Text-preprocessing: Tokenization and Stop words removal, Stemming and Lemmatization

INSTRUCTOR: DURESHAHWAR WASEEM

NED UNIVERSITY OF ENGINEERING AND TECHNOLOGY

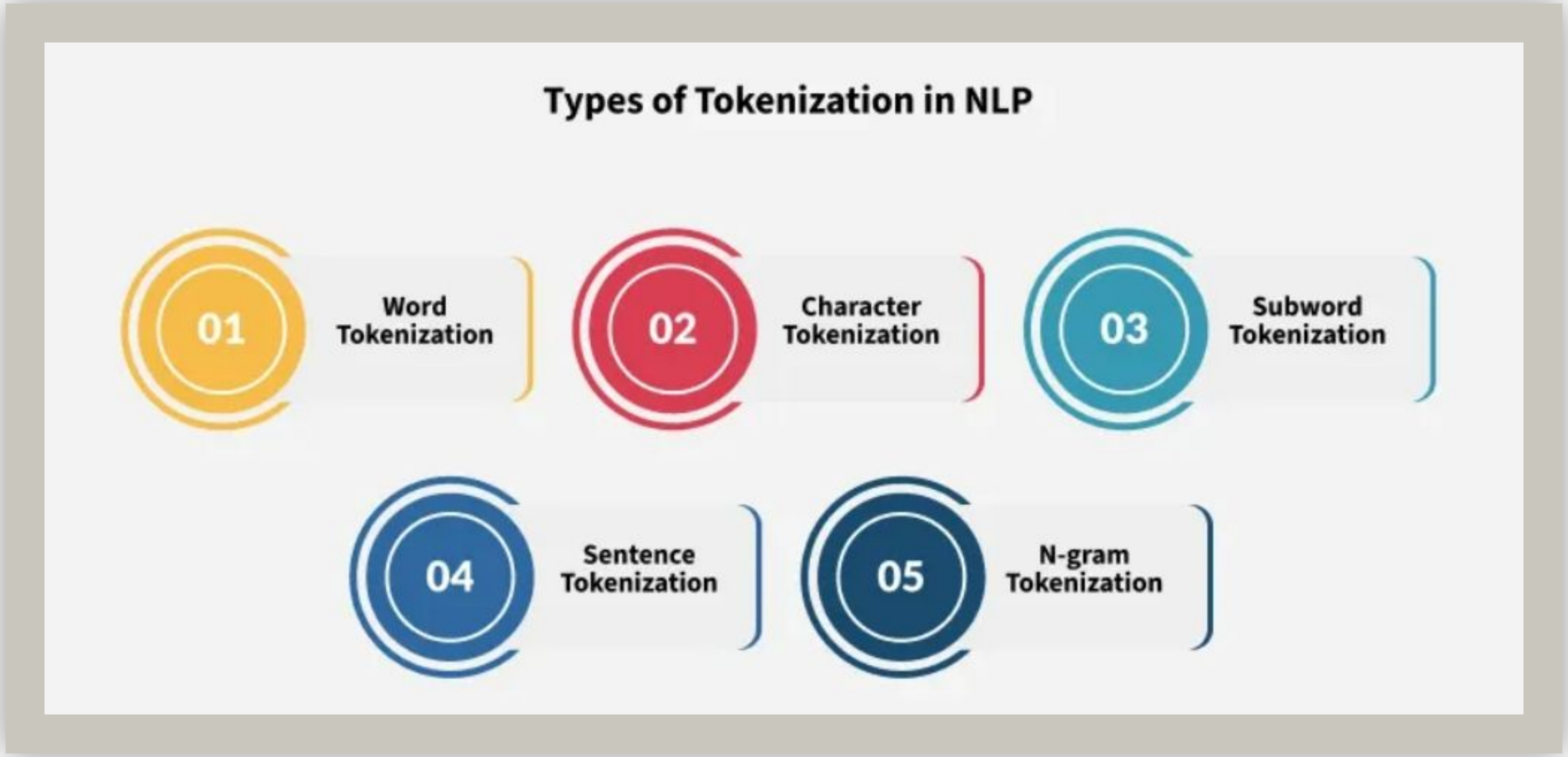
Text preprocessing:

Text preprocessing in Natural Language Processing (NLP) involves a series of steps to clean and prepare raw text data for analysis or model training. Tokenization, stop word removal, stemming, and lemmatization are fundamental techniques within this process.

