Basic NLP Concepts :

Why We Need Text Preprocessing:

1. **Improve Data Quality:** Cleans and removes noise from raw text data, making it more consistent.
2. **Reduce Dimensionality:** Simplifies the text, making it easier for models to process and analyze.
3. **Enhance Model Accuracy:** Helps models focus on relevant information, improving prediction accuracy.
4. **Standardize Input:** Ensures uniformity in text (e.g., lowercase, base forms), reducing variability.

When to Perform Text Preprocessing:

1. **Before Model Training:** Prepares text data to be fed into machine learning models.
2. **Before Feature Extraction:** Ensures that extracted features (like word vectors) are meaningful.
3. **In Real-Time Systems:** Preprocesses text on-the-fly in applications like chatbots.
4. **During Data Exploration:** Cleans text data before analysis to gain better insights.

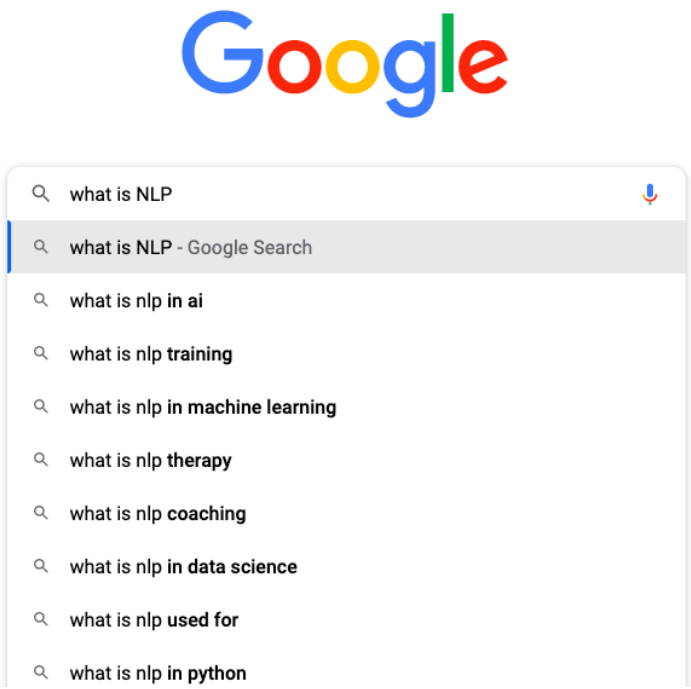
Natural Language Processing (NLP): A Comprehensive Overview

Common NLP Use Cases

1. **Language Translation**: Automatically translating text from one language to another.
2. **Speech Recognition**: Converting spoken language into text.
3. **Hiring and Recruitment**: Analyzing resumes and job descriptions.
4. **Chatbots**: Facilitating automated conversations with users.
5. **Sentiment Analysis**: Determining the sentiment (positive, negative, neutral) expressed in text.

Search Engine Results

Search engines use NLP to suggest relevant results based on user behavior and intent. For example, Google predicts what you'll type and shows relevant outcomes like a calculator for math equations or flight status for flight numbers.



Language Translation

NLP powers translation tools that convert text and voice formats between languages, with roots going back to early machine translation in the 1950s.

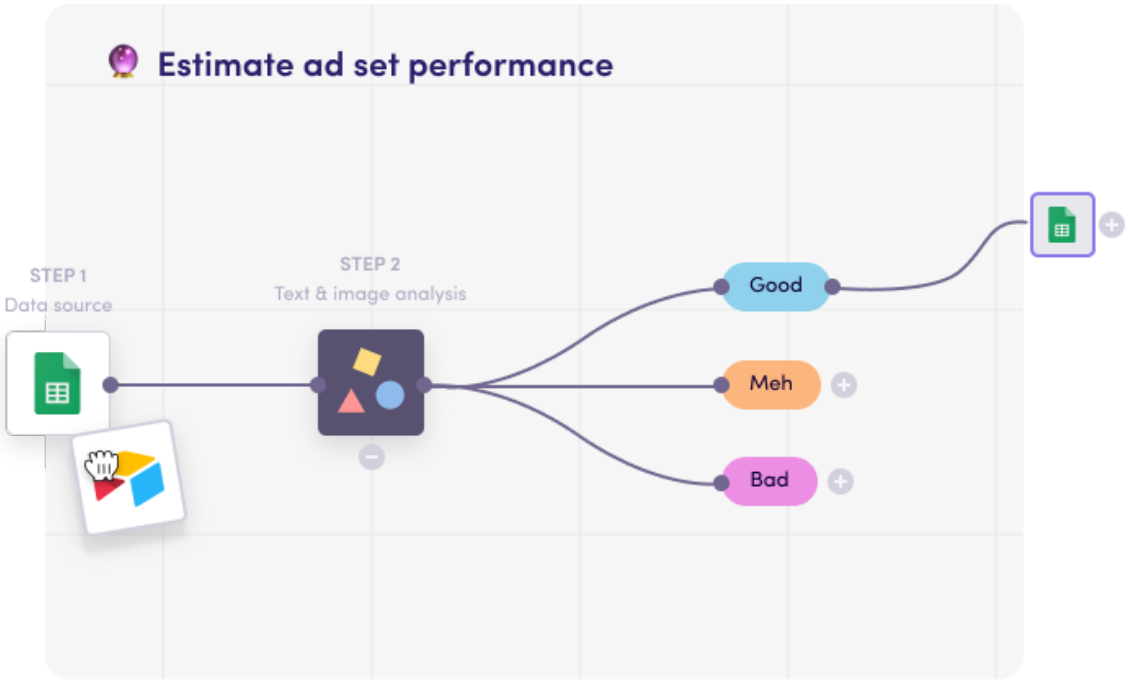
Semantic Search

NLP-driven semantic search improves customer experience by understanding the context of queries and suggesting relevant products.

