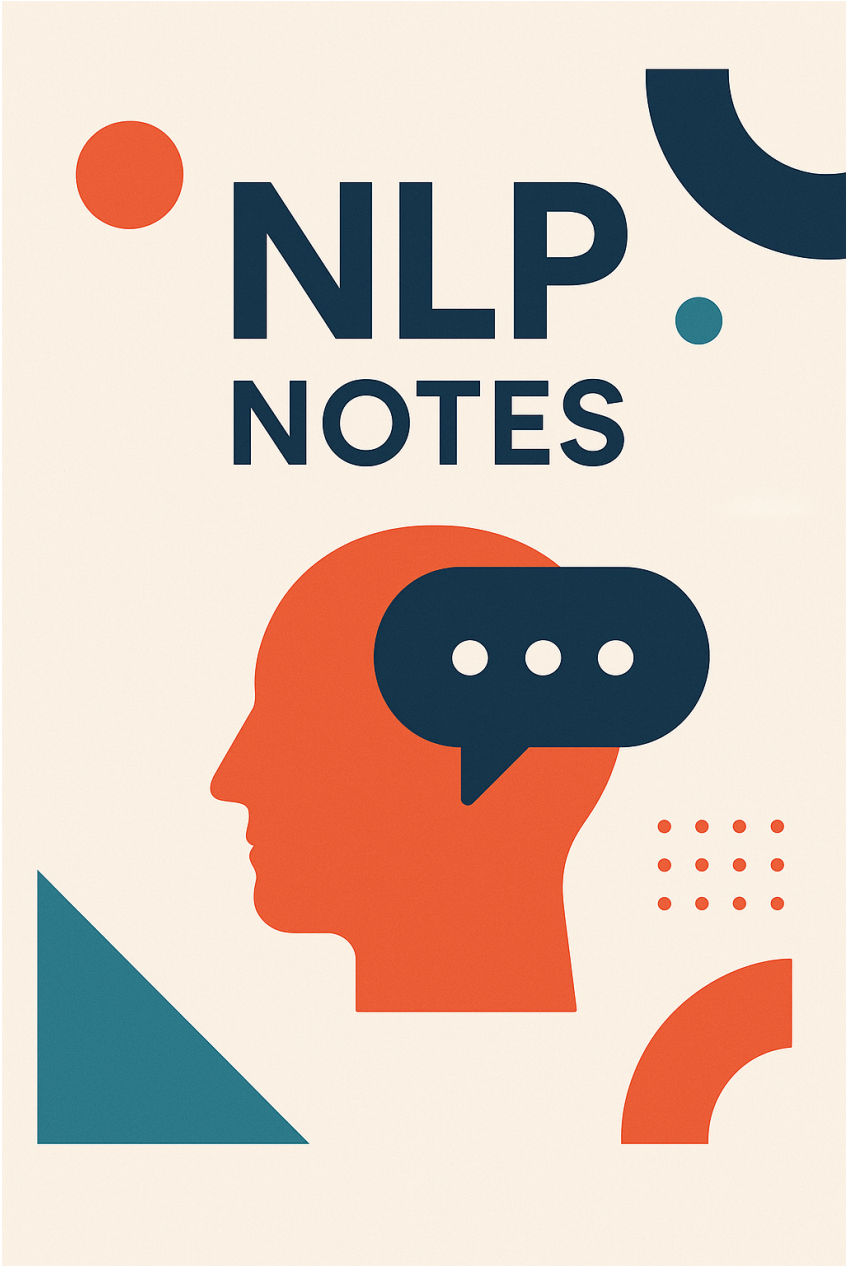
Natural Language Processing Fundamentals

Minor in AI

May 1, 2025



1. The Power of Understanding Language

Imagine you are developing a chatbot that helps students get answers to campus queries. The chatbot must understand sentences like: *"When is the next AI class?"* and *"I want to register for the NLP elective."* To build such a bot, we must understand how machines interpret language. This is where Natural Language Processing (NLP) steps in. Let's walk through the journey of NLP like you're building that chatbot, learning everything from scratch.

