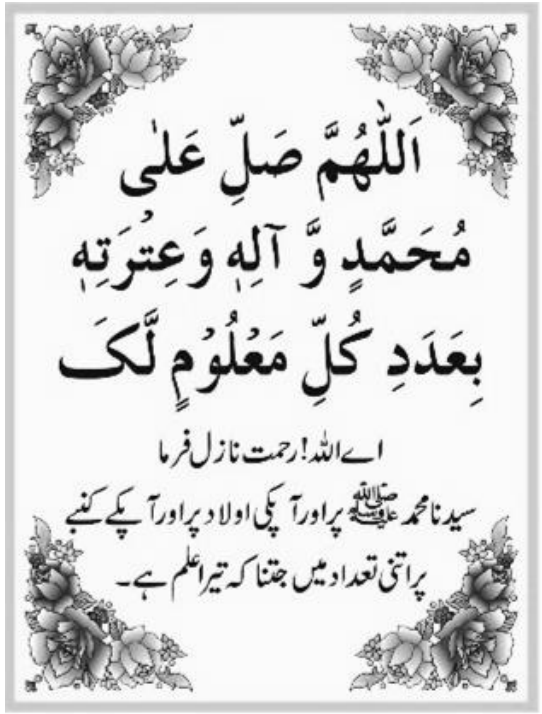
بِسْمِ اللّٰهِ الرَّحْمٰنِ الرَّحِيْمِ

Natural Language Processing (NLP)

Session 02 – Basic Text Processing

Instructor: Dr. Jawad Shafi

Dua – Take Help from Allah Before Starting Any Task



اللَّهُمَّ خِرْ لِيْ وَاخْتَرْ لِي

سُبْحَانَكَ لَا عِلْمَ لَنَا إِلَّا مَا عَلَّمْتَنَا ؕ
إِنَّكَ أَنْتَ الْعَلِيمُ الْحَكِيمُ

رَبِّ اشْرَحْ لِي صَدْرِي
وَيَسِّرْ لِي اَمْرِي
وَاحْلُلْ عُقْدَةً مِنْ لِسَانِي
يَفْقَهُوا قَوْلِي

Basic Preprocessing Techniques of NLP

1. Regular expressions
2. Stop words removal
3. Sentence tokenization
4. Word tokenization
5. Part-of-speech tagging
6. Stemming
7. Lemmatization

Regular Expressions

Basic Text Processing

in NLP

Regular expressions are used everywhere

- Part of every text processing task
  - Not a general NLP solution (for that we use large NLP systems we will see in later lectures)
  - But very useful as part of those systems (e.g., for pre-processing or text formatting)
- Necessary for data analysis of text data
- A widely used tool in industry and academics

Regular expressions

A formal language for specifying text strings

How can we search for mentions of these cute animals in text?

- woodchuck
- woodchucks
- Woodchuck
- Woodchucks
- Groundhog
- groundhogs

Regular Expressions: Disjunctions

Letters inside square brackets []

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

Ranges using the dash [A-Z]

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

Regular Expressions: Negation in Disjunction

Carat as first character in [] negates the list

- Note: Carat means negation only when it's first in []
- Special characters (., *, +, ?) lose their special meaning inside []

Pattern	Matches	Examples
[^A-Z]	Not an upper case letter	Oyfn pripetchik
[^Ss]	Neither ‘S’ nor ‘s’	I have no exquisite reason”
[^.]	Not a period	Our resident Djinn
[e^]	Either e or ^	Look up ^ now

Regular Expressions: Convenient aliases

Pattern	Expansion	Matches	Examples
\d	[0-9]	Any digit	Fahreneit 451
\D	[^0-9]	Any non-digit	Blue Moon
\w	[a-ZA-Z0-9_]	Any alphanumeric or _	Daiyu
\W	[^\w]	Not alphanumeric or _	Look!
\s	[\r\t\n\f]	Whitespace (space, tab)	Look␣up
\S	[^\s]	Not whitespace	Look up

Regular Expressions: More Disjunction

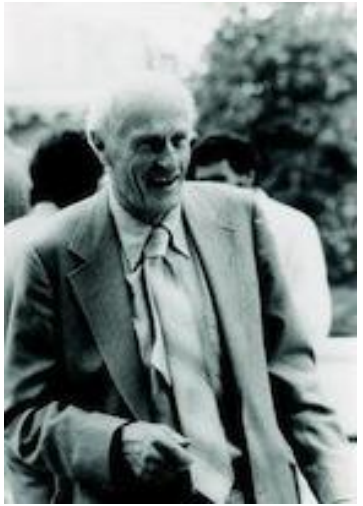
Groundhog is another name for woodchuck!

The pipe symbol | for disjunction

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

Wildcards, optionality, repetition: . ? * +

Pattern	Matches	Examples
beg.n	Any char	begin begun beg3n beg n
woodchucks?	Optional s	woodchuck woodchucks
to*	0 or more of previous char	t to too tooo
to+	1 or more of previous char	to too tooo toooo



Stephen C Kleene

Kleene *, Kleene +

Regular Expressions: Anchors ^ $

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

A note about Python regular expressions

- Regex and Python both use backslash "\" for special characters. You must type extra backslashes!
  - "\\d+" to search for 1 or more digits
  - "\n"a in Python means the "newline" character, not a "slash" followed by an "n". Need "\\n" for two characters.
- Instead: use Python's **raw string notation** for regex:
  - r"[tT]he"
  - r"\d+" matches one or more digits
    - instead of "\\d+"

The iterative process of writing regex's

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

the

Misses capitalized examples

[tT]he

Incorrectly returns other or Theology

\W[tT]he\W

False positives and false negatives

The process we just went through was based on **fixing two kinds of errors:**

1. Not matching things that we should have matched (**The**)

   False negatives

2. Matching strings that we should not have matched (**the**re, **the**n, o**the**r)

   False positives

Characterizing work on NLP

In NLP we are always dealing with these kinds of errors.

Reducing the error rate for an application often involves two antagonistic efforts:

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

Regular expressions play a surprisingly large role

Widely used in both academics and industry

1. Part of most text processing tasks, even for big neural language model pipelines
   - including text formatting and pre-processing
2. Very useful for data analysis of any text data

Basic Text Processing

More Regular Expressions: Substitutions and ELIZA

Substitutions

Substitution in Python and UNIX commands:

s/regexp1/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

He says I'm depressed much of the time.
I AM SORRY TO HEAR YOU ARE DEPRESSED

How ELIZA works

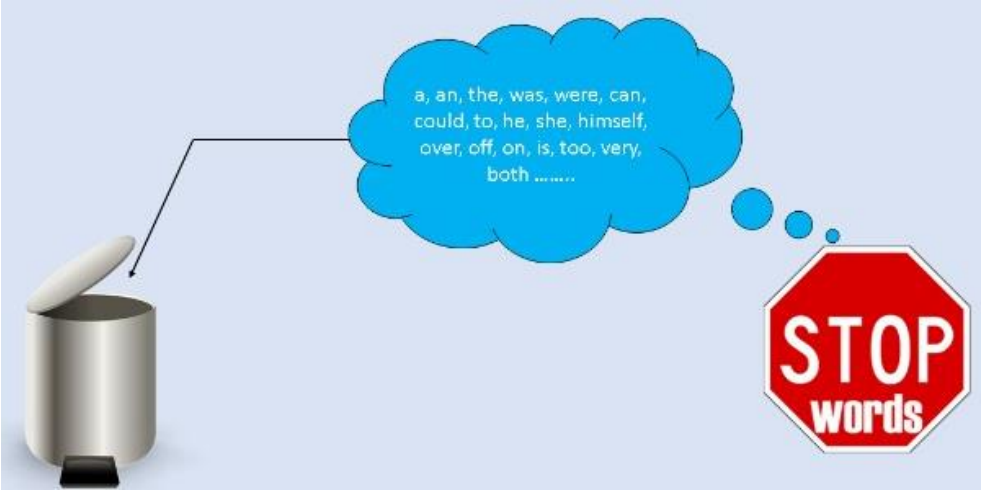
s/.* I’M (depressed|sad) .*/I AM SORRY TO HEAR YOU ARE \1/

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



Statistical Analysis: word frequency distribution

- Frequency of words will be inversely proportional to the rank of a words
  - Rank 1 is given to the most frequent word, 2 to the second most frequent word and so on.