Sentiment Analysis

Oftentimes, when businesses need help understanding their customer needs, they turn to sentiment analysis.

Sentiment analysis (also known as opinion mining) is an NLP strategy that can determine whether the meaning behind data is positive, negative, or neutral. For instance, if an unhappy client sends an email which mentions the terms “error” and “not worth the price”, then their opinion would be automatically tagged as one with negative sentiment.



Autocomplete & Autocorrect

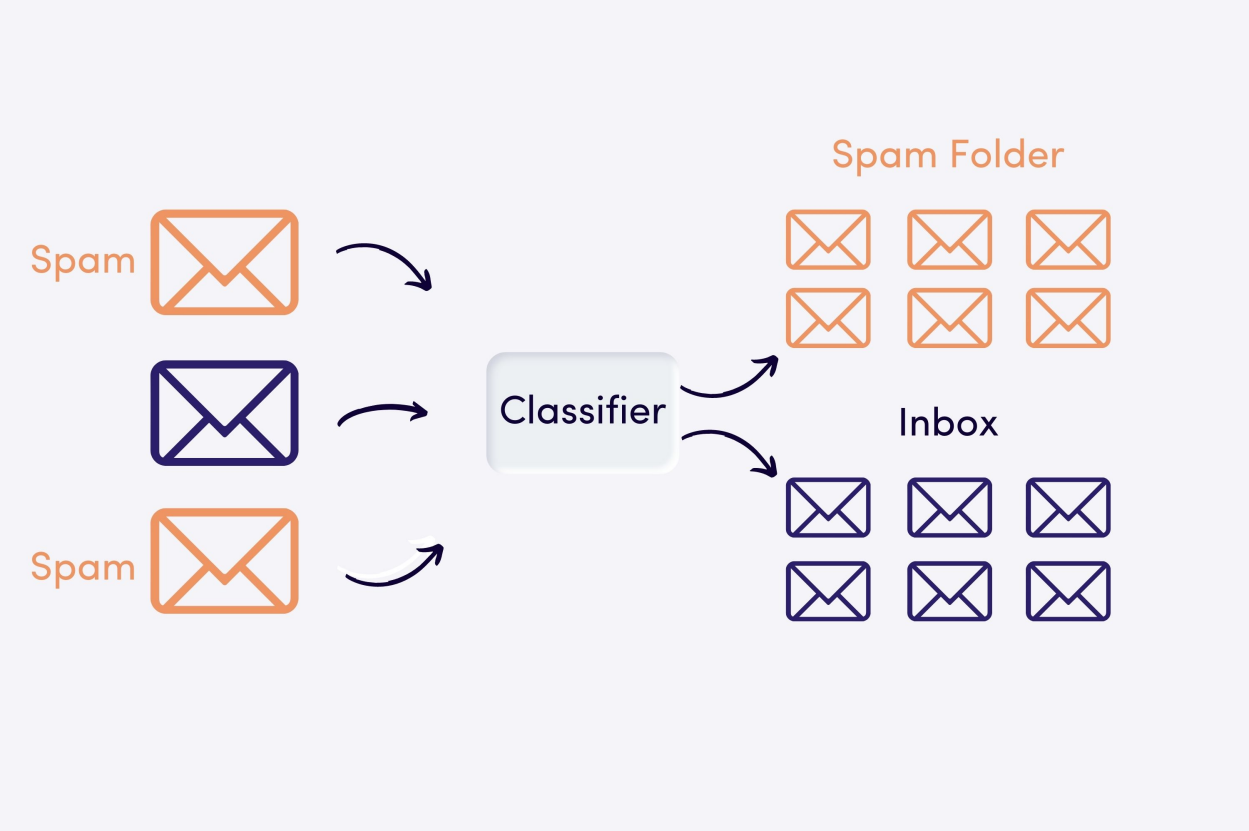
Autocomplete and autocorrect use NLP to predict and correct text as you type, improving typing accuracy and speed.

Spellcheck

Spellcheck notifies users of spelling errors and corrects them automatically, a common NLP application.

Email Filters

NLP enhances email filters, categorizing emails into primary, social, or promotions folders based on content.



Chatbots

NLP-powered chatbots provide quick, automated responses in customer service, helping businesses manage inquiries efficiently.

Smart Assistants

Smart assistants like Siri and Alexa use NLP for voice recognition and respond to everyday queries, even assisting with shopping.

Social Media Monitoring

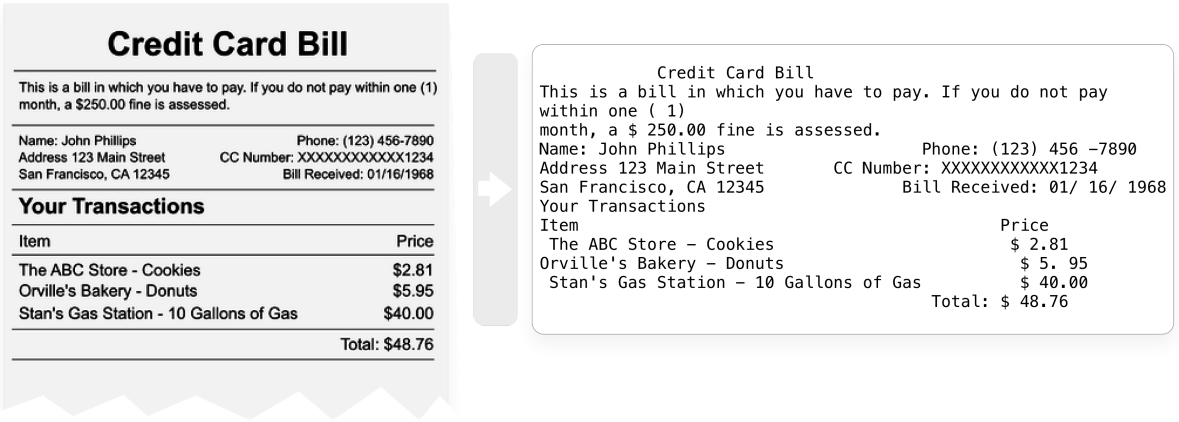
NLP helps monitor social media by filtering comments and analyzing sentiment to understand customer reactions.

