

# Understanding the Essentials: NLP Text Preprocessing Steps!



Awaldeep Singh · Follow

8 min read · Dec 31, 2023



Listen



Share

... More

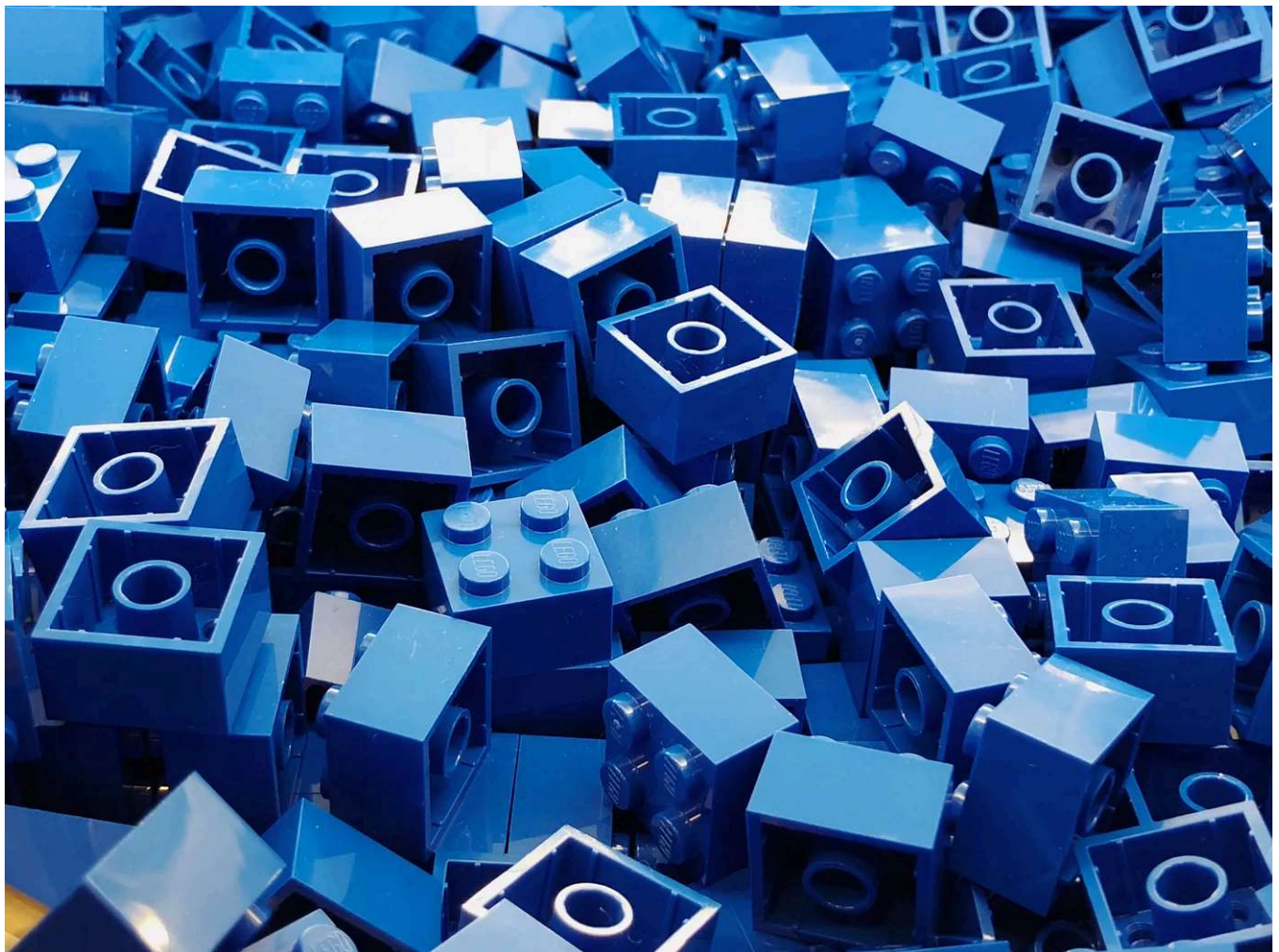


Photo by [Ryan Quintal](#) on [Unsplash](#)

## Introduction

Natural Language Processing (NLP) involves the use of machine learning and computational linguistics to enable computers to understand, interpret, and generate human language. Robust data preprocessing is a crucial step in any NLP

project, as it lays the foundation for effective model training and meaningful insights. In this blog post, we'll explore the essential steps and techniques for data preprocessing in NLP.