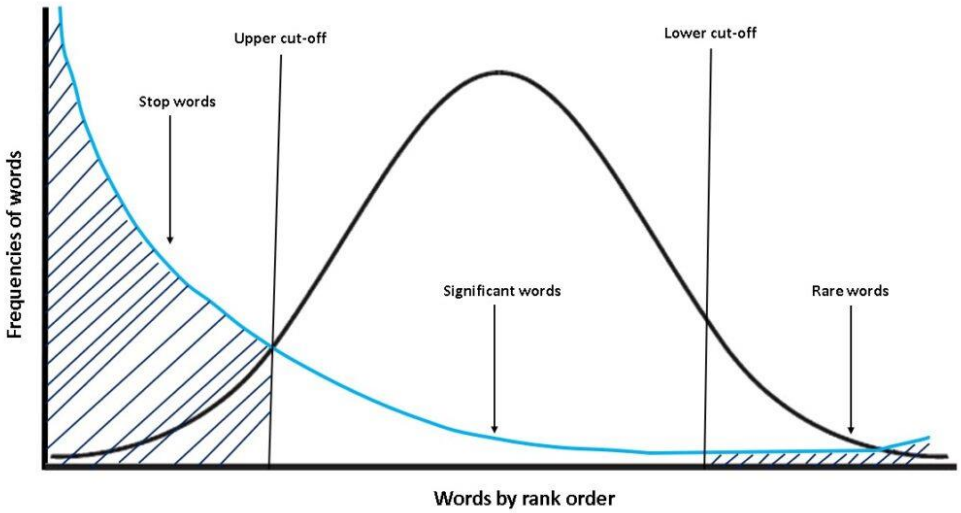


Fig: Graph showing relationship between frequency and rank of words

Stop words

- Highly occurring words in the document

or

- The words which are generally filtered out before processing a natural language are called stop words.

Stop Word(s)

These are actually the most common words in any language (like articles, prepositions, pronouns, conjunctions, etc.) and does not add much information to the text.

Examples of a few stop words in English are “the”, “a”, “an”, “so”, “what”.

Reason for Removing Stop Word(s)

1. They provide no meaningful information, especially if we are building a text classification model. Therefore, we have to remove stop words from our dataset.
2. As the frequency of stop words are too high, removing them from the corpus results in much smaller data in terms of size. Reduced size results in faster computations on text data and the text classification model need to deal with a lesser number of features resulting in a robust model.

Are they always useless for us?

The answer is no! 

We do not always remove the stop words.

The removal of stop words is highly dependent on the task we are performing and the goal we want to achieve.

Example, if we are training a model that can perform the sentiment analysis task, we might not remove the stop words.

***Movie review:* "The book was not good at all."**

***Text after removal of stop words:* "book good"**

Nutshell of removing stop words?

Tasks like text classification do not generally need stop words as the other words present in the dataset are more important and give the general idea of the text, i.e. → remove stop words in such tasks.

NLP has a lot of tasks that cannot be accomplished properly after the removal of stop words.

Words of caution: before removing stop words, research a bit about your task and the problem you are trying to solve, and then make your decision.

Nutshell of removing stop words?

Tasks like **text classification** **do not** generally need stop words as the other words present in the dataset are more important and give the general idea of the text, i.e. → **remove** stop words in such tasks.

NLP has a lot of tasks that **cannot be accomplished** properly after the removal of stop words.

Words of caution: before removing stop words, research a bit about your task and the problem you are trying to solve, and then make your decision.

Libraries to Remove Stop Words

Natural Language Toolkit (NLTK):

NLTK is an amazing library to play with natural language.

When you will start your NLP journey, this is the first library that you will use.

The steps to import the library and the English stop words list is given below:

```
import nltk
from nltk.corpus import stopwords
sw_nltk = stopwords.words('english’)
print(sw_nltk)
```

Output

```
['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you',
"you're", "you've", "you'll", "you'd", 'your', 'yours', 'yourself',
'yourselves', 'he', 'him', 'his', 'himself', 'she', "she's", 'her',
'hers', 'herself', 'it', "it's", 'its', 'itself', 'they', 'them',
'their', 'theirs', 'themselves', 'what', 'which', 'who', 'whom',
'this', 'that', "that'll", 'these', 'those', 'am', 'is', 'are',
'was', 'were', 'be', 'been', 'being', 'have', 'has', 'had', 'having',
'do', 'does', 'did', 'doing', 'a', 'an', 'the', 'and', 'but', 'if',
'or', 'because', 'as', 'until', 'while', 'of', 'at', 'by', 'for',
'with', 'about', 'against', 'between', 'into', 'through', 'during',
'before', 'after', 'above', 'below', 'to', 'from', 'up', 'down',
'in', 'out', 'on', 'off', 'over', 'under', 'again', 'further',
'then', 'once', 'here', 'there', 'when', 'where', 'why', 'how',
'all', 'any', 'both', 'each', 'few', 'more', 'most', 'other', 'some',
'such', 'no', 'nor', 'not', 'only', 'own', 'same', 'so', 'than',
'too', 'very', 's', 't', 'can', 'will', 'just', 'don', "don't",
'should', "should've", 'now', 'd', 'll', 'm', 'o', 're', 've', 'y',
'ain', 'aren', "aren't", 'couldn', "couldn't", 'didn', "didn't",
'doesn', "doesn't", 'hadn', "hadn't", 'hasn', "hasn't", 'haven',
"haven't", 'isn', "isn't", 'ma', 'mightn', "mightn't", 'mustn',
"mustn't", 'needn', "needn't", 'shan', "shan't", 'shouldn',
"shouldn't", 'wasn', "wasn't", 'weren', "weren't", 'won', "won't",
'wouldn', "wouldn't"]
```

How Many Stop Words?

```
print(len(sw_nltk))
```

Output:

```
179
```

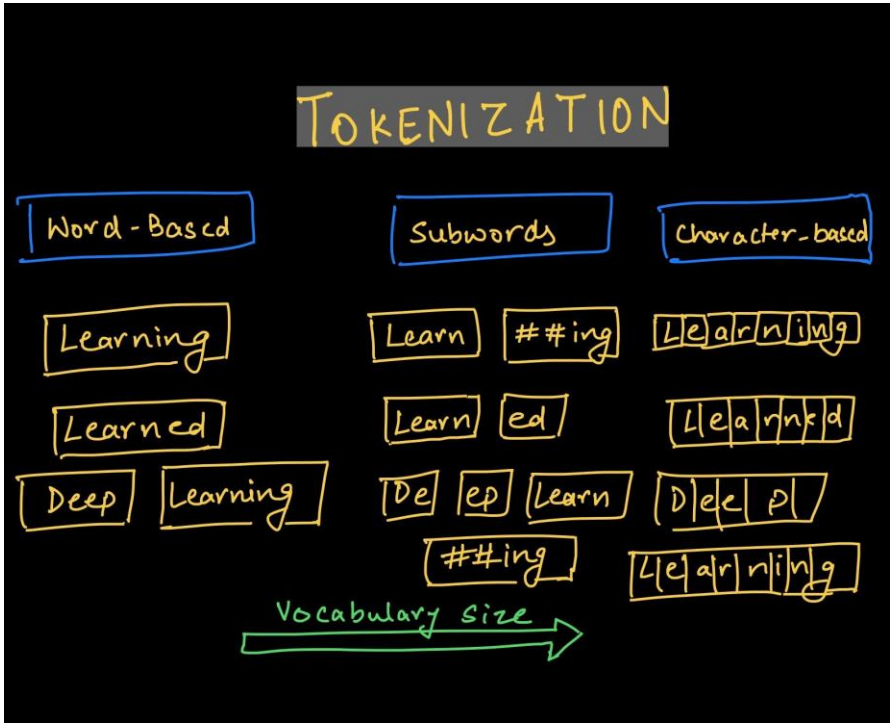
Code to Remove Stop Words?

```
text = "When I first met her she was very quiet. She remained quiet
during the entire two hour long journey from Stony Brook to New
York."

words = [word for word in text.split() if word.lower() not in
sw_nltk]
new_text = " ".join(words)

print(new_text)
print("Old length: ", len(text))
print("New length: ", len(new_text))
```

```
first met quiet. remained quiet entire two hour long journey Stony
Brook New York.
Old length:  129
New length:  82
```

Basic Text Processing

of words / sentences



Tokenization

Most of the NLP task requires text:

1. Word Tokenizing (segmenting)
2. Sentence Tokenization

Sentence Tokenization (ST)

Current immunosuppression protocols to prevent lung transplant rejection reduce pro-inflammatory and T-helper type 1 (Th1) cytokines. However, Th1 T-cell pro-inflammatory cytokine production is important in host defense against bacterial infection in the lungs. Excessive immunosuppression of Th1 T-cell pro-inflammatory cytokines leaves patients susceptible to infection.