Customer Service Automation

NLP automates customer service by responding to simple questions and routing support tickets to the right agents.

Optical Character Recognition (OCR)

OCR uses NLP to convert text from scanned documents or images into machine-readable formats, useful for tasks like translation.



Credit Card Bill

This is a bill in which you have to pay. If you do not pay within one (1) month, a $250.00 fine is assessed.

Name: John Phillips
Address 123 Main Street
San Francisco, CA 12345

Phone: (123) 456-7890
CC Number: XXXXXXXXXXX1234
Bill Received: 01/16/1968

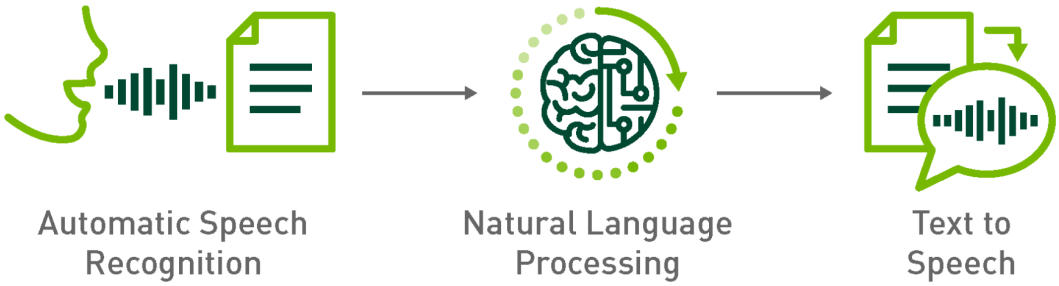
Your Transactions

Item	Price
The ABC Store - Cookies	$2.81
Orville's Bakery - Donuts	$5.95
Stan's Gas Station - 10 Gallons of Gas	$40.00
	Total: $48.76

```
          Credit Card Bill
This is a bill in which you have to pay. If you do not pay
within one ( 1)
month, a $ 250.00 fine is assessed.
Name: John Phillips                  Phone: (123) 456 -7890
Address 123 Main Street       CC Number: XXXXXXXXXXX1234
San Francisco, CA 12345             Bill Received: 01/ 16/ 1968
Your Transactions
Item                                              Price
 The ABC Store - Cookies                           $ 2.81
Orville's Bakery - Donuts                           $ 5. 95
 Stan's Gas Station - 10 Gallons of Gas             $ 40.00
                                        Total: $ 48.76
```

Speech Recognition

Speech recognition converts human speech into text, enabling voice search and other applications.



Natural Language Generation

NLP generates natural-sounding text or speech from data, used in smart assistants and customer service bots.

Syllabus :

- **Text Cleaning**
  - Removing Punctuation
  - Removing Special Characters
  - Removing Numbers
  - Removing Extra Whitespaces
  - Lowercasing Text
- **Tokenization**

    - Word Tokenization
    - Sentence Tokenization
    - Subword Tokenization
    - Character Tokenization
- **Stop Words Removal**
    - Identifying Stop Words
    - Removing Stop Words from Tokenized Text
- **Stemming and Lemmatization**
    - Stemming
    - Lemmatization
- **Text Normalization**
    - Converting Text to Lowercase
    - Removing Accents and Diacritics

- **Text Segmentation (Optional)**
    - Sentence Segmentation
    - Topic Segmentation
    - Chunking
- **N-Gram Generation (Optional)**
    - Unigrams
    - Bigrams
    - Trigrams
    - Higher-order N-Grams
- **Vectorization (Text to Numerical Form)**
    - OHE (one Hot encoding)
    - Bag of Words (BoW)
    - TF-IDF (Term Frequency-Inverse Document Frequency)

- **Word Embedding**
    - Word2Vec

- GloVe (Global Vectors for Word Representation)

Why We Need Text Preprocessing:

1. **Improve Data Quality:** Cleans and removes noise from raw text data, making it more consistent.
2. **Reduce Dimensionality:** Simplifies the text, making it easier for models to process and analyze.
3. **Enhance Model Accuracy:** Helps models focus on relevant information, improving prediction accuracy.
4. **Standardize Input:** Ensures uniformity in text (e.g., lowercase, base forms), reducing variability.

When to Perform Text Preprocessing:

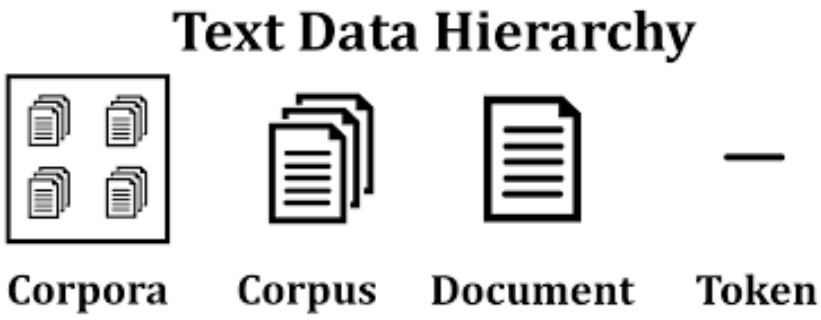
1. **Before Model Training:** Prepares text data to be fed into machine learning models.
2. **Before Feature Extraction:** Ensures that extracted features (like word vectors) are meaningful.
3. **In Real-Time Systems:** Preprocesses text on-the-fly in applications like chatbots.
4. **During Data Exploration:** Cleans text data before analysis to gain better insights.