Tokenization:

- There are roughly two classes of tokenization algorithms. In top-down tokenization, we define a standard and implement rules to implement that kind of tokenization.
- But more commonly instead of using words as the input to NLP algorithms we break up words into sub word tokens, which can be words or parts of words or even individual letters.
- These are derived via bottom-up tokenization, in which we use simple statistics of letter sequences to come up with the vocabulary of subword tokens, and break up the input into those subwords.

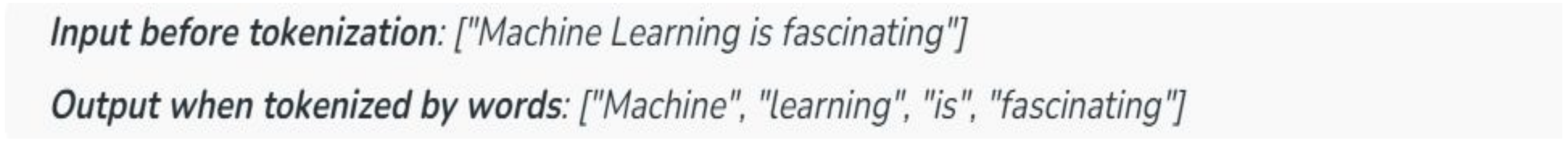
Types of Tokenization:



Types of Tokenization Techniques:

1. Word Tokenization:

- Word tokenization is the most commonly used method where text is divided into individual words. It works well for languages with clear word boundaries, like English. For example, "Machine learning is fascinating" becomes.

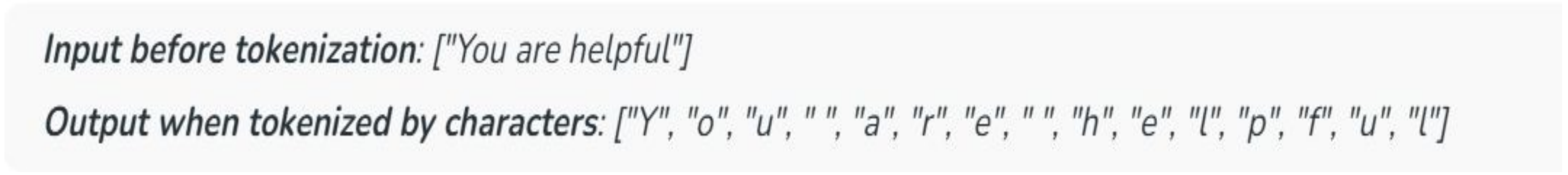


Input before tokenization: *["Machine Learning is fascinating"]*

Output when tokenized by words: *["Machine", "learning", "is", "fascinating"]*

2. Character Tokenization:

In Character Tokenization, the textual data is split and converted to a sequence of individual characters. This is beneficial for tasks that require a detailed analysis, such as spelling correction or for tasks with unclear boundaries. It can also be useful for modelling character-level language.

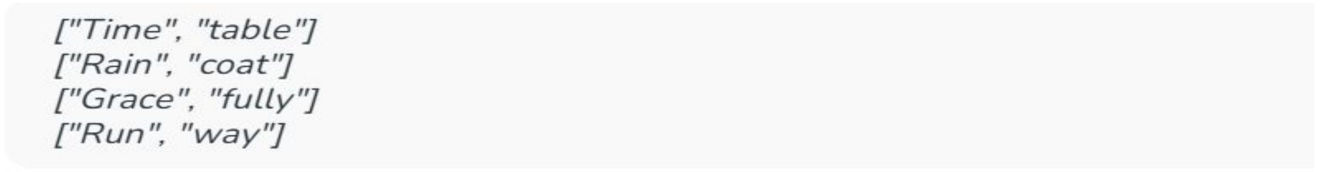


Input before tokenization: *["You are helpful"]*

Output when tokenized by characters: *["Y", "o", "u", " ", "a", "r", "e", " ", "h", "e", "l", "p", "f", "u", "l"]*