S1- Current immunosuppression protocols to prevent lung transplant rejection reduce pro-inflammatory and T-helper type 1 (Th1) cytokines.

S2- However, Th1 T-cell pro-inflammatory cytokine production is important in host defense against bacterial infection in the lungs.

S3- Excessive immunosuppression of Th1 T-cell pro-inflammatory cytokines leaves patients susceptible to infection.

A Heuristic Rule for Sentence Tokenization

SBD

= period + space(s) + capital letter

Regular expression in Perl

```
s/\. +([A-Z])/\.\n\1/g;
```

Errors

IL-33 is known to induce the production of Th2-associated cytokines (e.g. IL-5 and IL-13).

IL-33 is known to induce the production of Th2-associated cytokines

(e.g. IL-5 and IL-13).

Two solutions:

- **Add more rules to handle exceptions**

or

- **Machine learning**

Tools for Sentence Tokenization

- JASMINE
  - Rule-based
  - http://uvdb3.hgc.jp/ALICE/program_download.html
- Scott Piao's splitter
  - Rule-based
  - http://text0.mib.man.ac.uk:8080/scottpiao/sent_detector
- OpenNLP
  - Maximum-entropy learning
  - https://sourceforge.net/projects/opennlp/
  - Needs training data

Sentence Tokenization Challenges for Urdu

- **Does not use any special character Lowe/Upper**
- **Punctuation markers are not always ending delimiter**
- **Sentences are written without punctuation markers**

انڈیا میں آئی سی سی ورلڈ ٹی ۲۰ کا آغاز ۔

یو۔ ای ۔ اے میں کافی پاکستانی بستے ہیں ۔

« میری خیال میں ان کی وزیر خارجہ ۲۱ اگست کو آرہی ہیں »

آج ۲۰۱۵ - ۶ - ۳ ہے -

» «پاکستان» ۲ - ٤ سے جیت رہا ہے -

مشرف کو باہر کیوں جانے دیا گیا ؟

اس پر بھی عوام نہ سمجھے تو !

حضور والا ! آپ پوری ملک کے بادشاہ ہیں ۔

ابھی چند سال کا وقت لگے گا

Rule Base Approach: ST

1. If the current character is a period marker (.) *AND* the same mark appears after two or three characters, then consider it as an abbreviation and match it in the abbreviation list.
2. If within the next 9 characters (from any previous SBM marker), an exclamation mark (!) is found, then this is not a sentence boundary marker.
3. If the character before a double quote (») is a period (.) or question (») mark, then it is a sentence boundary marker.
4. Apply regular expressions for detecting the date and hyphenated numeric values.
5. In addition to this all the above rules from 1 to 4, split sentences based on the question (?), period (.), and exclamation (!) markers.

ML Approach for ST

- Probability (UMC data set[v] is used) that a word with "؟, ـ and !" occurs at the end of a sentence
- Probability (UMC data set[w] is used) that a word with "؟, ـ and !" occurs at the beginning of a sentence
- Length of a word with "؟, ـ and !"
- Length of a word after "؟, ـ and !"
- Is a sentence contains an abbreviation
- Is a sentence contains a date/numeration
- *Bi*- and *tri*-grams words information (preceding "؟, ـ and !") are used
- If a word before "؟, ـ and !" markers contains any one of the tag (NN, NNP, JJ, SC, PDM, PRS, CD, OD, FR, Q, and CC.POS tags) is not a sentence boundary

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

Word Tokenization (WT)

The protein is activated by IL2.

The protein is activated by IL2 .

- Convert a sentence into a sequence of *tokens*

- Why do we tokenize?
- Because we do not want to treat a sentence as a sequence of *characters*!

Word Tokenization Conti.

The protein is activated by IL2.

The protein is activated by IL2 .

- Tokenizing general English sentences is relatively straightforward.
- Use spaces as the boundaries
- Use some heuristics to handle exceptions

Word Tokenization Issues

- separate possessive endings or abbreviated forms from preceding words:
  - Mary's → Mary 's
    Mary's → Mary is
    Mary's → Mary has
- separate punctuation marks and quotes from words :
  - Mary. → Mary .
  - "new" → " new "

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)

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

WT Problems in Bio-text

- Commas
  - 2,6-diaminohexanoic acid
  - tricyclo(3.3.1.13,7)decanone
- Four kinds of hyphens
  - "Syntactic:"
  - *Calcium-dependent*
  - *Hsp-60*
  - Knocked-out gene: *lush-- flies*
  - Negation: *-fever*
  - Electric charge: *Cl-*

WT Conti.

- **Tokenization:** Divides the text into smallest units (usually words), removing punctuation. Challenge: What should be done with punctuation that has linguistic meaning?
- Negative charge **(Cl-)**
- Absence of symptom **(-fever)**
- Knocked-out gene **(Ski-/-)**
- Gene name **(IL-2 –mediated)**
- Plus, “syntactic”uses **(insulin-dependent)**

Challenges in Urdu Word Tokenization

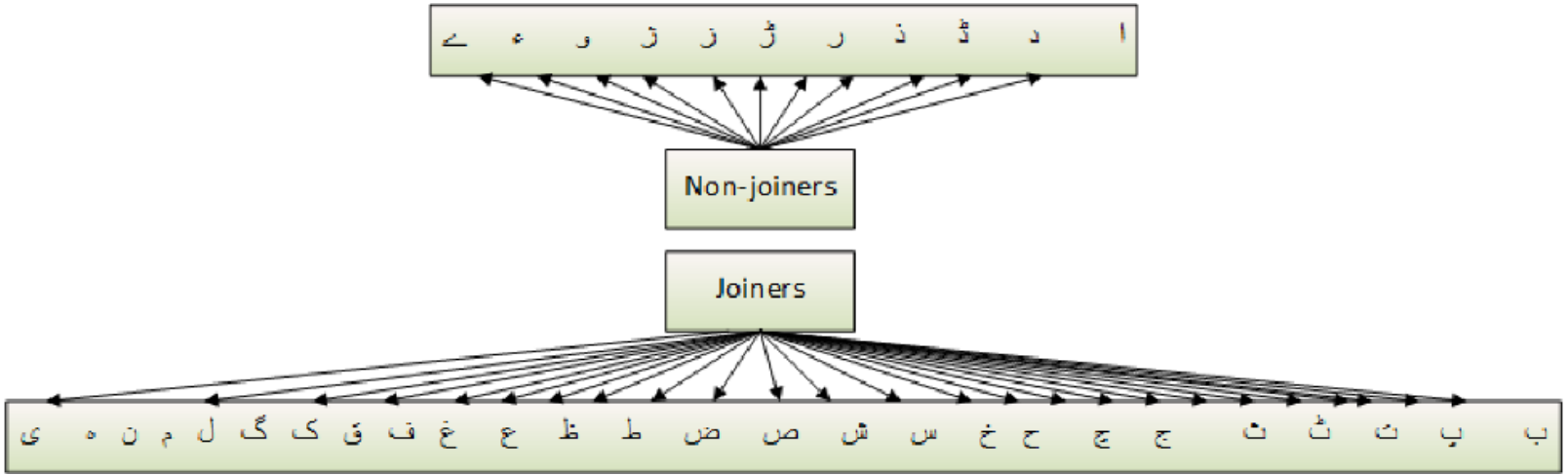
Space omission — Urdu uses Nastalique writing style and cursive script, in which Urdu text does not often contain spaces between words.