Key NLP Terminology

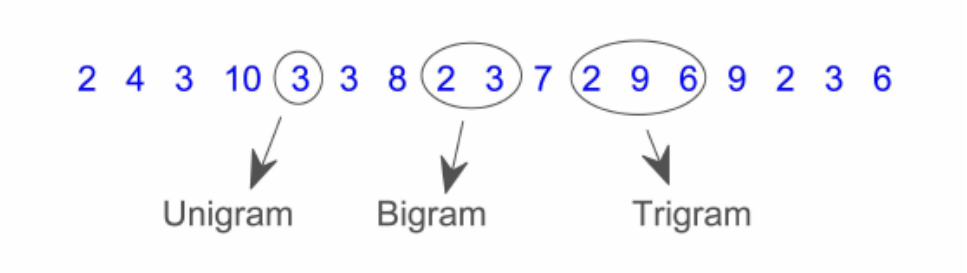
-
- **Corpus:**
  - **Explanation:** A large collection of text documents used for training NLP models. It's essentially the dataset containing all the text that will be analyzed.
- **Document:**

- **Explanation:** A single piece of text within a corpus. It can be an article, a paragraph, or even a sentence, depending on how the data is structured.

- **Vocabulary:**
  - **Explanation:** The set of all unique words present in the corpus. It's the dictionary of terms that the model will learn from.
- **Token:**
  - **Explanation:** The smallest unit of text, usually a word or punctuation mark, that results from the tokenization process. Tokens are the building blocks for text analysis.
- **Frequency:**
  - **Explanation:** The number of times a particular token (word) appears in a document or corpus. It's used to understand how common or rare certain words are in the text.



- **N-Grams:**
  - **Explanation:** A sequence of 'n' consecutive words or tokens in a text. For example, in the sentence "I love NLP," bigrams (2-grams) would be "I love" and "love NLP."
- **Unigram:**
  - **Explanation:** A single word or token in a text. For example, in "I love NLP," the unigrams are "I," "love," and "NLP."
- **Bigram:**
  - **Explanation:** A pair of consecutive words or tokens. For example, in "I love NLP," the bigrams are "I love" and "love NLP."
- **Trigram:**
  - **Explanation:** A sequence of three consecutive words or tokens. For example, in "I love learning NLP," the trigram is "I love learning."



1. Punctuation Removal

Why It's Important:

- Punctuation marks like commas, periods, and exclamation points often don't add much value in text analysis, especially when we're only interested in the content words. Removing them helps in standardizing the text and reducing noise.

Python Code:

The issue in your code arises because `string.punctuation` contains individual punctuation characters (e.g., `.`, `,`, `!`), and you need to check if each character in a word is a punctuation mark. If your goal is to remove punctuation from each word in `filtered_tokens`, you need to iterate over each word and remove punctuation from it. Here's how you can do it:

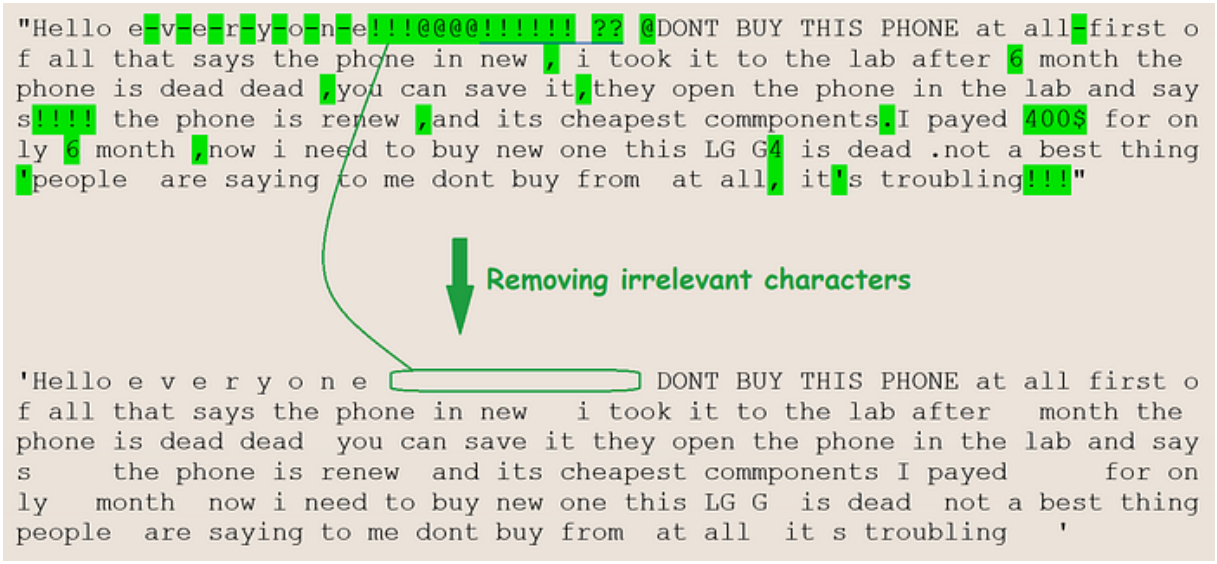
```
import string

# Example filtered tokens
filtered_tokens = ['Hello!', 'This', 'is', 'a', 'test.', 'Let\'s', 'rem
ove', 'punctuation!']

# Remove punctuation from each word
punctuation_removed = [''.join(char for char in word if char not in str
ing.punctuation) for word in filtered_tokens]
```

```
# Remove empty strings if any word consisted only of punctuation
punctuation_removed = [word for word in punctuation_removed if word]

print("Without Punctuation:", punctuation_removed)
```