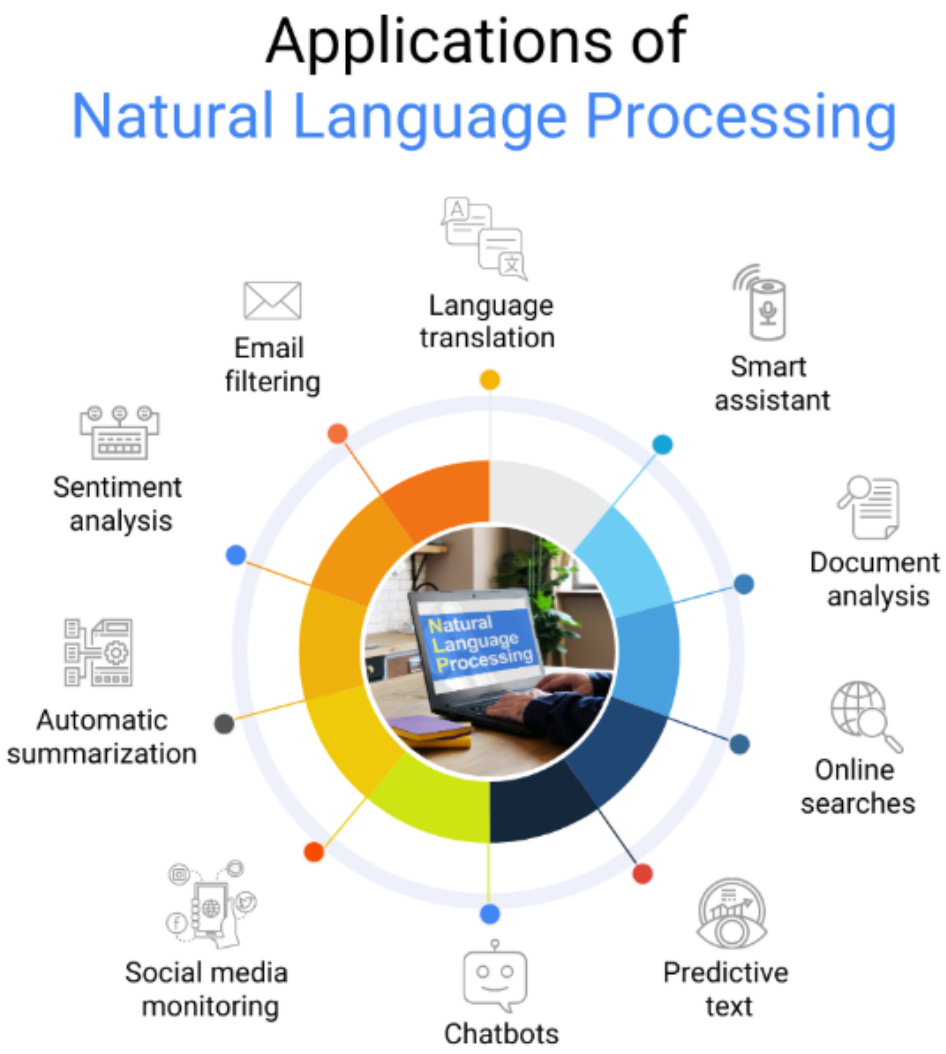


Figure 1: Common applications of NLP in modern technology

In this lecture, the instructor introduces several fundamental NLP concepts, each playing a critical role in transforming raw text into structured data that machine learning models can process. These concepts include **tokenization, n-grams, part-of-speech tagging, named entity recognition, stop word removal, stemming, and lemmatization**. Understanding these techniques is essential for building robust NLP systems, as they form the backbone of text preprocessing and feature extraction.

2. NLP Concepts

The Librarian of Words

A librarian named Tia had a special task. She was in charge of maintaining a magical library where instead of books, sentences were stored. But these sentences couldn't be stored as-is—they had to be broken down into meaningful units so that the library could properly index and understand them. For every student who came in and said something like, *"I love NLP."*, she would carefully break it down into its building blocks: ["I", "love", "NLP", "."]. This act of breaking down a sentence into smaller components is called **tokenization**.

Tokenization is the most fundamental step in NLP. It is like creating LEGO bricks from a full structure, enabling the machine to process, manipulate, and analyze individual components. Every word, punctuation, or even symbol is identified and isolated. This helps further stages like POS tagging, parsing, and sentiment analysis work effectively.

Tokenization not only helps in processing texts but also in preserving meaning. For example, "Mr." should be preserved as a single token in some contexts, while other words need to be split. A well-designed tokenizer takes all of this into account.

Tokenization

Tokenization is the process of breaking down text into smaller units called tokens, which can be words, sentences, or even subwords. Think of it like dissecting a sentence into individual building blocks—each word or punctuation mark becomes a separate token. This step is crucial because raw text is unstructured, and machines need discrete units to analyze language.