3. Sub-word Tokenization

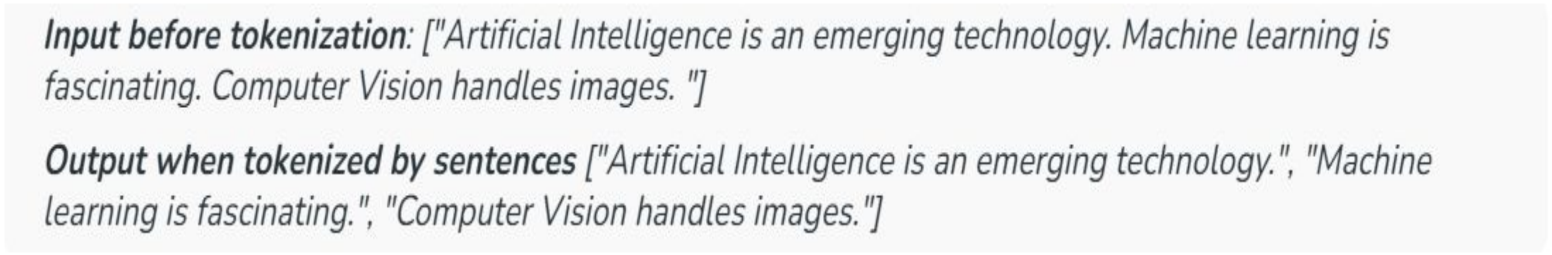
- This strikes a balance between word and character tokenization by breaking down text into units that are larger than a single character but smaller than a full word. This is useful when dealing with morphologically rich languages or rare words.



```
["Time", "table"]
["Rain", "coat"]
["Grace", "fully"]
["Run", "way"]
```

4. Sentence Tokenization

Sentence tokenization is also a common technique used to make a division of paragraphs or large set of sentences into separated sentences as tokens. This is useful for tasks requiring individual sentence analysis or processing.

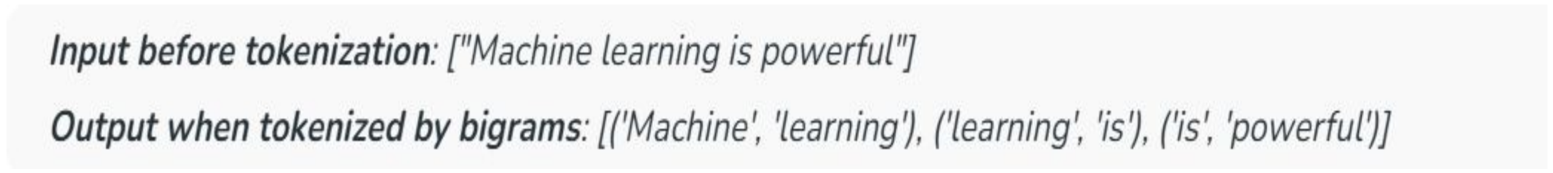


Input before tokenization: *["Artificial Intelligence is an emerging technology. Machine learning is fascinating. Computer Vision handles images. "]*

Output when tokenized by sentences *["Artificial Intelligence is an emerging technology.", "Machine learning is fascinating.", "Computer Vision handles images."]*

5. N-gram Tokenization

- N-gram tokenization splits words into fixed-sized chunks (size = n) of data.

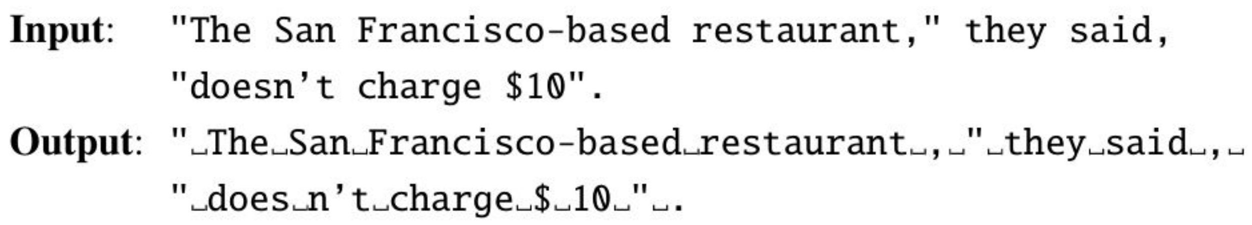


***Input before tokenization**: ["Machine learning is powerful"]*

***Output when tokenized by bigrams**: [('Machine', 'learning'), ('learning', 'is'), ('is', 'powerful')]*

Penn TreeBank:

- One commonly used tokenization standard is known as the Penn Treebank tokenization standard, used for the parsed corpora (treebanks) released by the Lin guistic Data Consortium (LDC), the source of many useful datasets.
- This standard separates out clitics (doesn’t becomes does plus n’t), keeps hyphenated words together, and separates out all punctuation (to save space we’re showing visible spaces ‘ ’ between tokens, although newlines is a more common output):



Input:

```
"The San Francisco-based restaurant," they said,
"doesn’t charge $10".
```

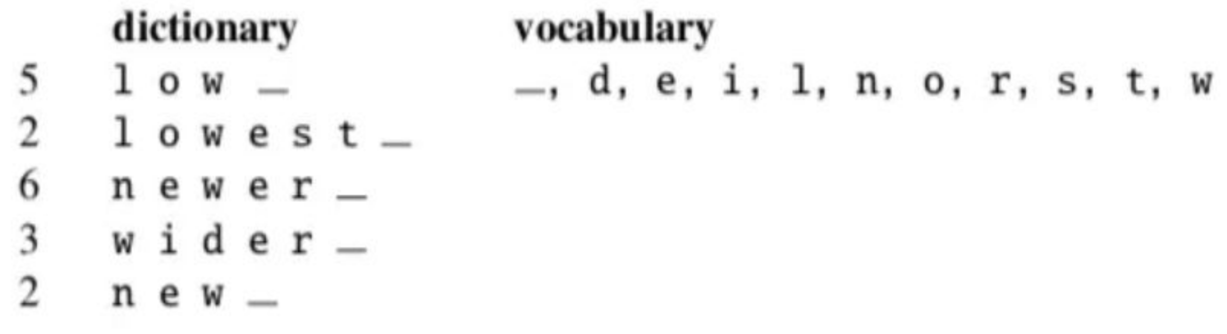
Output:

```
"␣The␣San␣Francisco-based␣restaurant␣,␣"␣they␣said␣,␣
"␣does␣n’t␣charge␣$␣10␣"␣.
```

Advanced Techniques:

Byte-Pair Encoding: A Bottom-up Tokenization Algorithm:

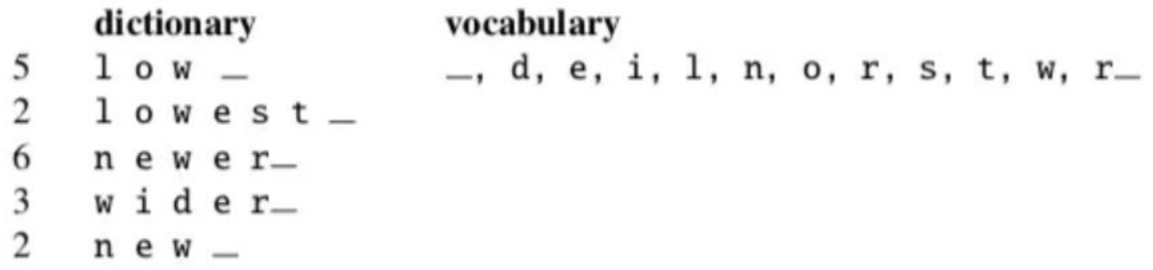
- The BPE token learner begins with a vocabulary that is just the set of all individual characters.
- It then examines the training corpus, chooses the two symbols that are most frequently adjacent (say ‘A’, ‘B’), adds a new merged symbol ‘AB’ to the vocabulary, and replaces every adjacent ’A’ ’B’ in the corpus with the new ‘AB’.
- It continues to count and merge, creating new longer and longer character strings, until k merges have been done creating k novel tokens; k is thus a parameter of the algorithm.
- The resulting vocabulary consists of the original set of characters plus k new symbols.
- The algorithm is usually run inside words (not merging across word boundaries), so the input corpus is first white-space-separated to give a set of strings, each corresponding to the characters of a word, plus a special end-of-word symbol , and its counts.



	dictionary	vocabulary
5	l o w _	_, d, e, i, l, n, o, r, s, t, w
2	l o w e s t _	
6	n e w e r _	
3	w i d e r _	
2	n e w _	