Here is a comprehensive list of common techniques and steps involved in text preprocessing:

1. Text lowercasing
2. Tokenization
3. Stop-word removal
4. Handling Numerical values
5. Handling Special characters
6. Whitespace stripping
7. Lemmatization/Stemming
8. Spelling correction
9. Part-of -Speech tagging
10. Handling Contractions
11. Removal of short/rare words
12. Text encoding/ Vectorization
13. Padding/Truncation

These steps are not exhaustive, and the choice of techniques depends on the specific goals of the NLP task in hand. It's common to experiment with different preprocessing steps to determine what works best for the particular use case.

Let's walk through these steps one by one in detail:

### **Lowercasing**

The first step in text preprocessing is converting all text to lowercase. This ensures consistency and reduces the complexity of the text data.

```
text = "Hello World! This is an EXAMPLE."  
lowercased_text = text.lower()  
print(lowercased_text)
```

Output: "hello world! this is an example."

## Tokenization

Tokenization involves breaking the text into individual words or tokens.

```
from nltk.tokenize import word_tokenize  
  
text = "Tokenization is an essential step in NLP."  
tokens = word_tokenize(text)  
print(tokens)
```

Output: ["Tokenization", "is", "an", "essential", "step", "in", "NLP", "."]

There's a detailed [blog](#) about tokenizers and it's HuggingFace 🤗 implementation.

## Stop Word Removal

Eliminate common words (stop words) that do not carry much information.

```
from nltk.corpus import stopwords  
  
stop_words = set(stopwords.words('english'))  
filtered_tokens = [word for word in tokens if word not in stop_words]  
print(filtered_tokens)
```

Output: ["Tokenization", "essential", "step", "NLP"]

## Handling Numerical values

Handling numerical values in text preprocessing typically involves either removing them or transforming them into a standardized representation. This decision depends on the specific requirements of your NLP task. Below I've provided example to remove numeric values from the text.

```
text = "Text preprocessing cleaning and transforming raw text data into a \
format suitable for analysis and machine learning models. \
There are 1234 examples in this text."

import re
text = re.sub(r'\d+', '', text)
print(text) print("\n - -\n")
```

*Output: Text preprocessing cleaning and transforming raw text data into a format suitable for analysis and machine learning models. There are examples in this text.*

### Handling Special Characters

Handling special characters and punctuation is a common step in text preprocessing for NLP. Typically, you may choose to remove them or replace them with a space, depending on your specific requirements. Below code removes punctuation from text.

```
import re

text = "Remove! Punctuation? From, This String."
cleaned_text = re.sub(r'^[\w\s]', '', text)
print(cleaned_text)
```

*Output: Remove Punctuation From This String*

### Whitespace stripping

Stripping or removing leading and trailing white spaces is a common practice in text preprocessing for consistency, easy comparisons, tokenization & storage efficiency:

```
text = " This has extra white spaces. "
text_stripped = text.strip()
print(text_stripped)
```

*Output: This has extra white spaces*

### Lemmatization/Stemming

Lemmatization and stemming are both techniques used in natural language processing (NLP) for reducing words to their base or root form. While they share the goal of simplifying words, they achieve it in slightly different ways.

**Stemming** is a heuristic process that removes suffixes from words to obtain their root form. The resulting stems may not be valid words in the language, but they are intended to represent the core meaning of a word.

**Lemmatization**, on the other hand, involves reducing words to their base or dictionary form, called the lemma. Unlike stemming, lemmatization produces valid words, preserving the grammatical and semantic meaning of the word.

The choice between stemming and lemmatization depends on the specific requirements of your NLP task. In some cases, stemming might be sufficient, while in others, lemmatization might be preferred for more accurate and meaningful results.