Removing all irrelevant characters (Numbers and Punctuation)

Remove numbers if they are not relevant to your analyses (0–9). And punctuation also will be remove. Punctuation is basically the set of symbols [!"#$%&'()*+,-./:;⇔?@[\]^_`{|}~]:

> Result: All numeric and punctuation has been replaced with space ' ' .




```
import re

def clean_text(text):
    # Remove all characters except alphabets and spaces
    cleaned_text = re.sub(r'[^A-Za-z\s]', '', text)
    return cleaned_text
```

```
paragraph = "Here's a paragraph with numbers 123 and special character
s! Like @#$%."
cleaned_paragraph = clean_text(paragraph)
print(cleaned_paragraph)
```

3. Whitespace Removal

Why It's Important:

- Extra spaces and tabs can create inconsistencies in text processing. Removing them ensures that the text is clean and uniform, preventing potential errors in analysis.

Python Code:

```
pythonCopy code

text_with_whitespace = "  This   is   an example     sentence with    e
xtra spaces.  "

# Remove extra whitespace
cleaned_text = " ".join(text_with_whitespace.split())

print("Cleaned Text:", cleaned_text)
```

Output:

```
vbnetCopy code
Cleaned Text: This is an example sentence with extra spaces.
```

Summary

- **Stop Words Removal:** Reduces data size and enhances focus on meaningful words.
- **Punctuation Removal:** Eliminates noise, making the text more consistent.
- **Whitespace Removal:** Ensures clean and uniform text, preventing processing errors.

Tokenization —

Tokenization is the process of splitting the given text into smaller pieces called tokens. Words, numbers, punctuation marks, and others can be considered as tokens. We will use Natural language tool kit (nltk) library for tokenization.

1. Tokenization

Definition

: The process of breaking down text into smaller units called tokens (words, phrases, or symbols).

When to Use

: At the beginning of the preprocessing pipeline to facilitate further analysis.

Advantages

- Simplifies text analysis by creating manageable units.
- Essential for subsequent steps like POS tagging and NER.

Disadvantages:

- Can lead to loss of context if not done carefully (splitting contractions).
- May not handle multilingual text effectively.

> Result: As we can see the string has been changed into tokens, that has been stored in the form of ‘list of string’ .

```
'hello e v e r y o n e                dont buy this phone at all first o
f all that says the phone in new   i took it to the lab after   month the
phone is dead dead  you can save it they open the phone in the lab and say
s     the phone is renew  and its cheapest commponents i payed      for on
ly   month  now i need to buy new one this lg g  is dead  not a best thing
people  are saying to me dont buy from  at all  it s troubling   '
```


```
['hello', 'e', 'v', 'e', 'r', 'y', 'o', 'n', 'e', 'dont', 'buy', 'this',
'phone', 'at', 'all', 'first', 'of', 'all', 'that', 'says', 'the',
'phone', 'in', 'new', 'i', 'took', 'it', 'to', 'the', 'lab', 'after',
'month', 'the', 'phone', 'is', 'dead', 'dead, 'you', 'can', 'save', 'it',
'they', 'open', 'the', 'phone', 'in', 'the', 'lab', 'and', 'says', 'the',
'phone', 'is', 'renew', 'and', 'its', 'cheapest', 'commponents', 'i',
'payed', 'for', 'only', 'month', 'now', 'i', 'need', 'to', 'buy', 'new',
'one', 'this', 'lg', 'g', 'is', 'dead', 'not', 'a', 'best', 'thing',
'people', 'are', 'saying', 'to', 'me', 'dont', 'buy', 'from', 'at', 'all',
'it', 's', 'troubling']
```

```
import nltk
from nltk.tokenize import word_tokenize

# Download the necessary NLTK resources (if you haven't already)
nltk.download('punkt')

# Example sentence
sentence = "Natural Language Processing with NLTK is fun and exciting!"

# Tokenize the sentence
tokens = word_tokenize(sentence)

# Print the tokens
```

```
print(tokens)
```

1. Word Tokenization

- **Definition:** Splits text into individual words.
- **Example:** "Hello, world!" → ["Hello", "world"]
- **Code Example (Python):**

```
from nltk.tokenize import word_tokenize

text = "Hello, world!"
tokens = word_tokenize(text)
print(tokens)  # Output: ['Hello', ',', 'world', '!']
```

2. Sentence Tokenization

- **Definition:** Splits text into sentences.
- **Example:** "Hello, world! How are you?" → ["Hello, world!", "How are you?"]
- **Code Example (Python):**

```
from nltk.tokenize import sent_tokenize

text = "Hello, world! How are you?"
sentences = sent_tokenize(text)
print(sentences)  # Output: ['Hello, world!', 'How are you?']
```

3. Character Tokenization

- **Definition:** Splits text into individual characters.
- **Example:** "Hello" → ["H", "e", "l", "l", "o"]

- **Code Example (Python):**

```
pythonCopy code
text = "Hello"
tokens = list(text)
print(tokens)  # Output: ['H', 'e', 'l', 'l', 'o']
```