Space insertion — irregular use of spaces within two or more words.

- Above two problems are because of
  - Joiner and nonjoiner

Ambiguity in defining Urdu words—in some cases, Urdu words lead to an ambiguity problem because there is no clear agreement to classify them as a single word or multiple word

Challenges in Urdu Word Tokenization



Challenges in Urdu Word Tokenization

Type	Correct	Incorrect	Translation
Affixation	خوش ـ اخلاق KHOSH AKHLAK	خوشاخلاق KHOSHAKHLAK	Polite
Abbreviations	ای ـ ایل ـ این AYN AYL AY	ایناایلای AYNAYLAY	NLE
Compound word	تغیر ـ پذیر TGHYR PZYR	تغیرپذیر TGHYR PZYR	Variable
English word	ورک ـ نٹ NYT ORK	نٹورک NYTORK	Network
Proper noun	انڈیز ـ ویسٹ OYST ANDYZ	ویسٹانڈیز OYSTANDYZ	West Indies
Reduplication	فانن ـ آنن AANN FANN	آننفانن AANNFANN	Quickly

Challenges in Urdu Word Tokenization

Original Text	Tokenization Issue	Example After Tokenization	Explanation
حکومت نے ملکی معیشت بہتر بنانے کا اعلان کیا۔	Space Omission	حکومت نے ملکی معیشت بہتر بنانے کا اعلان کیا۔	The phrase "ملکی معیشت"(national economy) and "بہتر بنانے" (improve) should be tokenized as separate words, requiring the insertion of spaces where omitted.
اعلیٰ عدالت نے فیصلہ سنا دیا۔	Compound Word	اعلیٰ عدالت نے فیصلہ سنا دیا۔	"اعلیٰ عدالت"(High Court) is a compound word that should be recognized as a single token during processing to maintain its semantic integrity.
کراچی کے قریبی علاقوں میں شدید بارشیں ہوئیں۔	Space Omission	کراچی کے قریبی علاقوں میں شدید بارشیں ہوئیں۔	The phrase "کے قریبی "(nearby) requires a space to be inserted for accurate tokenization.
انہوں نے کہا کہ "سب ٹھیک ہو جائے گا"۔	Space Omission in Quotes	انہوں نے کہا کہ "سب ٹھیک ہو جائے گا"۔	The phrase"ہو جائے گا " (will be okay) inside quotes should be tokenized with spaces:" ہو جائے گا ".
وزیر اعظم نے سی پیک کے معاہدے پر دستخط کیے۔	Space Omission	وزیر اعظم نے سی پیک کے معاہدے پر دستخط کیے۔	"سی پیک"(CPEC) is a compound word requiring proper tokenization by inserting spaces around it.

How to do word tokenization in Urdu?

1- Rule-based maximum matching,

2- Dictionary lookup,

3- Statistical tri-gram Maximum Likelihood Estimation (MLE)

with back-off to bi-gram and uni-gram MLE

4- Smoothing

Tokenization in NLTK

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

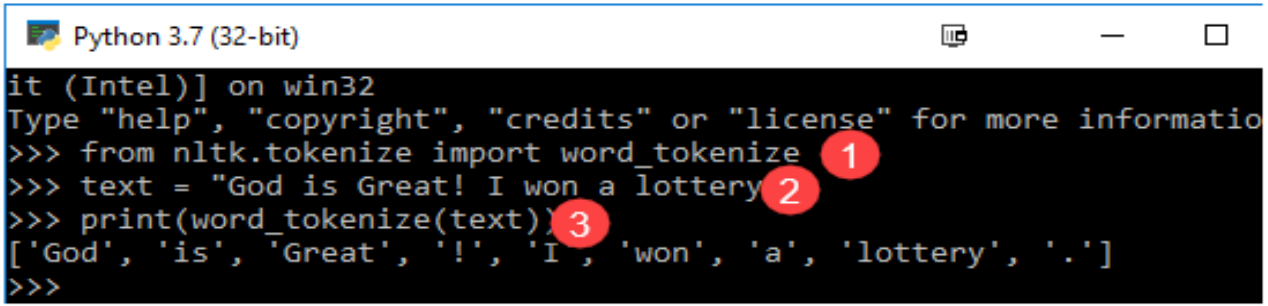
```
>>> text = 'That U.S.A. poster-print costs $12.40...'
>>> pattern = r'''(?x)    # set flag to allow verbose regexps
...     ([A-Z]\.)+        # abbreviations, e.g. U.S.A.
...   | \w+(-\w+)*        # words with optional internal hyphens
...   | \$?\d+(\.\d+)?%?  # currency and percentages, e.g. $12.40, 82%
...   | \.\.\.            # ellipsis
...   | [][.,;"'?():-_`]  # these are separate tokens; includes ], [
... '''
>>> nltk.regexp_tokenize(text, pattern)
['That', 'U.S.A.', 'poster-print', 'costs', '$12.40', '...']
```

NLTK Word Tokenizer Code

Code

```
from nltk.tokenize import word_tokenize
text = "God is Great! I won a lottery."
print(word_tokenize(text))

Output: ['God', 'is', 'Great', '!', 'I', 'won', 'a', 'lottery', '.']
```

Output



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) Tokenization

Byte-Pair Encoding (BPE) was initially developed as an algorithm to compress texts, and

Letter used by OpenAI for tokenization when pretraining the GPT model. It's used by a lot of Transformer models, including GPT, GPT-2, RoBERTa, BART, and DeBERTa.

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 – Running Example

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 – Example conti.

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 _	

Merge er _ to 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 _	

BPE – Conti.

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 _	

Merge n e 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 _	

BPE – Conti.

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*

Building a Tokenizer

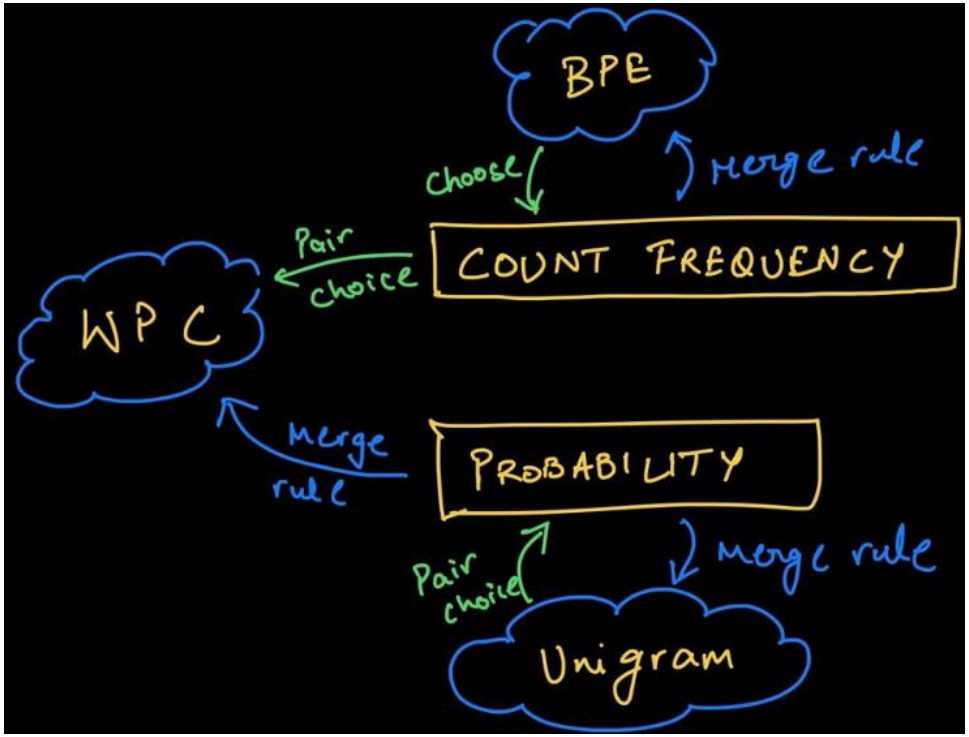
Tokenization comprises several steps:

- Normalization (any cleanup of the text that is deemed necessary, such as removing spaces or accents, Unicode normalization, etc.)
- Pre-tokenization (splitting the input into words)
- Running the input through the model (using the pre-tokenized words to produce a sequence of tokens)
- Post-processing (adding the special tokens of the tokenizer, generating the attention mask and token type IDs)

Basic Text Processing

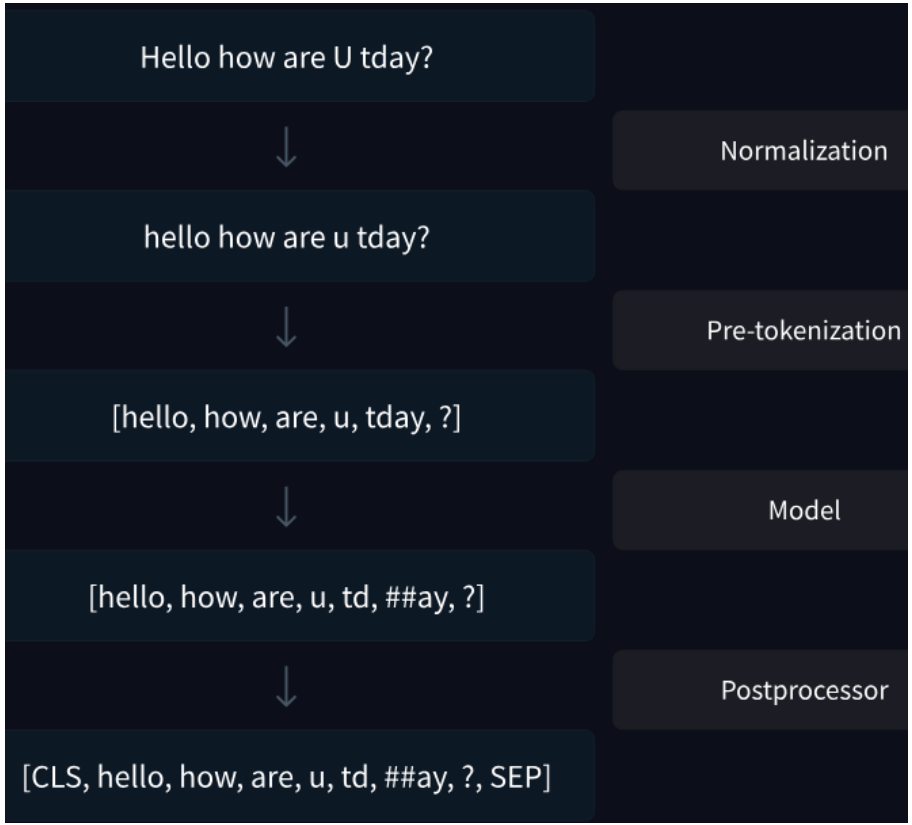
Tokenizers

Building



Building a Tokenizer

As a reminder, here's another look at the overall process:



Building a Tokenizer - Code

1- Take a corpus

```
corpus = [
    "This is the Hugging Face Course.",
    "This chapter is about tokenization.",
    "This section shows several tokenizer algorithms.",
    "Hopefully, you will be able to understand how they are trained and generate tokens.",
]
```

2- Pre-tokenize that corpus into words → here we are using gpt2

```
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("gpt2")
```

Building a Tokenizer - Code

3- Compute the frequencies of each word in the corpus as we do the pre-tokenization:

```
from collections import defaultdict

word_freqs = defaultdict(int)

for text in corpus:
    words_with_offsets = tokenizer.backend_tokenizer.pre_tokenizer.pre_tokenize_str(text)
    new_words = [word for word, offset in words_with_offsets]
    for word in new_words:
        word_freqs[word] += 1

print(word_freqs)
```

defaultdict(int, {'This': 3, 'Ġis': 2, 'Ġthe': 1, 'ĠHugging': 1, 'ĠFace': 1, 'ĠCourse': 1, '.': 4, 'Ġchapter': 1, 'Ġabout': 1, 'Ġtokenization': 1, 'Ġsection': 1, 'Ġshows': 1, 'Ġseveral': 1, 'Ġtokenizer': 1, 'Ġalgorithms': 1, 'Hopefully': 1, ',': 1, 'Ġyou': 1, 'Ġwill': 1, 'Ġbe': 1, 'Ġable': 1, 'Ġto': 1, 'Ġunderstand': 1, 'Ġhow': 1, 'Ġthey': 1, 'Ġare': 1, 'Ġtrained': 1, 'Ġand': 1, 'Ġgenerate': 1, 'Ġtokens': 1})

Building a Tokenizer - Code

4- To compute the base vocabulary, formed by all the characters used in the corpus:

```
alphabet = []

for word in word_freqs.keys():
    for letter in word:
        if letter not in alphabet:
            alphabet.append(letter)
alphabet.sort()

print(alphabet)
```

[',', '.', 'C', 'F', 'H', 'T', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'k', 'l', 'm', 'n', 'o', 'p', 'r', 's', 't', 'u', 'v', 'w', 'y', 'z', 'Ġ']

5- Add the special tokens used by the model at the beginning of that vocabulary. In the case of GPT-2, the only special token is "<|endoftext|>":

```
vocab = ["<|endoftext|>"] + alphabet.copy()
```

Building a Tokenizer - Code

6- Now need to split each word into individual characters, to be able to start training:

```
splits = {word: [c for c in word] for word in word_freqs.keys()}
```

7- Ready for training, let's write a function that computes the frequency of each pair. We'll need to use this at each step of the training:

```
def compute_pair_freqs(splits):
    pair_freqs = defaultdict(int)
    for word, freq in word_freqs.items():
        split = splits[word]
        if len(split) == 1:
            continue
        for i in range(len(split) - 1):
            pair = (split[i], split[i + 1])
            pair_freqs[pair] += freq
    return pair_freqs
```

Note: BPE token learner algorithm i.e. slide 67

Building a Tokenizer - Code

Let's have a look at a part of this dictionary after the initial splits:

```
pair_freqs = compute_pair_freqs(splits)

for i, key in enumerate(pair_freqs.keys()):
    print(f"{key}: {pair_freqs[key]}")
    if i >= 5:
        break
```

```
('T', 'h'): 3
('h', 'i'): 3
('i', 's'): 5
('Ġ', 'i'): 2
('Ġ', 't'): 7
('t', 'h'): 3
```

8 - Now, finding the most frequent pair only takes a quick loop:

```
best_pair = ""
max_freq = None

for pair, freq in pair_freqs.items():
    if max_freq is None or max_freq < freq:
        best_pair = pair
        max_freq = freq

print(best_pair, max_freq)
```

```
('Ġ', 't') 7
```

Building a Tokenizer - Code

The first merge to learn is ('Ġ', 't') -> 'Ġt', and we add 'Ġt' to the vocabulary:

```
merges = {("Ġ", "t"): "Ġt"}
vocab.append("Ġt")
```

9- To continue, we need to apply that merge in our splits dictionary. Let's write another function for this:

```
def merge_pair(a, b, splits):
    for word in word_freqs:
        split = splits[word]
        if len(split) == 1:
            continue

        i = 0
        while i < len(split) - 1:
            if split[i] == a and split[i + 1] == b:
                split = split[:i] + [a + b] + split[i + 2 :]
            else:
                i += 1
        splits[word] = split
    return splits
```

```
splits = merge_pair("Ġ", "t", splits)
print(splits["Ġtrained"])
```

```
['Ġt', 'r', 'a', 'i', 'n', 'e', 'd']
```

Building a Tokenizer - Code

10- Now we have everything we need to loop until we have learned all the merges we want. Let's aim for a vocab size of 50:

```
vocab_size = 50

while len(vocab) < vocab_size:
    pair_freqs = compute_pair_freqs(splits)
    best_pair = ""
    max_freq = None
    for pair, freq in pair_freqs.items():
        if max_freq is None or max_freq < freq:
            best_pair = pair
            max_freq = freq
    splits = merge_pair(*best_pair, splits)
    merges[best_pair] = best_pair[0] + best_pair[1]
    vocab.append(best_pair[0] + best_pair[1])
```