Lets understand both with an example:

```
text = "Text preprocessing is an important step in natural language processing.  
It involves cleaning and transforming raw text data into a format \  
suitable for analysis and machine learning models."
```

```
from nltk.tokenize import word_tokenize  
tokens = word_tokenize(text)  
  
from nltk.stem import PorterStemmer  
stemmer = PorterStemmer()  
stemmed_tokens = [stemmer.stem(word) for word in tokens]  
print("Stemming:")  
print(stemmed_tokens)  
  
from nltk.stem import WordNetLemmatizer  
lemmatizer = WordNetLemmatizer()  
lemmatized_tokens = [lemmatizer.lemmatize(word) for word in tokens]  
print("Lemmatization:")  
print(lemmatized_tokens)  
print("\n---\n")
```

Output:

Stemming:

```
['text', 'preprocess', 'is', 'an', 'import', 'step', 'in', 'natur', 'languag', 'process', '.', 'it',
 'involv', 'clean', 'and', 'transform', 'raw', 'text', 'data', 'into', 'a', 'format', 'suitabl', 'for',
 'analysi', 'and', 'machin', 'learn', 'model', '.']
```

Lemmatization:

```
['Text', 'preprocessing', 'is', 'an', 'important', 'step', 'in', 'natural', 'language', 'processing',
 '.', 'It', 'involves', 'cleaning', 'and', 'transforming', 'raw', 'text', 'data', 'into', 'a', 'format',
 'suitable', 'for', 'analysis', 'and', 'machine', 'learning', 'model', '.']
```

Open in app ↗



Search



Spelling correction is an essential step in text preprocessing, and there are various libraries and techniques available to achieve this in Python. One popular library for spelling correction is *'pyspellchecker'*.

```
# ! pip install pyspellchecker -> to install the module

from spellchecker import SpellChecker

text_with_typos = "Ths is an example sentennce with sspellingng mstakes."

# Create a SpellChecker instance
spell = SpellChecker()
# Tokenize the text into words
words = text_with_typos.split()
# Identify misspelled words
misspelled = spell.unknown(words)
# Correct misspelled words
corrected_text = [spell.correction(word) if word in misspelled else word for word in words]
# Join the corrected words back into a string
corrected_text = ' '.join(corrected_text)

print(corrected_text)
```

Output: *Ths is an example sentence with spelling mistakes*



## Part-of-Speech tagging

Part-of-speech tagging (POS tagging) is the process of assigning a specific part of speech (e.g., noun, verb, adjective) to each word in a given text. This is an important step in natural language processing as it helps in understanding the grammatical structure of sentences. In Python, the Natural Language Toolkit (NLTK) provides a convenient way to perform part-of-speech tagging.

Let's generate the POS tags on the text we tokenized earlier:

```
tokens = word_tokenize(text)
pos_tags = pos_tag(tokens)

print(pos_tags)
```

*Output:*

```
[('Text', 'NNP'), ('preprocessing', 'NN'), ('is', 'VBZ'), ('an', 'DT'), ('important', 'JJ'), ('step', 'NN'), ('in', 'IN'), ('natural', 'JJ'), ('language', 'NN'), ('processing', 'NN'), (',', ','), ('It', 'PRP'), ('involves', 'VBZ'), ('cleaning', 'VBG'), ('and', 'CC'), ('transforming', 'VBG'), ('raw', 'JJ'), ('text', 'NN'), ('data', 'NNS'), ('into', 'IN'), ('a', 'DT'), ('format', 'NN'), ('suitable', 'JJ'), ('for', 'IN'), ('analysis', 'NN'), ('and', 'CC'), ('machine', 'NN'), ('learning', 'NN'), ('models', 'NNS'), (',', ',')]
```

## Handling Contractions

Handling contractions is an important step in text preprocessing. Contractions are shortened forms of words or phrases, often formed by combining two words, and they are commonly used in everyday language. Here's an example of how you can handle contractions:

```
contractions_dict = {
    "isn't": "is not",
    "don't": "do not",
    "aren't": "are not",
    "can't": "cannot",
    "couldn't": "could not",
    "didn't": "did not"
} # add more accordingly

text = "I don't know if she's coming. It's a great day, isn't it?"
```

```
for contraction, expansion in contractions_dict.items():
    text = text.replace(contraction, expansion)

print(text)
```

*Output: I do not know if she's coming. It's a great day, is not it?*

### Removal of short/rare words

Rare words, which occur infrequently, might not contribute significantly to the overall understanding of the text and may introduce noise in the analysis. Below example shows how one can remove such words using nltk.

```
from nltk import FreqDist

words = word_tokenize(text)
word_freq = FreqDist(words)
threshold=2
filtered_words = [word for word in words if word_freq[word] > threshold]
```

We can get the frequency of each word in the corpus and remove the rare ones.