4. Subword Tokenization

- **Definition:** Splits text into subword units, useful for handling rare or unknown words.
- **Example:** "unbelievable" → ["un", "believable"]
- **Code Example (Python with `SentencePiece`):**

```
import sentencepiece as spm

sp = spm.SentencePieceProcessor(model_file='m.model')
text = "unbelievable"
tokens = sp.encode_as_pieces(text)
print(tokens)  # Output example: ['▁un', 'believable']
```

5. Whitespace Tokenization

- **Definition:** Splits text based on whitespace.
- **Example:** "Hello world" → ["Hello", "world"]
- **Code Example (Python):**

```
text = "Hello world"
tokens = text.split()
print(tokens)  # Output: ['Hello', 'world']
```

Removing Stopwords

“Stopwords” are the most common words in a language like “the”, “a”, “me”, “is”, “to”, “all”,. These words do not carry important meaning and are usually removed from texts. It is possible to remove stopwords using Natural Language Toolkit (nltk). You also may check the list of stopwords by using following code.

Stop Words Removal

Definition

: The process of removing common words that do not add significant meaning to the text (e.g., "the," "is," "and").

When to Use

: After tokenization to reduce noise in the data.

Advantages

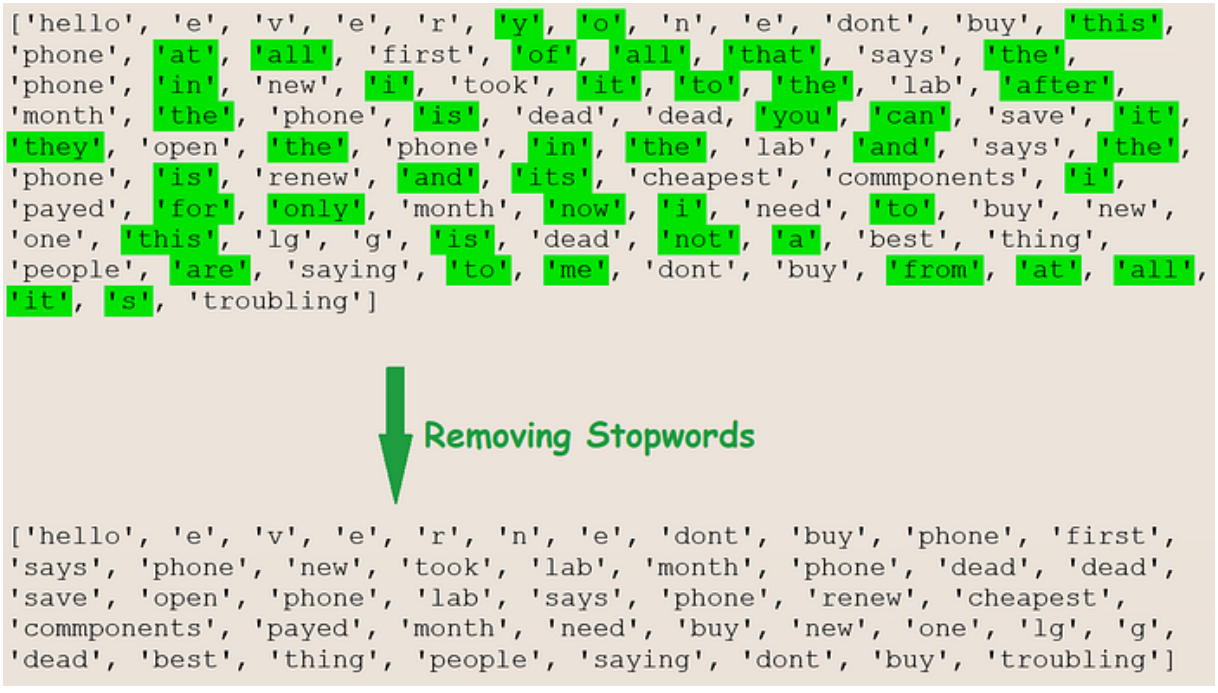
:

- Reduces dimensionality of the dataset.
- Enhances the focus on meaningful words.

Disadvantages

:

- May remove important context in certain applications (e.g., sentiment analysis).
- Requires customization based on the specific use case and language.



1. Stop Words Removal

Why It's Important:

- Stop words like "is," "the," and "and" are common words that usually don't contribute much to the meaning of the text. Removing them reduces the size of the data and helps the model focus on more significant words.

Python Code:

```
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize

# Sample text
text = "This is an example sentence, demonstrating the removal of stop
words."

# Tokenize the text
tokens = word_tokenize(text)

# Remove stop words
stop_words = set(stopwords.words('english'))
filtered_tokens = [word for word in tokens if word.lower() not in stop_
words]

print("Filtered Tokens:", filtered_tokens)
```

Stemming and Lemmatization :

Stemming usually refers to a crude process that chops off the ends of words in the hope of achieving this goal correctly most of the time, and often includes the removal of derivational units (the obtained element is known as the stem).

lemmatization consists in doing things properly with the use of a vocabulary and morphological analysis of words, to return the base or dictionary form of a word, which is known as the lemma.

$$\text{I } \mathbf{saw} \text{ an amazing thing} \xrightarrow{\text{stem}} \text{I } \mathbf{s} \text{ an amazing thing}$$

$$\text{I } \mathbf{saw} \text{ an amazing thing} \xrightarrow{\text{lemma}} \text{I } \mathbf{see} \text{ an amazing thing}$$

Stemming

Definition

: Reduces words to their root form by removing suffixes ("running" to "run").

When to Use

: When a rough approximation of word forms is sufficient.