Building a Tokenizer - Code

As a result, we've learned 19 merge rules (the initial vocabulary had a size of 31 — 30 characters in the alphabet, plus the special token):

```
print(merges)
```

{('Ġ', 't'): 'Ġt', ('i', 's'): 'is', ('e', 'r'): 'er', ('Ġ', 'a'): 'Ġa', ('Ġt', 'o'): 'Ġto', ('e', 'n'): 'en', ('T', 'h'): 'Th', ('Th', 'is'): 'This', ('o', 'u'): 'ou', ('s', 'e'): 'se', ('Ġto', 'k'): 'Ġtok', ('Ġtok', 'en'): 'Ġtoken', ('n', 'd'): 'nd', ('Ġ', 'is'): 'Ġis', ('Ġt', 'h'): 'Ġth', ('Ġth', 'e'): 'Ġthe', ('i', 'n'): 'in', ('Ġa', 'b'): 'Ġab', ('Ġtoken', 'i'): 'Ġtokeni'}

Vocabulary is composed of the special token, the initial alphabet, and all the results of the merges:

```
print(vocab)
```

['<|endoftext|>', ',', '.', 'C', 'F', 'H', 'T', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'k', 'l', 'm', 'n', 'o', 'p', 'r', 's', 't', 'u', 'v', 'w', 'y', 'z', 'Ġ', 'Ġt', 'is', 'er', 'Ġa', 'Ġto', 'en', 'Th', 'This', 'ou', 'se', 'Ġtok', 'Ġtoken', 'nd', 'Ġis', 'Ġth', 'Ġthe', 'in', 'Ġab', 'Ġtokeni']

Building a Tokenizer - Code

To tokenize a new text, we pre-tokenize it, split it, then apply all the merge rules learned:

```
def tokenize(text):
    pre_tokenize_result = tokenizer._tokenizer.pre_tokenizer.pre_tokenize_str(text)
    pre_tokenized_text = [word for word, offset in pre_tokenize_result]
    splits = [[l for l in word] for word in pre_tokenized_text]
    for pair, merge in merges.items():
        for idx, split in enumerate(splits):
            i = 0
            while i < len(split) - 1:
                if split[i] == pair[0] and split[i + 1] == pair[1]:
                    split = split[:i] + [merge] + split[i + 2 :]
                else:
                    i += 1
            splits[idx] = split

    return sum(splits, [])
```

We can try this on any text composed of characters in the alphabet:

```
tokenize("This is not a token.")
```

```
['This', 'Ġis', 'Ġ', 'n', 'o', 't', 'Ġa', 'Ġtoken', '.']
```

WordPiece tokenization

Almost same:

Like BPE, WordPiece starts from a small vocabulary including the special tokens used by the model and the initial alphabet.
Since it identifies subwords by adding a prefix (like ## for BERT), each word is initially split by adding that prefix to all the characters inside the word. So, for instance, "word" gets split like this:

w ##o ##r ##d

WordPiece tokenization

The main difference lies is the way the pair to be merged is selected.

Instead of selecting the most frequent pair, WordPiece computes a score for each pair, using the following formula:

$$\text{score} = (\text{freq\_of\_pair})/(\text{freq\_of\_first\_element} \times \text{freq\_of\_second\_element})$$

WordPiece Tokenization

The main difference lies is the way the pair to be merged is selected.

Instead of selecting the most frequent pair, WordPiece computes a score for each pair, using the following formula:

$$\text{score} = (\text{freq\_of\_pair})/(\text{freq\_of\_first\_element} \times \text{freq\_of\_second\_element})$$

WordPiece Tokenization

By dividing the frequency of the pair by the product of the frequencies of each of its parts, the algorithm prioritizes the merging of pairs where the individual parts are less frequent in the vocabulary.

For instance, it won't necessarily merge **("un", "##able")** even if that pair occurs very frequently in the vocabulary, because the two pairs **"un"** and **"##able"** will likely each appear in a lot of other words and have a high frequency. In contrast, a pair like **("hu", "##gging")** will probably be merged faster (assuming the word "**hugging**" appears often in the vocabulary) since **"hu"** and **"##gging"** are likely to be less frequent individually.

Uni-gram Tokenization

Compared to BPE and WordPiece, Unigram works in the other direction:

It starts from a big vocabulary and removes tokens from it until it reaches the desired vocabulary size.

At each step of the training, the Unigram algorithm computes a loss over the corpus given the current vocabulary.

Word Normalization & other Issues

Letter	Variations		
ش			
ط			
و			
ك			
م			
ر			
ق			

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

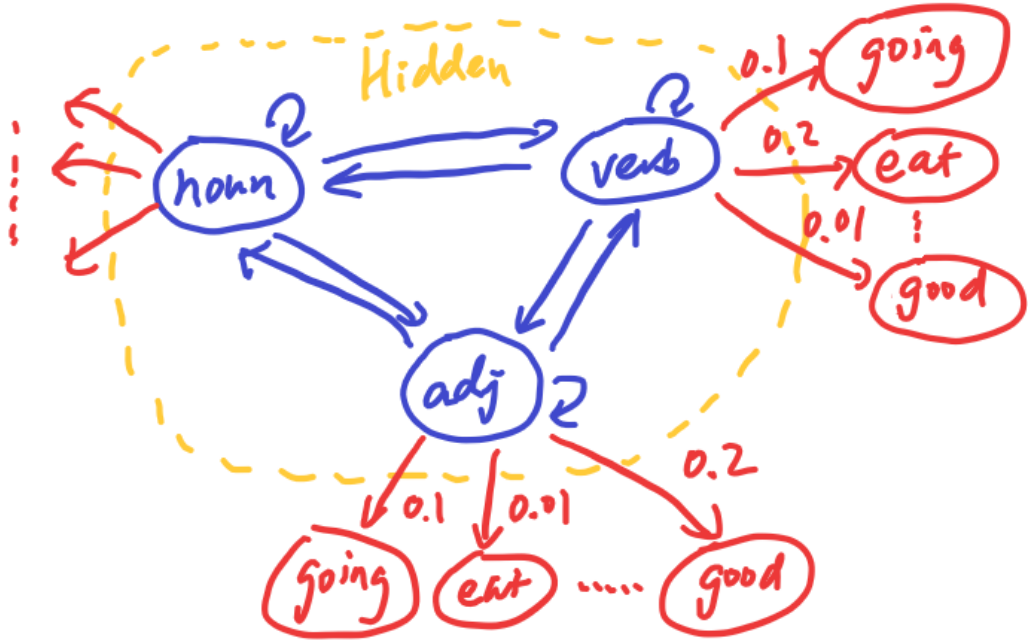
Applications like IR: reduce all letters to lower case

- 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)

Basic Text Processing



POS Tagging (Grammatical Tagging)

An important NLP pre-processing task and will be covered in coming lectures.

Basic Text Processing

STEMMING & LEMMATIZATION

Stemming

Reduce terms to their “roots” before indexing

“Stemming” suggest crude affix chopping

- language dependent
- e.g., ***automate(s), automatic, automation*** all reduced to ***automat***.

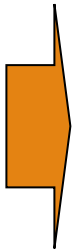
for example compressed and compression are both accepted as equivalent to compress.

for exampl compress and compress ar both accept as equival to compress

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:

$$\begin{aligned}
\text{ATIONAL} &\rightarrow \text{ATE} \quad \text{(e.g., relational} \rightarrow \text{relate)} \\
\text{ING} &\rightarrow \epsilon \quad \text{if stem contains vowel (e.g., motoring} \rightarrow \text{motor)} \\
\text{SSES} &\rightarrow \text{SS} \quad \text{(e.g., grasses} \rightarrow \text{grass)}
\end{aligned}$$

Program for Stemming

```
from nltk.stem import PorterStemmer
e_words= ["wait", "waiting", "waited", "waits"]
ps =PorterStemmer()
for w in e_words:
    rootWord=ps.stem(w)
    print(rootWord)
```

Output:

```
wait
wait
wait
wait
```

Lemmatization

Reduce inflectional/variant forms to base form

E.g.,

- *am, are, is* → *be*
- *car, cars, car's, cars'* → *car*

the boy's cars are different colors → *the boy car be different color*

Lemmatization implies doing "proper" reduction to dictionary headword form

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*

Techniques for Lemmatization

1- Rule-based lemmatization involves the application of predefined rules to derive the base or root form of a word. Unlike machine learning-based approaches, which learn from data, rule-based lemmatization relies on linguistic rules and patterns.