### Text encoding/ Vectorization

Text encoding or vectorization is the process of converting textual data into numerical representations that can be fed into machine learning models. There are various techniques for text encoding, and one common approach is using vectorization methods. Two popular vectorization techniques are Bag of Words (BoW) and Term Frequency-Inverse Document Frequency (TF-IDF).

```
from sklearn.feature_extraction.text import TfidfVectorizer

# Sample documents
documents = [
    "This is the first document.",
    "This document is the second document.",
    "And this is the third one.",
    "Is this the first document?",
]

# Create a TF-IDF vectorizer
vectorizer = TfidfVectorizer()
```



```
# Fit and transform the documents
tfidf_matrix = vectorizer.fit_transform(documents)

# Get feature names (words) from the vectorizer
feature_names = vectorizer.get_feature_names_out()

# Convert the TF-IDF matrix to a dense array and display the results
dense_array = tfidf_matrix.toarray()
print("TF-IDF Matrix:")
print(dense_array)

# Display feature names
print("\nFeature Names:")
print(feature_names)
```

## ## OUTPUT

```
TF-IDF Matrix:
[[0.43877674 0.54197657 0.43877674 0.35872874 0.43877674 0.54197657]
 [0.43877674 0.54197657 0.43877674 0.35872874 0.43877674 0.54197657]
 [0.43877674 0.54197657 0.43877674 0.35872874 0.43877674 0.54197657]
 [0.64781454 0.          0.64781454 0.52547275 0.          0.          ]]

Feature Names:
['and', 'document', 'first', 'is', 'one', 'second']
```

In this TF-IDF matrix, each row corresponds to a document, and each column corresponds to a unique word (feature). The values in the matrix represent the TF-IDF score of each word in each document.

## Padding/Truncation

Padding and truncation are common techniques used in text preprocessing, particularly when dealing with sequences of variable lengths. These techniques are often applied before feeding the data into machine learning models that expect fixed-length input sequences. Padding involves adding zeros or a special token to the sequences to make them a consistent length, while truncation involves removing elements from sequences that exceed a certain length.

```
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
```

```
# Sample texts
texts = [
    "This is the first document.",
    "This document is the second document.",
    "And this is the third one.",
    "Is this the first document?",
]

# Create a tokenizer
tokenizer = Tokenizer()
tokenizer.fit_on_texts(texts)

# Convert texts to sequences
sequences = tokenizer.texts_to_sequences(texts)

# Define a maximum sequence length
max_length = 10

# Perform padding or truncation
padded_sequences = pad_sequences(sequences, maxlen=max_length, padding='post',

# Display the results
print("Original Sequences:")
print(sequences)
print("\nPadded/Truncated Sequences:")
print(padded_sequences)
```

In this example:

- The `Tokenizer` class from Keras is used to convert the texts to sequences of integers.
- The `pad_sequences` function is then used to perform padding or truncation on the sequences.
- The `maxlen` parameter defines the maximum length of the sequences.
- The `padding` parameter specifies whether to add padding at the beginning ('pre') or the end ('post') of the sequences.
- The `truncating` parameter specifies whether to truncate from the beginning ('pre') or the end ('post') of the sequences if they exceed the specified length.

## ## OUTPUT

Original Sequences:

```
[[1, 2, 3, 4, 5, 6], [1, 2, 3, 4, 2, 3, 7, 6], \  
[8, 1, 2, 3, 4, 5, 9, 6], [10, 1, 2, 3, 4, 5, 6]]
```

Padded/Truncated Sequences:

```
[[ 1  2  3  4  5  6  0  0  0  0]  
[ 1  2  3  4  2  3  7  6  0  0]  
[ 8  1  2  3  4  5  9  6  0  0]  
[10  1  2  3  4  5  6  0  0  0]]
```

In the padded/truncated sequences, zeros are added at the end ('post') to make all sequences of length 10.