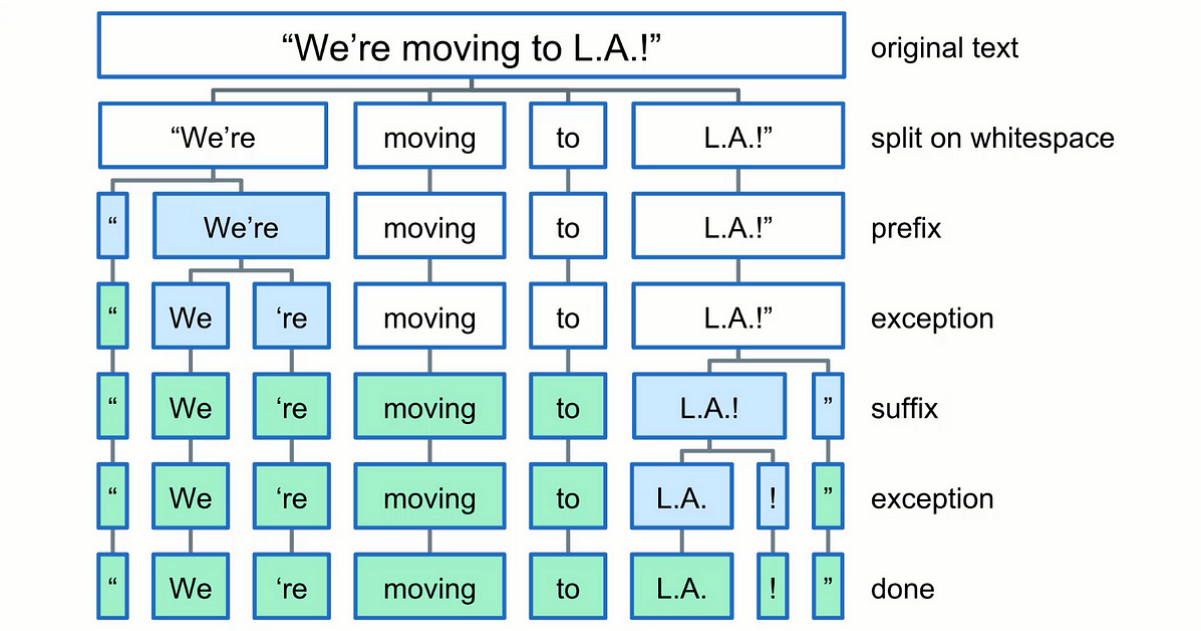


Figure 2: Visual representation of word and sentence tokenization

For example, the sentence *"I love NLP!"* would be tokenized into `["I", "love", "NLP", "!"]`. Tokenization helps in further processing, such as counting word frequencies, identifying parts of speech, or training machine learning models.

There are different types of tokenization:

- **Word Tokenization:** Splits text into individual words.
- **Sentence Tokenization:** Divides paragraphs into sentences.
- **Subword Tokenization:** Used in advanced models (e.g., BERT) to handle rare words by breaking them into smaller units (e.g., "unhappiness" → "un", "happiness").

Listing 1: Python Example of Tokenization

```
import spacy

nlp = spacy.load("en_core_web_sm")
text = "Tokenization␣splits␣text␣into␣tokens."
doc = nlp(text)

for token in doc:
    print(token.text)
```

Output:

```
Tokenization
splits
text
into
tokens
.
```

Discussion:

- The `spacy` library processes text into tokens, including punctuation.
- Each token is an object with properties like `token.text` (the word itself), `token.is_alpha` (checks if it's alphabetic), and `token.is_punct` (checks if it's punctuation).

Understanding Movie Sentiment

Imagine you're analyzing movie reviews. A student writes: *"I did not like the movie."* A naive model using just single words (called **unigrams**) might mistakenly think it's a positive review—after all, the word *like* is present, which often indicates a good opinion.

But what the model misses is the crucial context created by combining consecutive words. The phrase *"not like"* flips the sentiment completely. This is where **N-gram** models—especially **bigrams** (pairs of consecutive words)—become powerful. They help preserve the context and meaning of expressions by looking beyond individual words.

In an N-gram approach, instead of seeing words in isolation, we treat groups of N words together.

N-grams (Unigrams, Bigrams, Trigrams)

N-grams are contiguous sequences of n words extracted from a text. They help capture context and word relationships, which is essential for tasks like language modeling, text generation, and sentiment analysis.