Here's a simplified example of rule-based lemmatization for English verbs:

Rule: For regular verbs ending in "-ed," remove the "-ed" suffix.

Example:

Word: "walked"

Rule Application: Remove "-ed"

Result: "walk

Techniques for Lemmatization

2- Dictionary-based lemmatization

relies on predefined dictionaries or lookup tables to map words to their corresponding base forms or lemmas. Each word is matched against the dictionary entries to find its lemma. This method is effective for languages with well-defined rules.

Suppose we have a dictionary with lemmatized forms for some words:

'running' -> 'run'

'better' -> 'good'

'went' -> 'go'

Techniques for Lemmatization

3- ML lemmatization leverages computational models to automatically learn the relationships between words and their base forms. Unlike rule-based or dictionary-based approaches, machine learning (ML) models, such as neural networks or statistical models, are trained on large text datasets to generalize patterns in language.

e.g. Consider a machine learning-based lemmatizer trained on diverse texts. When encountering the word 'went,' the model, having learned patterns, predicts the base form as 'go.' Similarly, for 'happier,' the model deduces 'happy' as the lemma. The advantage lies in the model's ability to adapt to varied linguistic nuances and handle irregularities, making it robust for lemmatizing diverse vocabularies.

Lemmatization

1. WordNet
2. WordNet (with POS tag)
3. TextBlob
4. TextBlob (with POS tag)
5. spaCy
6. TreeTagger
7. Pattern
8. Gensim
9. Stanford CoreNLP

Wordnet Lemmatizer

Wordnet is a publicly available lexical database of over 200 languages that provides semantic relationships between its words. It is one of the earliest and most commonly used lemmatizer technique.

It is present in the NLTK library in python.

Wordnet links words into semantic relations. (eg. synonyms)

It groups synonyms in the form of synsets.

- synsets : a group of data elements that are semantically equivalent.

Lemmatization Code

```
import nltk
nltk.download('wordnet')
from nltk.stem import WordNetLemmatizer

# Create WordNetLemmatizer object
wnl = WordNetLemmatizer()

# single word lemmatization examples
list1 = ['kites', 'babies', 'dogs', 'flying', 'smiling',
         'driving', 'died', 'tried', 'feet']
for words in list1:
    print(words + " ---> " + wnl.lemmatize(words))

#> kites ---> kite
#> babies ---> baby
#> dogs ---> dog
#> flying ---> flying
#> smiling ---> smiling
#> driving ---> driving
#> died ---> died
#> tried ---> tried
#> feet ---> foot
```

Wordnet Lemmatizer

```
# sentence lemmatization examples
string = 'the cat is sitting with the bats on the striped mat under many flying geese'

# Converting String into tokens
list2 = nltk.word_tokenize(string)
print(list2)
#> ['the', 'cat', 'is', 'sitting', 'with', 'the', 'bats', 'on',
#   'the', 'striped', 'mat', 'under', 'many', 'flying', 'geese']

lemmatized_string = ' '.join([wnl.lemmatize(words) for words in list2])

print(lemmatized_string)
#> the cat is sitting with the bat on the striped mat under many flying goose
```

Wordnet Lemmatizer (with POS tag)

In the above approach, we observed that Wordnet results were not up to the mark. Words like ‘sitting’, ‘flying’ etc remained the same after lemmatization. This is because these words are treated as a noun in the given sentence rather than a verb. To overcome come this, we use POS (Part of Speech) tags.
We add a tag with a particular word defining its type (verb, noun, adjective etc.)
e.g.,

Word	+	Type (POS tag)	—>	Lemmatized Word
driving	+	verb ‘v’	—>	drive
dogs	+	noun ‘n’	—>	dog

Wordnet Lemmatizer

```
# WORDNET LEMMATIZER (with appropriate pos tags)

import nltk
from nltk.stem import WordNetLemmatizer
nltk.download('averaged_perceptron_tagger')
from nltk.corpus import wordnet

lemmatizer = WordNetLemmatizer()

# Define function to lemmatize each word with its POS tag

# POS_TAGGER_FUNCTION : TYPE 1
def pos_tagger(nltk_tag):
    if nltk_tag.startswith('J'):
        return wordnet.ADJ
    elif nltk_tag.startswith('V'):
        return wordnet.VERB
    elif nltk_tag.startswith('N'):
        return wordnet.NOUN
    elif nltk_tag.startswith('R'):
        return wordnet.ADV
    else:
        return None

sentence = 'the cat is sitting with the bats on the striped mat under many badly flying geese'
```

Wordnet Lemmatizer

```
# tokenize the sentence and find the POS tag for each token
pos_tagged = nltk.pos_tag(nltk.word_tokenize(sentence))

print(pos_tagged)
#>[('the', 'DT'), ('cat', 'NN'), ('is', 'VBZ'), ('sitting', 'VBG'), ('with', 'IN'),
# ('the', 'DT'), ('bats', 'NNS'), ('on', 'IN'), ('the', 'DT'), ('striped', 'JJ'),
# ('mat', 'NN'), ('under', 'IN'), ('many', 'JJ'), ('flying', 'VBG'), ('geese', 'JJ')]

# As you may have noticed, the above pos tags are a little confusing.

# we use our own pos_tagger function to make things simpler to understand.
wordnet_tagged = list(map(lambda x: (x[0], pos_tagger(x[1])), pos_tagged))
print(wordnet_tagged)
#>[('the', None), ('cat', 'n'), ('is', 'v'), ('sitting', 'v'), ('with', None),
# ('the', None), ('bats', 'n'), ('on', None), ('the', None), ('striped', 'a'),
# ('mat', 'n'), ('under', None), ('many', 'a'), ('flying', 'v'), ('geese', 'a')]

lemmatized_sentence = []
for word, tag in wordnet_tagged:
    if tag is None:
        # if there is no available tag, append the token as is
        lemmatized_sentence.append(word)
    else:
        # else use the tag to lemmatize the token
        lemmatized_sentence.append(lemmatizer.lemmatize(word, tag))
lemmatized_sentence = " ".join(lemmatized_sentence)

print(lemmatized_sentence)
#> the cat can be sit with the bat on the striped mat under many fly geese
```

TreeTagger for POS and Lemmatization

The TreeTagger is a tool for annotating text with part-of-speech and lemma information. The TreeTagger has been successfully used to tag over 25 languages and is adaptable to other languages if a manually tagged training corpus is available.

Word	POS	Lemma
the	DT	the
TreeTagger	NP	TreeTagger
is	VBZ	be
easy	JJ	easy
to	TO	to
use	VB	use
.	SENT	.