## Conclusion

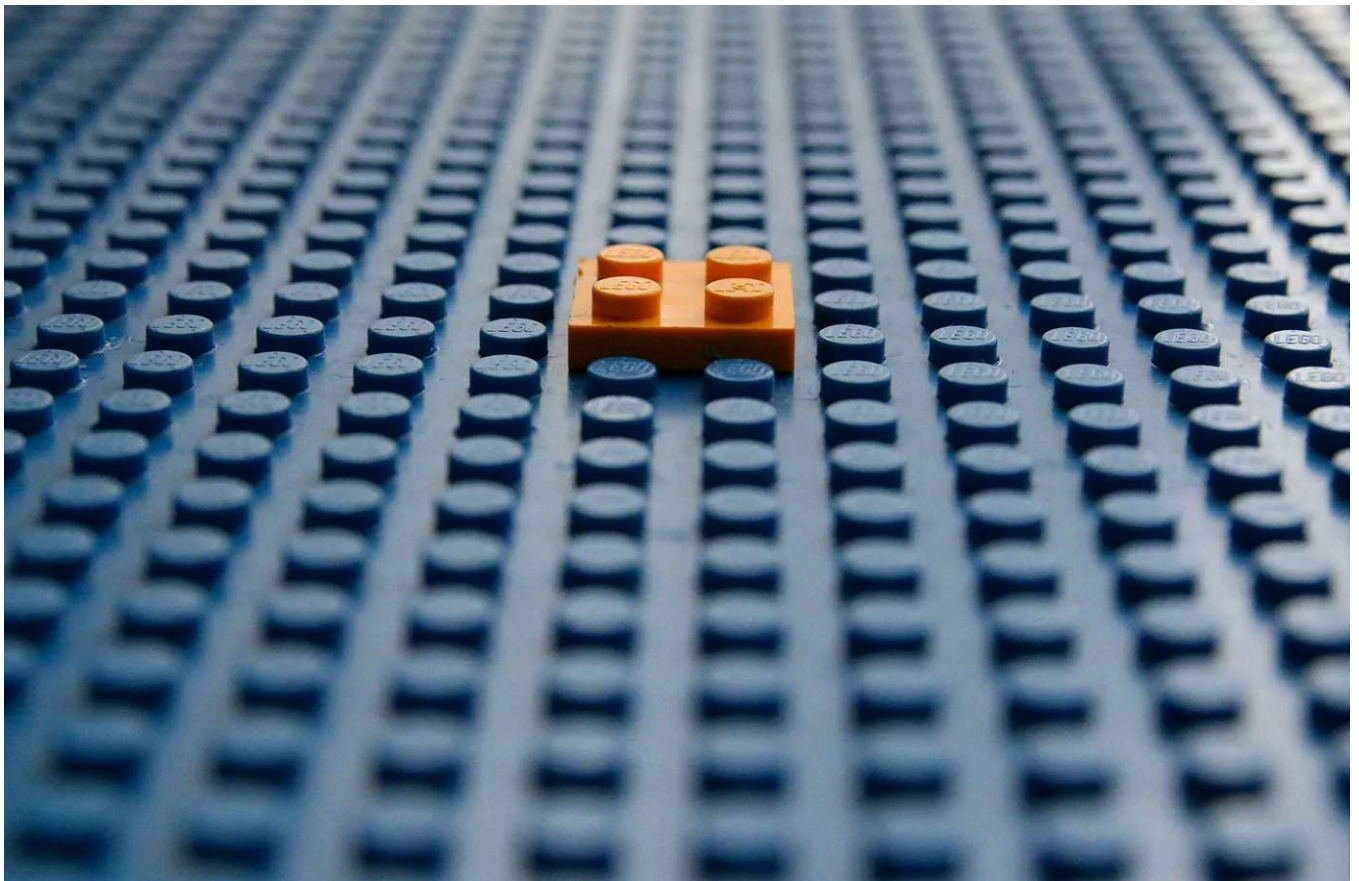


Photo by [Glen Carrie](#) on [Unsplash](#)

Mastering the essentials of Natural Language Processing (NLP) text preprocessing is a pivotal step towards extracting meaningful insights from unstructured text data. In this blog, we've delved into various indispensable steps that lay the foundation for robust NLP models. From handling contractions to removing rare words and

employing powerful techniques like lemmatization and part-of-speech tagging, each preprocessing step contributes to the refinement and normalization of textual information.