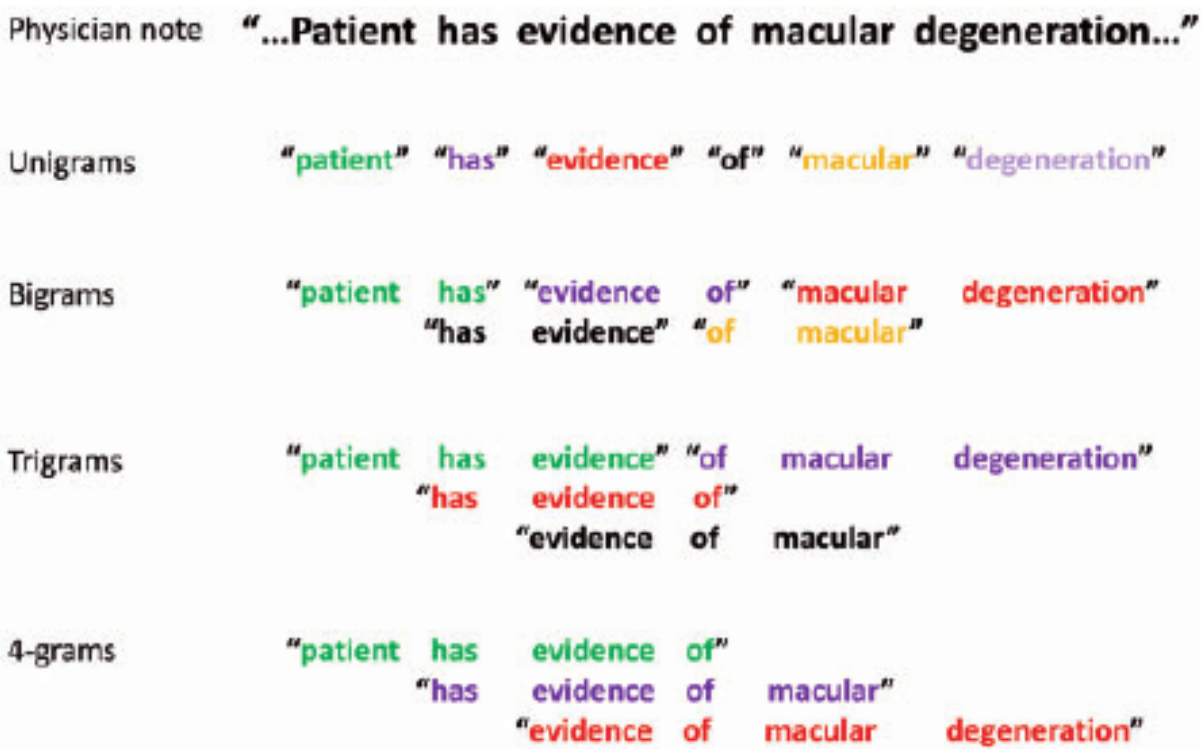


Figure 3: Comparison of unigrams, bigrams and trigrams

- **Unigrams:** Single words (e.g., "I", "love", "NLP").
- **Bigrams:** Pairs of consecutive words (e.g., "I love", "love NLP").
- **Trigrams:** Three-word sequences (e.g., "I love NLP").

Why Use N-grams?

- They help in understanding context. For example, "not good" conveys negativity, whereas analyzing "not" and "good" separately loses meaning.
- They improve language models by predicting the next word in a sequence (e.g., autocomplete suggestions).

Mathematics Behind N-grams: The probability of a sentence using bigrams is calculated as:

$$P(w_1, w_2, \ldots, w_n) \approx \prod_{i=1}^{n} P(w_i|w_{i-1})$$

Where:

- $P(w_i|w_{i-1})$ is the probability of word w_i given the previous word w_{i-1}.

- This is estimated using counts from a corpus:

$$P(w_i|w_{i-1}) = \frac{\text{Count}(w_{i-1}, w_i)}{\text{Count}(w_{i-1})}$$

Listing 2: Python Example of N-grams

```
from nltk import ngrams

text = "I love NLP and machine learning."
words = text.split()

unigrams = words
bigrams = list(ngrams(words, 2))
trigrams = list(ngrams(words, 3))

print("Unigrams:", unigrams)
print("Bigrams:", bigrams)
print("Trigrams:", trigrams)
```

Output:

```
Unigrams: ['I', 'love', 'NLP', 'and', 'machine', 'learning.']
Bigrams: [('I', 'love'), ('love', 'NLP'), ('NLP', 'and'),
          ('and', 'machine'), ('machine', 'learning.')]
Trigrams: [('I', 'love', 'NLP'), ('love', 'NLP', 'and'),
           ('NLP', 'and', 'machine'), ('and', 'machine', 'learning.')]
```

Discussion:

- Bigrams and trigrams capture meaningful phrases ("machine learning") that unigrams miss.
- The downside is that vocabulary size grows exponentially with higher n, increasing computational cost.