TreeTagger for POS and Lemmatization

```
# 6. TREETAGGER LEMMATIZER
import pandas as pd
import treetaggerwrapper as tt

t_tagger = tt.TreeTagger(TAGLANG ='en', TAGDIR ='C:\Windows\TreeTagger')

pos_tags = t_tagger.tag_text("the bats saw the cats with best stripes hanging upside down by their feet")

original = []
lemmas = []
tags = []
for t in pos_tags:
    original.append(t.split('\t')[0])
    tags.append(t.split('\t')[1])
    lemmas.append(t.split('\t')[-1])

Results = pd.DataFrame({'Original': original, 'Lemma': lemmas, 'Tags': tags})
print(Results)

#>      Original  Lemma Tags      # 7   stripes  stripe  NNS
# 0       the     the   DT       # 8   hanging    hang  VVG
# 1      bats     bat  NNS       # 9    upside  upside   RB
# 2       saw     see  VVD       # 10     down    down   RB
# 3       the     the   DT       # 11       by      by   IN
# 4      cats     cat  NNS       # 12    their   their  PP$
# 5      with    with   IN       # 13     feet    foot  NNS
# 6      best    good  JJS
```

Stemming and lemmatization

The goal of both stemming and lemmatization is to reduce inflectional

forms and sometimes derivationally related forms of a word to a common

base form. For instance

e.g., ***car*** = ***automobile***

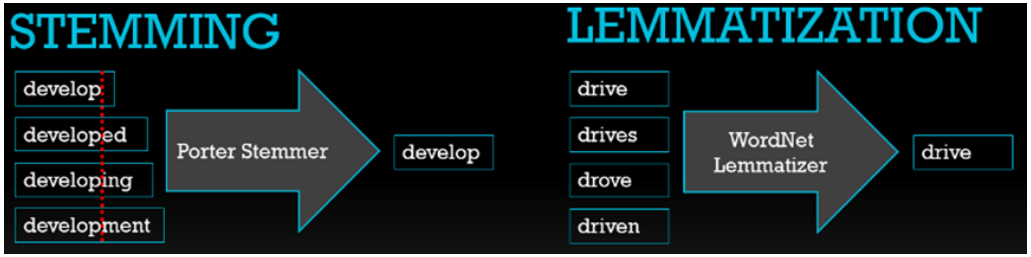
- ***color*** = ***colour***

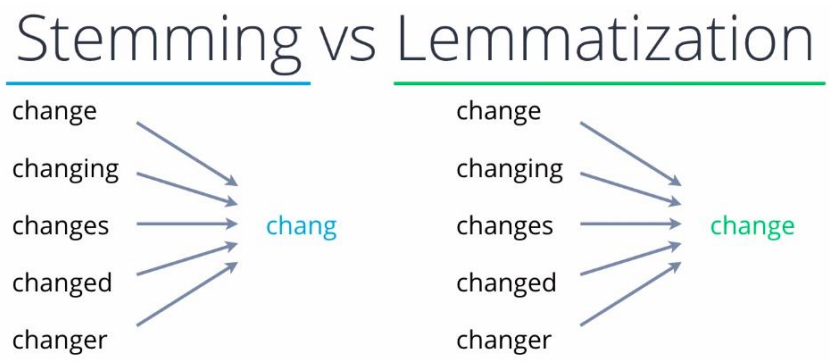
Rewrite to form equivalence classes

Index such equivalences

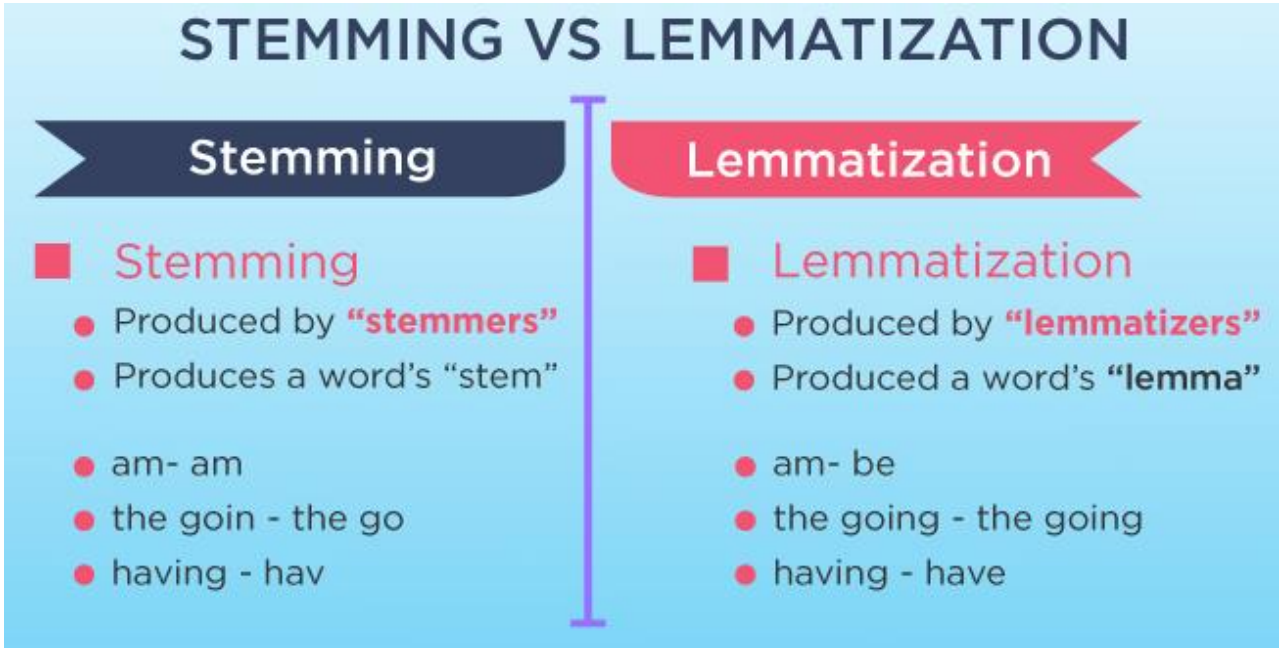
- When the document contains ***automobile***, index it under ***car*** as well (usually, also vice-versa)

Stemming and lemmatization





Stemming and lemmatization



Comparison between Lemmatization and Stemming

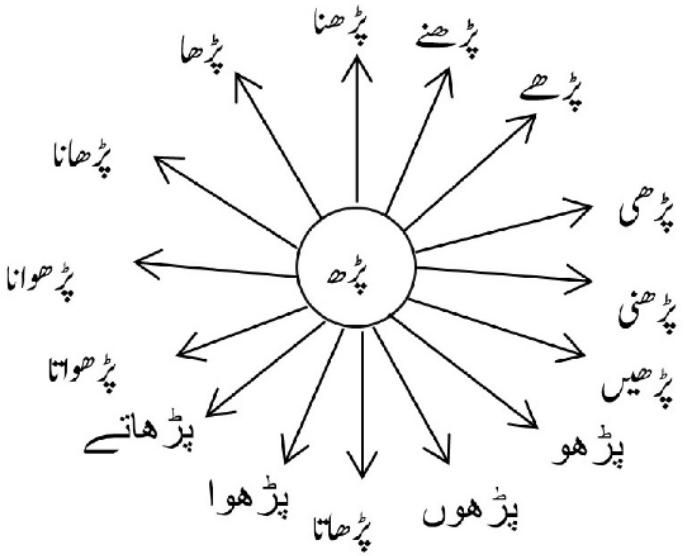
1. ***A stemmer is a rule-based technique, and hence, it is much faster than the lemmatizer.*** However, a stemmer typically gives less accurate results than a lemmatizer.
2. A ***lemmatizer is slower because of the dictionary lookup but gives better results than a stemmer.*** Now, it is important to know that for a lemmatizer to perform accurately, you need to provide the part-of-speech tag of the input word (noun, verb, adjective, etc.). But it is important to note that there are often cases when the POS tagger itself is quite inaccurate on our text and may worsen the performance of the lemmatizer as well. In short, ***we may consider a stemmer if we notice that POS tagging is inaccurate.***

Dealing with complex morphology is necessary for many languages

- e.g., the Turkish, Urdu, Arabic, etc:

No	Query	Query(English)	Query(Roman)	Lemma	Variants
1	یونیورسٹیز	Universities	Universities	یونیورسٹی	یونیورسٹیاں، یونیورسٹیوں، یونیورسٹیز
2	عمارتیں	Buildings	Amarten	عمارت	عمارتوں، عمارات، عمارتیں
3	فائدے	Benefits	Faiday	فائدہ	فائدوں، فوائد، فائدے
4	عدالتیں	Courts	Adalten	عدالت	عدالتوں، عدالتی، عدالتیں

Input Word	Root / Lemma
مہنگائیں	مہنگا
دُکانیں	دُکان
نباتات	نبات
اسلامیات	اسلام




WHAT HAVE
YOU LEARNED?

Recap

In this session we learned about:

- ✓ NLP text preprocessing operations e.g. RE, Normalization etc.
- ✓ Techniques and methods for word, sentence tokenization
- ✓ Stemming and Lemmatization