The significance of these preprocessing steps becomes evident in their ability to enhance the efficiency and effectiveness of downstream NLP tasks. By addressing challenges such as noisy data, variations in language structure, and the presence of irrelevant information, we pave the way for more accurate analyses, sentiment predictions, and information extractions.

Happy learning!

NLP

Python

Machine Learning

Text Preprocessing

Data Science



Follow

## Written by Awaldeep Singh

24 Followers

Data Scientist @ YABX | <https://www.linkedin.com/in/awaldeep/>

---

### More from Awaldeep Singh






Introduction

8 min read · Jul 10, 2023

 9

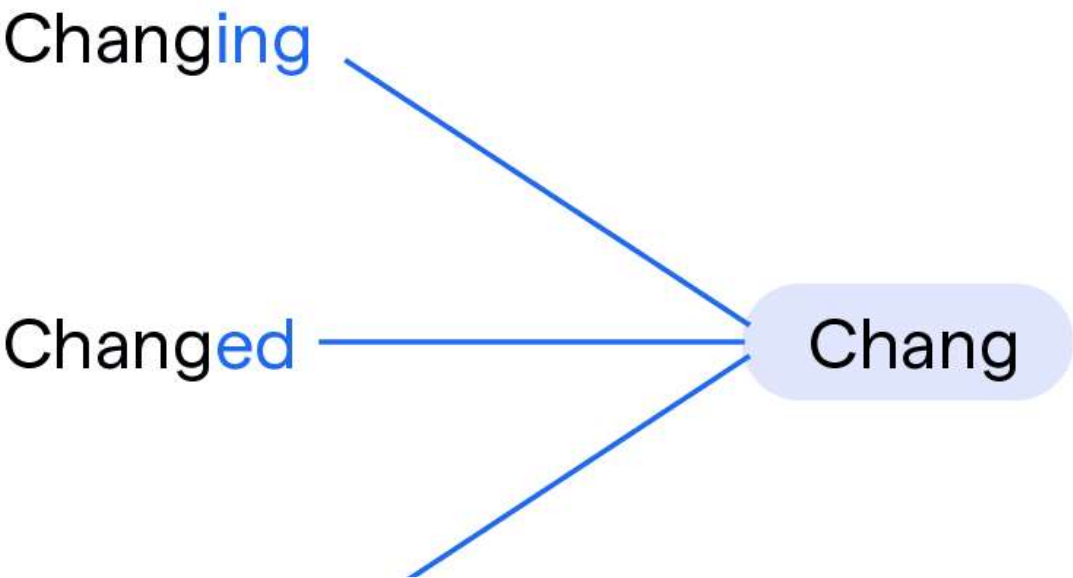






See all from Awaldeep Singh

Recommended from Medium



 Chandu Aki in The Deep Hub

NLP—Text PreProcessing—Part 3 (Stemming & Lemmatization)

NLP—Text Preprocessing—Part3

4 min read · Feb 16, 2024

 31

 1







Display options for sense: (gloss) "an example sentence"

## Noun

- **S: (n) dog, domestic dog, *Canis familiaris*** (a member of the genus *Canis* (probably descended from the common wolf) that has been domesticated by man since prehistoric times; occurs in many breeds) *"the dog barked all night"*
- **S: (n) frump, dog** (a dull unattractive unpleasant girl or woman) *"she got a reputation as a frump"; "she's a real dog"*
- **S: (n) dog** (informal term for a man) *"you lucky dog"*
- **S: (n) cad, bounder, blackguard, dog, hound, heel** (someone who is morally reprehensible) *"you dirty dog"*
- **S: (n) frank, frankfurter, hotdog, hot dog, dog, wiener, wienerwurst, weenie** (a smooth-textured sausage of minced beef or pork usually smoked; often served on a bread roll)
- **S: (n) pawl, detent, click, dog** (a hinged catch that fits into a notch of a ratchet to move a wheel forward or prevent it from moving backward)
- **S: (n) andiron, firedog, doa, doa-iron** (metal supports for logs in a fireplace) *"the*



om pramod

## WordNet in Action: Enhancing NLP Capabilities Through NLTK

Words might seem distinct at first glance but often share similarities within certain contexts. Consider the words "big" and "large." At a...