The Grammar Wizard

Meet Arya, a grammar wizard. She has an incredible ability: she can listen to any sentence and immediately tell whether each word is a noun, a verb, an adjective, or something else entirely. This magical skill helps her understand sentence structure and meaning with ease. In the world of NLP, this ability is called **Part of Speech (POS) tagging**.

Just like Arya, an NLP model equipped with POS tagging can dissect a sentence and label each word with its grammatical role. This is incredibly useful for downstream tasks like parsing, sentiment analysis, named entity recognition, and machine translation. For instance, knowing that *"love"* is a verb in the sentence *"I love NLP."* allows the model to better understand the sentence's intention and structure.

POS tagging assigns labels such as NOUN, VERB, ADJ (adjective), ADV (adverb), etc., to each token. These labels help computers understand how words relate to each other within a sentence.

Part-of-Speech (POS) Tagging

Part-of-speech tagging assigns grammatical categories (e.g., noun, verb, adjective) to each word in a sentence. This helps in understanding sentence structure, improving parsing, and enabling advanced NLP tasks like named entity recognition.

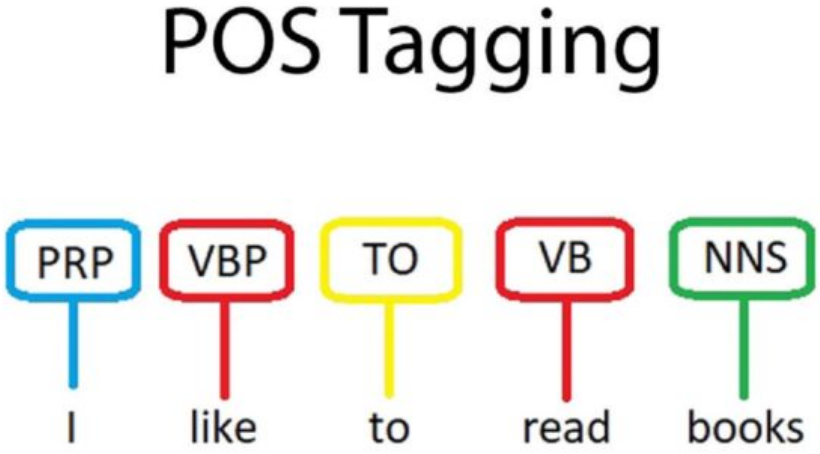


Figure 4: Example of POS tagging in a sentence

Common POS Tags in Spacy:

Tag	Meaning	Example
NOUN	Noun	”cat”, ”book”
VERB	Verb	”run”, ”eating”
ADJ	Adjective	”happy”, ”blue”
ADV	Adverb	”quickly”, ”very”
PRON	Pronoun	”I”, ”they”
PROPN	Proper Noun	”John”, ”Google”

Table 1: Common POS tags and their meanings

Listing 3: Python Example of POS Tagging

```
doc = nlp("I␣love␣NLP␣and␣machine␣learning.")
for token in doc:
    print(token.text, token.pos_, spacy.explain(token.pos_))
```

Output:

```
I PRON pronoun
love VERB verb
NLP PROPN proper noun
and CCONJ coordinating conjunction
machine NOUN noun
```

```
learning NOUN noun
. PUNCT punctuation
```

Discussion:

- `token.pos_` provides coarse-grained tags (e.g., "VERB" for all verbs)
- `token.tag_` gives fine-grained tags (e.g., "VBZ" for third-person singular verbs like "runs")
- POS tags are essential for understanding grammatical relationships between words

Named Entity Recognition (NER)

Named Entity Recognition identifies and classifies key elements in text, such as:

- **Persons (PER):** "Elon Musk"
- **Organizations (ORG):** "Tesla"
- **Locations (LOC):** "New York"
- **Dates (DATE):** "January 2023"