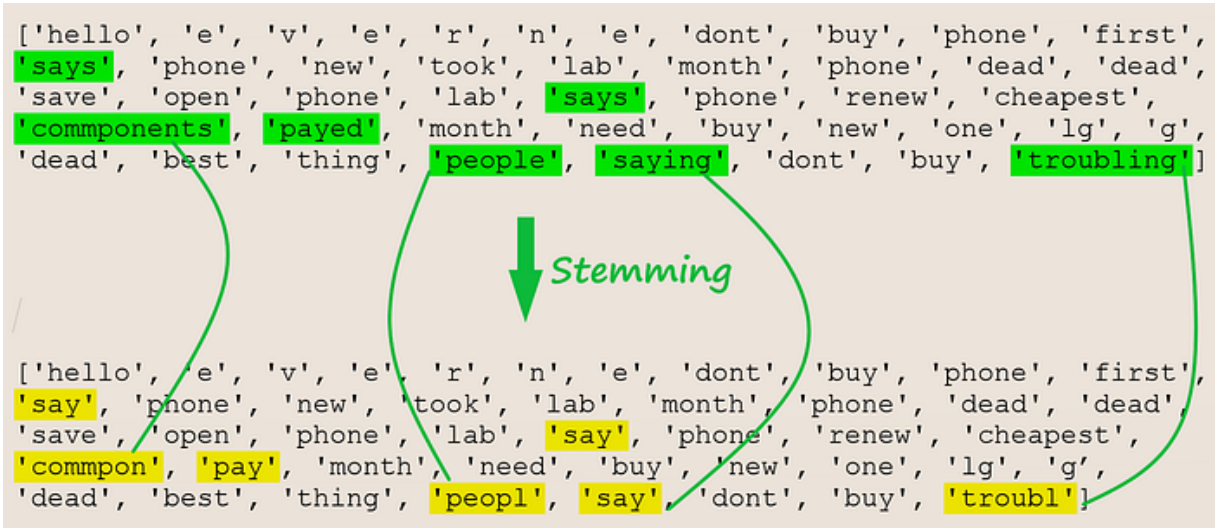
Advantages

- Simple and computationally efficient.
- Helps in standardizing words to a common base form.

Disadvantages

:

- Can produce non-words ("universe" to "univers").
- May lose some meaning or context.



Stemming

```
import nltk
from nltk.stem import PorterStemmer

# Initialize the Porter Stemmer
stemmer = PorterStemmer()

# Example words
words = ["running", "runner", "ran", "easily", "fairly"]

# Apply stemming
stemmed_words = [stemmer.stem(word) for word in words]

print("Stemmed Words:", stemmed_words)
```

Lemmatization

Definition:

Reduces words to their base or dictionary form, considering the context (e.g., "better" to "good").

When to Use

: When accurate word forms are required for analysis.

Advantages

:

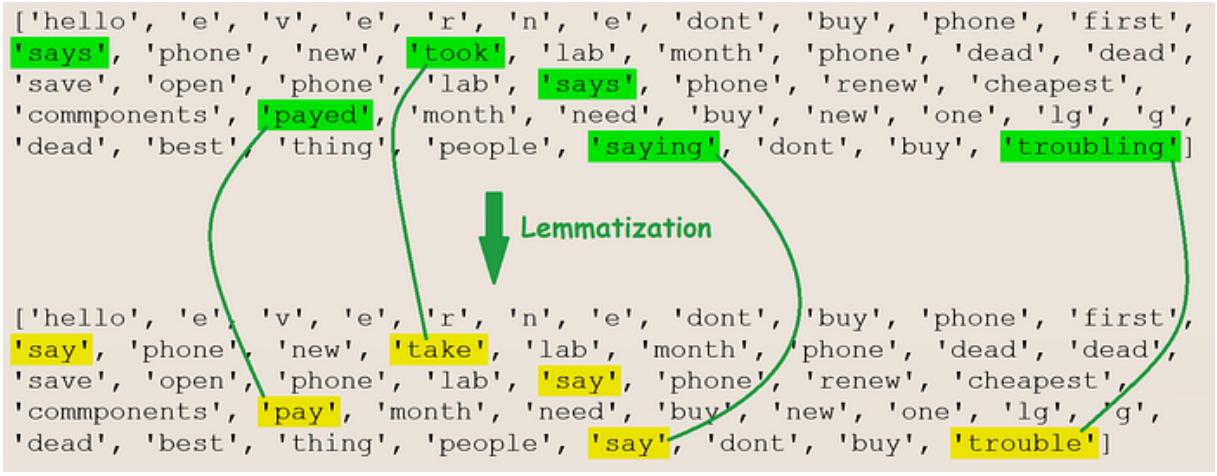
- Provides more meaningful and contextually appropriate results than stemming.
- Preserves grammatical correctness.

Disadvantages

:

- More computationally intensive than stemming.
- Requires a comprehensive dictionary and context understanding.
- ***Lemmatization:***

> Result: Now here we can see it finds the root word, like ‘troubling’ to ‘trouble’, ‘took’ to ‘take’ and ‘payed’ to ‘pay’ . So, As opposed to stemming, lemmatization does not simply chop off inflections. Instead it uses lexical knowledge bases to get the correct base forms of words.



