk' may appear as 'walk', 'walked', 'walks' or 'walking'. The base form, 'walk', that one might look up in a dictionary, is called the *lemma* for the word. The association of the base form with a part of speech is often called a *lexeme* of the word.

Source: https://en.wikipedia.org/wiki/Lemmatization

Lemmatization

- Task of determining whether two words have same root despite surface differences

Lemmatization

- The most sophisticated methods for lemmatization involve complete **morphological parsing** of the word.
- Morphology is the study of morpheme the way words are built up from smaller meaning-bearing units called **morphemes**.
- Two broad classes of morphemes can be distinguished:
  - **stems**—the central moraffix pheme of the word, supplying the main meaning— and **affixes**—adding "additional" meanings of various kinds.
  - So, for example, the word **fox** consists of one morpheme (the morpheme **fox**) and the word **cats** consists of two: the morpheme **cat** and the morpheme -**s**.

Stemming vs Lemmatization

Stemming

achieve -> achiev
achieving -> achiev

- Can reduce words to a stem that is not an existing word
- Operates on a single word without knowledge of the context
- Simpler and faster

Lemmatization

achieve -> achieve
achieving -> achieve

- Reduces inflected words to their lemma, which is always an existing word
- Can leverage context to find the correct lemma of a word
- More accurate but slower

In-class activity

Exercise1:

- Convert these list of words into base form using Stemming and Lemmatization and observe the transformations
  - ['running', 'painting', 'walking', 'dressing', 'likely', 'children', 'whom', 'good', 'ate', 'fishing']

- Write a short note on the words that have different base words using stemming and Lemmatization

In class activity

Write a python code to use NLTK library and convert the base forms using different Stemmers and Lemmatizers

- use different stemmers and lemmatizers provided by NLTK
- see https://www.nltk.org/howto/stem.html for full NLTK stemmer module documentations

Reference materials

- https://vlanc-lab.github.io/mu-nlp-course/

- Lecture notes

- (A) Speech and Language Processing by Daniel Jurafsky and James H. Martin
- (B) Natural Language Processing with Python. (updated edition based on Python 3 and NLTK 3) Steven Bird et al. O'Reilly Media