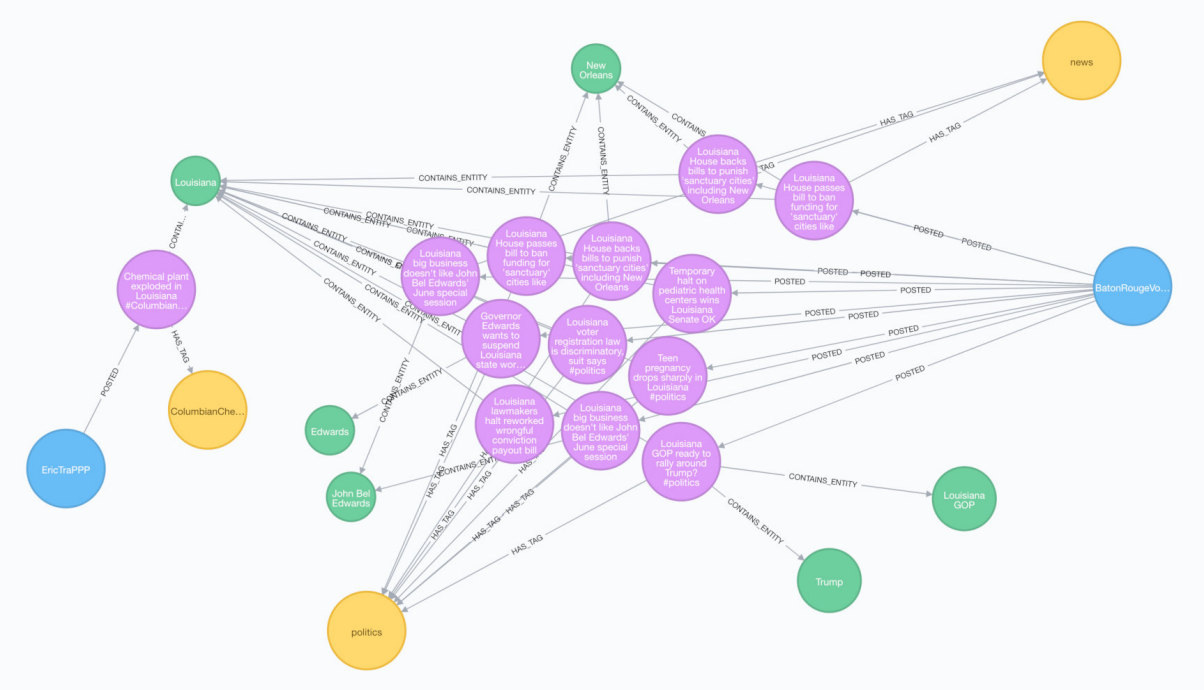


Figure 5: NER visualization showing identified entities

Applications:

- Extracting company names from financial reports
- Identifying locations in travel-related queries
- Filtering news articles by entities (e.g., "Show me articles about Apple")

Listing 4: Python Example of NER

```
doc = nlp("Tesla␣acquired␣Twitter␣for␣$45␣billion.")
for ent in doc.ents:
    print(ent.text, ent.label_, spacy.explain(ent.label_))
```

Output:

```
Tesla ORG Companies, agencies, institutions
Twitter ORG Companies, agencies, institutions
$45 billion MONEY Monetary values
```

Discussion:

- NER models sometimes misclassify entities (e.g., "X" may not be recognized as an organization)
- Custom NER models can be trained for domain-specific terms (e.g., medical entities in healthcare texts)
- NER performance depends heavily on the training data and domain

The Word Dietician

Meet Ravi, the word dietician. Ravi isn't concerned about calorie intake, but rather about cutting down the bulk from text data. In the world of Natural Language Processing, a large portion of words in a text don't contribute much meaning when it comes to machine learning tasks. Words like *"is"*, *"the"*, *"an"*, *"in"*, and *"of"* occur frequently across texts but carry very little semantic weight.

These commonly occurring, low-value words are called **stopwords**. Removing them from text helps reduce the dimensionality of the vocabulary, speeds up processing, and often improves the performance of NLP models by eliminating noise. Imagine trying to understand the gist of a news article — you'd probably skip over these filler words subconsciously. Machines can do the same!

In NLP pipelines, stopword removal is typically a preprocessing step. Libraries like spaCy and NLTK come with precompiled lists of stopwords that can be easily leveraged to filter out these non-informative words.

Stop Word Removal

Stop words are common words (e.g., "the", "is", "and") that add little semantic value. Removing them reduces noise and improves computational efficiency.