2 min read · Dec 14, 2023



609



## Lists



### Predictive Modeling w/ Python

20 stories · 1161 saves



### Practical Guides to Machine Learning

10 stories · 1402 saves



### Coding & Development

11 stories · 596 saves



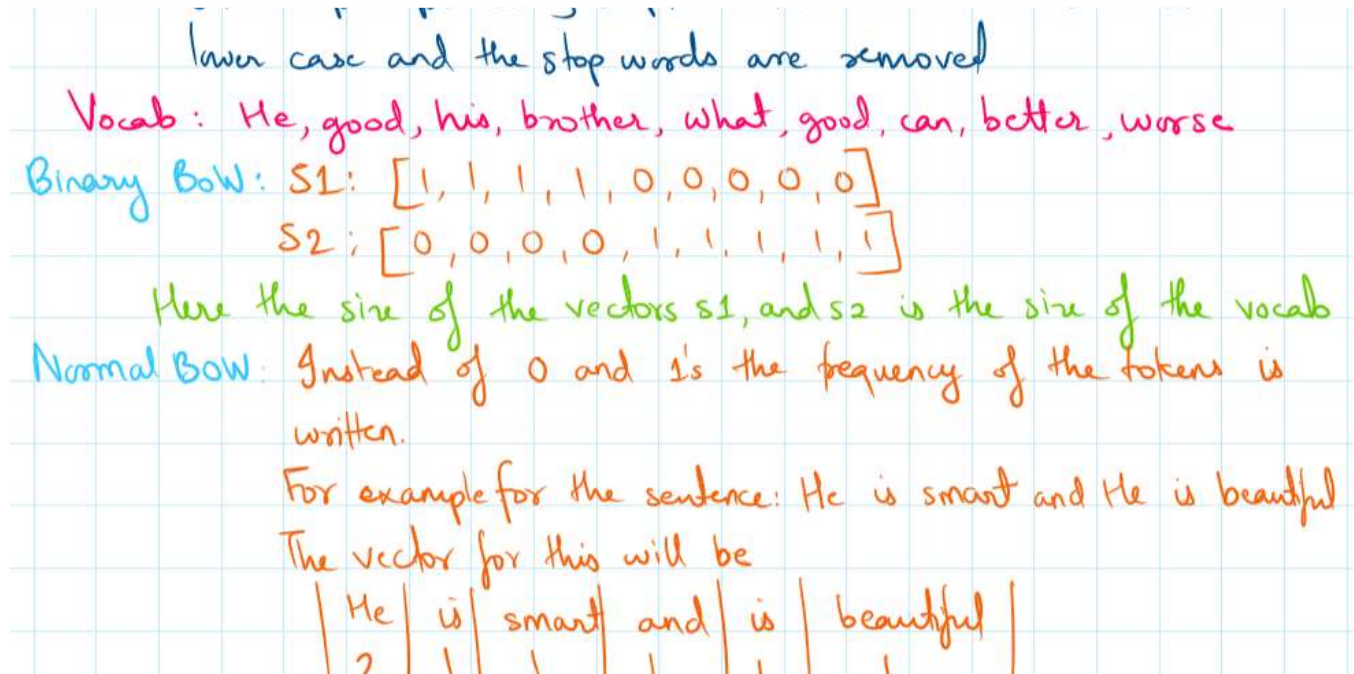
### Natural Language Processing

1434 stories · 930 saves



## Accessing Text Corpora and Lexical Resources

9 min read · 3 days ago



Abhishek Jain

## Bag of Words (BoW) in NLP

Bag of Words represents unstructured textual data into structured and numerical format, making it suitable for various NLP applications

3 min read · Feb 2, 2024





aditya goel

## Sentiment Classification in NLP | Part-1 | Features Extraction

If you are landing here directly, it may be good to read through this blog first.

10 min read · Feb 8, 2024



See more recommendations