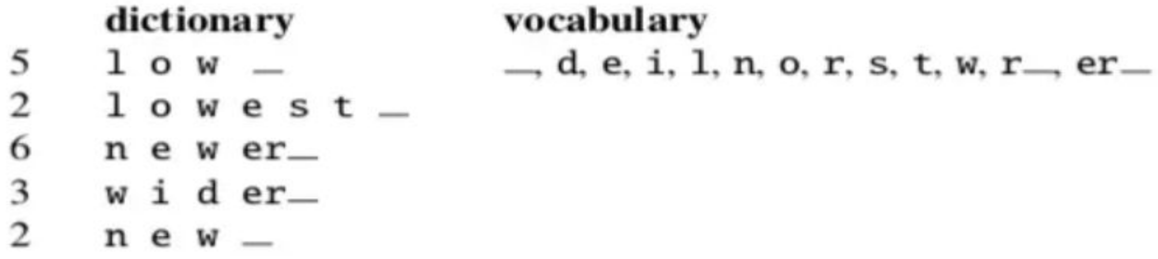
- 1st Column: frequency of the word in the corpus,
- 2nd Column: Dictionary represents the words corresponding to the frequency in the corpus.
- 3rd Column Vocabulary is the set of initial tokens extracted using all unique characters used in the dataset. The end of the sentence is represented by the underscore '_'.
- Now, using the vocabulary, we will iteratively merge tokens to get new tokens depending upon the frequency they are present in the data.

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



	dictionary	vocabulary
5	l o w _	_, d, e, i, l, n, o, r, s, t, w, r_
2	l o w e s t _	
6	n e w e r_	
3	w i d e r_	
2	n e w _	

Now the most frequent pair is er __, which we merge; our system has learned that there should be a token for word-final er, represented as er__:



	dictionary	vocabulary
5	l o w _	_, d, e, i, l, n, o, r, s, t, w, r_, er_
2	l o w e s t _	
6	n e w er_	
3	w i d er_	
2	n e w _	

Next n e(total count of 8)get merged to ne:

corpus		**vocabulary**
5	l o w _	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne
2	l o w e s t _	
6	ne w er_	
3	w i d er_	
2	ne w _	

If we continue, the next merges are:

merge	**current vocabulary**
(ne, w)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new
(l, o)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo
(lo, w)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo, low
(new, er_)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo, low, newer_
(low, _)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo, low, newer_, low_

Token Segmenter:

- Once we’ve learned vocabulary, the token segmenter is used to tokenize a test sentence.
- The token segmenter just runs on the merges we have learned from the training data on the test data. It runs them greedily, in the order we learned them. (Thus the frequencies in the test data don’t play a role, just the frequencies in the training data).
- So first we segment each test sentence word into characters. Then we apply the first rule:
- **Rule 1:**

  Replace every instance of e r in the test corpus with er.
- **Rule 2:**

  Replace every instance of er __ in the test corpus with er__ , and so on.

By the end, if the test corpus contained the character sequence n e w e r __ , it would be tokenized as a full word. But the characters of a new (unknown) word like l o w e r __ would be merged into the two tokens low er__.

Tokenization Challenges:

Ambiguity:

- Language is inherently ambiguous. Consider the sentence "Flying planes can be dangerous." Depending on how it's tokenized and interpreted, it could mean that the act of piloting planes is risky or that planes in flight pose a danger. Such ambiguities can lead to vastly different interpretations.

Languages without clear boundaries:

- Some languages, like Chinese, Japanese, or Thai, lack clear spaces between words, making tokenization more complex. Determining where one word ends and another begins is a significant challenge in these languages.
- To address this, advancements in multilingual tokenization models have made significant strides. For instance:

1. XLM-R (Cross-lingual Language Model - RoBERTa) uses subword tokenization and large-scale pretraining to handle over 100 languages effectively, including those without clear word boundaries.

2. mBERT (Multilingual BERT) employs Word Piece tokenization and has shown strong performance across a variety of languages, excelling in understanding syntactic and semantic structures even in low-resource languages.

Implementing Tokenization:

There are many libraries available, a few are quite popular and help a lot in performing many different NLP tasks.

1. Python: NLTK (Natural Language Toolkit):

- NLTK is a powerful library for natural language processing in Python. It provides a list of stop words for various languages and functions to remove them.