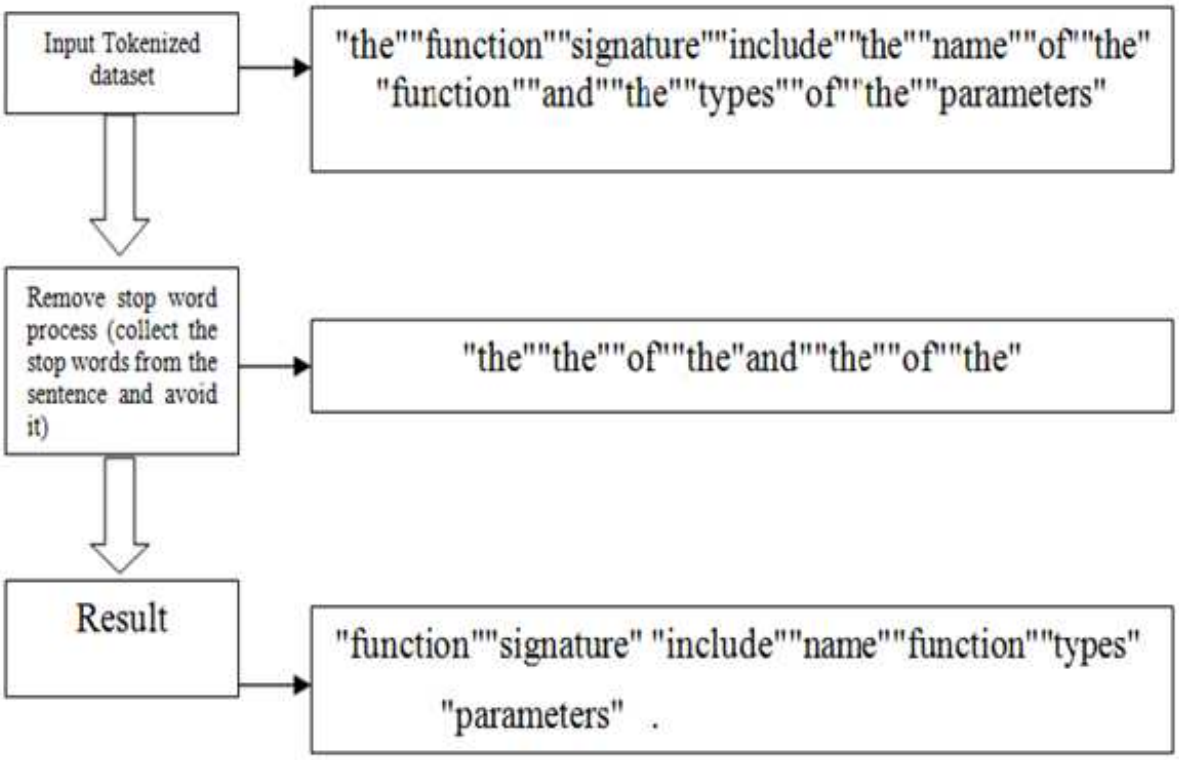


Figure 6: Example of stop word removal process

Why Remove Stop Words?

- They dominate word frequency counts without adding meaning
- They are often irrelevant in search engines and topic modeling
- Reduces dimensionality of text data

Listing 5: Python Example of Stop Word Removal

```
from spacy.lang.en.stop_words import STOP_WORDS

text = "This is an example sentence with stop words."
filtered_words = [word for word in text.split()
                  if word.lower() not in STOP_WORDS]
print(filtered_words)
```

Output:

```
['example', 'sentence', 'stop', 'words.']
```

Discussion:

- Some stop words can be meaningful (e.g., "not" in sentiment analysis)
- Custom stop word lists can be created for specialized applications
- The choice of stop words depends on the specific NLP task

Word Family Reunion

Think of a word family that includes "run", "running", "ran", and "runs". Even though these words appear in different forms, they all convey the same fundamental idea — the act of running. Now imagine we are trying to create a dictionary (or vocabulary) of all words in a large text dataset. If we include all these variations as separate entries, the vocabulary becomes unnecessarily large and redundant. To solve this, NLP uses **Stemming** and **Lemmatization**.

Stemming is a crude heuristic process that chops off word endings to arrive at the root form. For instance, "running" becomes "runn" — not a valid word, but a stem that represents several variations. It's fast and simple, but can sometimes be inaccurate or overly aggressive.

Lemmatization, on the other hand, is more sophisticated. It uses a vocabulary and morphological analysis to return a real word — the *lemma*. For example, both "running" and "ran" are correctly reduced to "run". This makes lemmatization more reliable and useful for applications where accuracy is important.

These processes help reduce the complexity of text data, improve generalization, and boost model performance by treating different forms of the same word as one.

Stemming vs. Lemmatization

Both techniques reduce words to their base forms but differ in approach:

Feature	**Stemming**	**Lemmatization**
Method	Rule-based chopping	Dictionary-based normalization
Accuracy	Low (e.g., "running" → "run")	High (e.g., "better" → "good")
Speed	Faster	Slower (requires linguistic analysis)
Use Case	Search engines, IR systems	Chatbots, grammar analysis

Table 2: Comparison between stemming and lemmatization

Figure 7: Difference between stemming and lemmatization processes

Listing 6: Python Example of Stemming vs Lemmatization

```
from nltk.stem import PorterStemmer, WordNetLemmatizer

words = ["running", "ran", "runs"]
stemmer = PorterStemmer()
lemmatizer = WordNetLemmatizer()

print("Stemming:", [stemmer.stem(word) for word in words])
print("Lemmatization:", [lemmatizer.lemmatize(word, pos='v')
                         for word in words])
```

Output:

```
Stemming: ['run', 'ran', 'run']
Lemmatization: ['run', 'run', 'run']
```

Discussion:

- Stemming is faster but less accurate (e.g., "ran" remains "ran")
- Lemmatization requires POS tags for best results
- Choice depends on application needs (speed vs accuracy)

4.Key Takeaways

- **Tokenization** is the first step in NLP, breaking text into analyzable units
- **N-grams** capture context but increase computational complexity
- **POS Tagging** improves syntactic understanding of sentences
- **NER** extracts meaningful entities (people, organizations, dates)
- **Stop Word Removal** reduces noise in text data
- **Lemmatization** is more accurate than stemming for word normalization

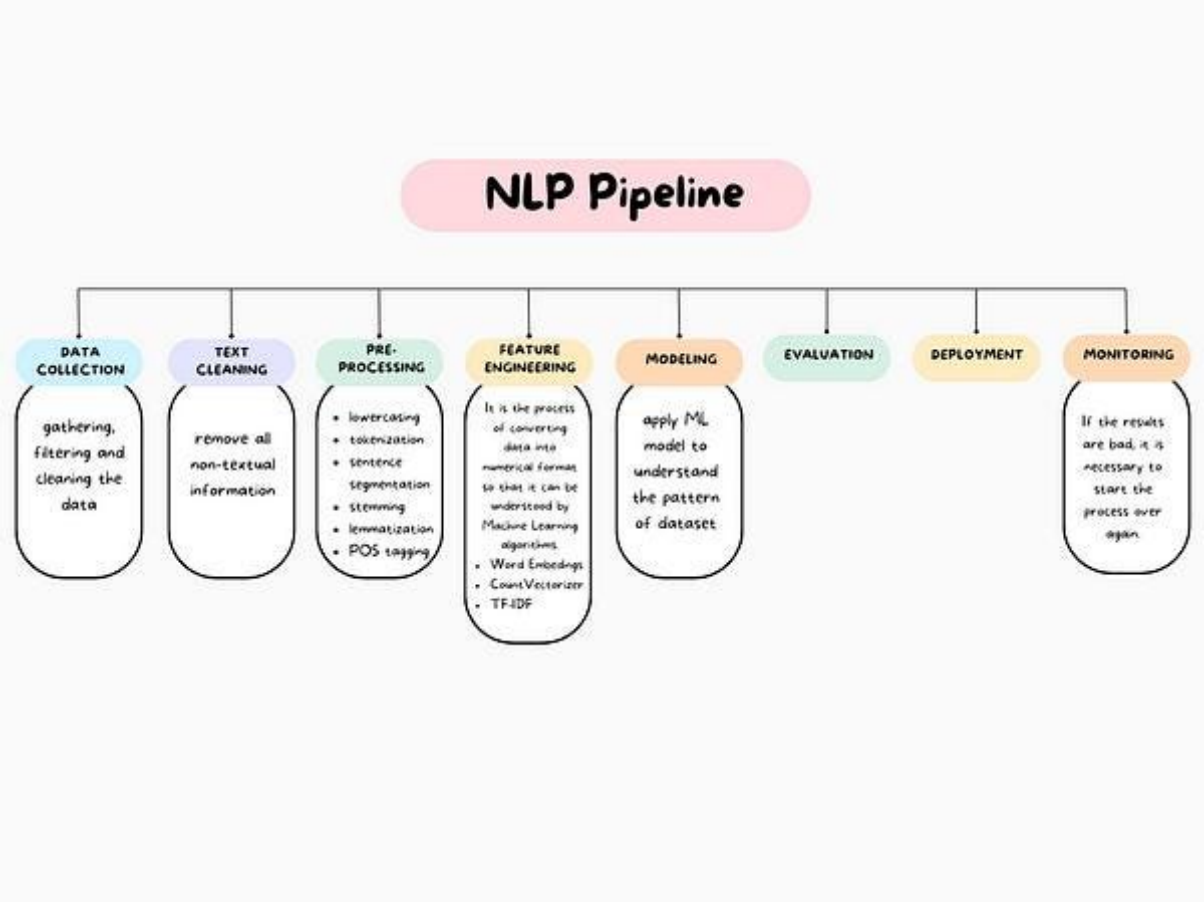


Figure 8: Complete NLP text preprocessing pipeline

These techniques form the foundation for advanced NLP applications like:

- Sentiment analysis
- Machine translation
- Chatbots
- Text summarization
- Information extraction