2. Python: spaCy:

- A modern and efficient alternative to NLTK, Spacy is another Python-based NLP library. It boasts speed and supports multiple languages, making it a favorite for large-scale applications.

3. BERT tokenizer:

- Emerging from the BERT pre-trained model, this tokenizer excels in context-aware tokenization. It's adept at handling the nuances and ambiguities of language, making it a top choice for advanced NLP projects.

Tokenization – Python Script:

```
# Run once in your notebook or terminal:
!pip install spacy
!python -m spacy download en_core_web_sm
 import spacy

 nlp = spacy.load("en_core_web_sm")
 doc = nlp("Natural Language Processing is great! Let's explore it.")

 # Sentence segmentation
 sentences = [sent.text for sent in doc.sents]
 print("Sentences:", sentences)
 # → ['Natural Language Processing is great!', "Let's explore it."]

 # Word-level tokens
 words = [token.text for token in doc]
 print("Tokens:", words)
 # → ['Natural', 'Language', 'Processing', 'is', 'great', '!', 'Let', "'s", 'explore', 'it', '.']
```

Output:

```
Sentences: ['Natural Language Processing is great!', "Let's explore it."] Tokens: ['Natural',
'Language', 'Processing', 'is', 'great', '!', 'Let', "'s", 'explore', 'it', '.']
```

Stop Words Removal:

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

Why Remove Stop Words?:

- Stop words are available in abundance in any human language. By removing these words, we remove the low-level information from text in order to give more focus to the important information.
- Removing them helps focus on important words that provide more context.
- It reduces the size of the dataset, speeding up computations.

Example of Stop Words:

- "The cat is on the mat." Stop words: "The", "is", "on", "the"

Stop Words Removal – Python Script:

```
import nltk
nltk.download('punkt')

from nltk.corpus import stopwords
stop_words = set(stopwords.words('english'))
text = "This is a sample sentence“
filtered_words = [word for word in text.split() if
word.lower() not in stop_words]
print(filtered_words)
```

Output:

```
['sample', 'sentence']
```

Stemming:

- Stemming is a process that removes prefixes and suffixes from words, reducing them to their root or stem form. The goal is to standardize words that share the same root but may appear in different forms in the text.

Common Stemming Algorithms:

- **Porter Stemmer**: One of the most widely used stemming algorithms.
- **Lancaster Stemmer**: A more aggressive approach compared to Porter.
- **Snowball Stemmer**: A variation of the Porter Stemmer with support for multiple languages.

Example:

- "running" → "run"
- "flies" → "fli"

Why Use Stemming?

- Reduces variations of the same word to a common root, simplifying analysis.
- Helps in search engines, text classification, and information retrieval by matching different forms of a word.

Stemming – Python Script:

```
from nltk.stem import PorterStemmer

stemmer = PorterStemmer()
print(stemmer.stem("running"))
```

Output:

```
run
```

Lemmatization:

- Lemmatization is a more sophisticated technique compared to stemming. While stemming often produces crude root forms, lemmatization reduces words to their base or dictionary form (lemma), considering the word's meaning and part of speech (POS).

Difference Between Stemming and Lemmatization:

- **Stemming**: Simply removes prefixes/suffixes without considering context. Example: "running" → "run"
- **Lemmatization**: Returns the base form of the word based on its intended meaning. Example: "running" (verb) → "run", "better" (adjective) → "good“

Why Use Lemmatization?

- It produces more meaningful results by returning actual dictionary words.
- It considers the context, such as part of speech, to produce grammatically correct forms.

Lemmatization – Python Script:

```
from nltk.stem import WordNetLemmatizer

lemmatizer = WordNetLemmatizer()
print(lemmatizer.lemmatize("running", pos="v"))
```

Output:

```
run
```

Combining Techniques for Efficient Preprocessing:

- In a real-world NLP pipeline, you’ll often use a combination of these preprocessing techniques to clean and prepare the data before further analysis or modeling. Here's a typical flow for text preprocessing:
- **Tokenization**: Break down the text into words or sentences.
- **Stop Words Removal**: Eliminate common, unimportant words.
- **Stemming or Lemmatization**: Reduce words to their base forms.
- **Further Analysis**: Extract meaningful features or perform sentiment analysis, classification, etc.