Lemmatization

```
import nltk
from nltk.stem import WordNetLemmatizer

# Download the necessary NLTK resources (if you haven't already)
nltk.download('wordnet')
nltk.download('omw-1.4')

# Initialize the WordNet Lemmatizer
lemmatizer = WordNetLemmatizer()

# Example words
words = ["running", "runner", "ran", "easily", "fairly"]
```

```
# Apply lemmatization
lemmatized_words = [lemmatizer.lemmatize(word) for word in words]

print("Lemmatized Words:", lemmatized_words)
```

Key Differences:

- **Stemming** may produce non-meaningful roots ("easily" to "easili"), whereas **lemmatization** produces actual words.
- **Lemmatization** considers the context and part of speech (POS) of a word, while **stemming** does not.
- **Stemming** is faster but less accurate; **lemmatization** is slower but more accurate and meaningful.

Text Normalization

Definition

: The process of converting text to a standard format (lowercasing, removing punctuation).

When to Use

: Early in the preprocessing pipeline to ensure consistency.

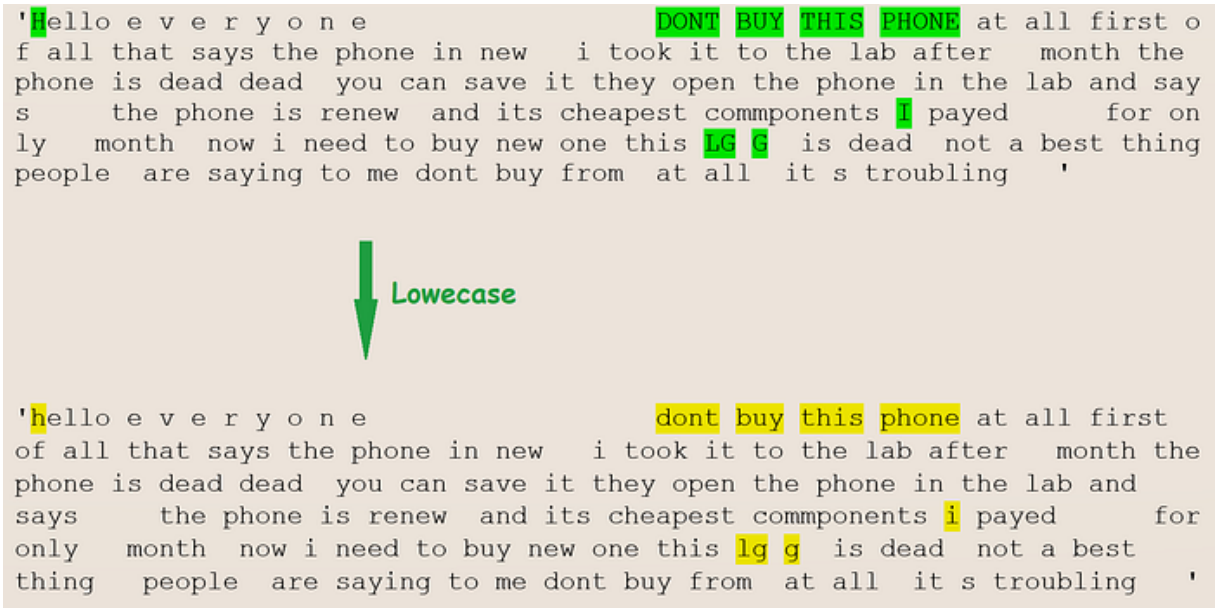
Advantages

- Reduces variability in text data.
- Helps in improving model performance by standardizing input.

Disadvantages

- May remove important features (capitalization for proper nouns).
- Requires careful consideration of what to normalize.

Convert all characters into lowercase —



Convert all characters into lowercase

Here's a paragraph you can use for converting characters into lowercase:

```
paragraph = "This is an Example Paragraph with Mixed CASE letters."
lowercase_paragraph = paragraph.lower()
print(lowercase_paragraph)
```

1. Lowercasing

Lowercasing involves converting all characters in the text to lowercase. This helps in standardizing the text and reducing the complexity of further text processing steps.

Example:

```
text = "Natural Language Processing is an Interesting Field."

# Convert to lowercase
lowercased_text = text.lower()
```

```
print("Lowercased Text:", lowercased_text)
```

Output:

```
sqlCopy code
Lowercased Text: natural language processing is an interesting field.
```

4. Text Segmentation

Text segmentation involves breaking down a text into meaningful chunks, such as sentences or phrases.

Example:

```
pythonCopy code
import nltk
nltk.download('punkt')
from nltk.tokenize import sent_tokenize, word_tokenize

# Sample text
text = "Natural language processing is an interesting field. It has man
y applications in AI."

# Sentence segmentation
sentences = sent_tokenize(text)

# Word segmentation
words = word_tokenize(text)

print("Sentences:", sentences)
print("Words:", words)
```

Summary

- **Lowercasing** standardizes text by converting it to lowercase.
- **Stemming** reduces words to their base form, often resulting in non-words.

- **Lemmatization** reduces words to their dictionary form, producing actual words.
- **Text Segmentation** breaks down text into meaningful chunks, such as sentences and words.

Part-of-Speech Tagging (POS Tagging)

Definition

: Categorizes each word in a sentence into its grammatical function (nouns, verbs, adjectives, etc.).

When to Use

: After tokenization to understand the grammatical structure of the text.

Advantages

- Enhances understanding of the text's meaning and structure.
- Useful for tasks like parsing and NER.

Disadvantage

- Requires accurate models, which can be language-dependent.
- May struggle with ambiguous words based on context.

. Part of Speech (POS) Tagging

POS tagging is the process of marking up a word in a text as corresponding to a particular part of speech, based on both its definition and context.

```
import nltk
from nltk.tokenize import word_tokenize

# Download necessary NLTK resources (if you haven't already)
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')

# Example sentence
sentence = "The quick brown fox jumps over the lazy dog."
```

```
# Tokenize the sentence
tokens = word_tokenize(sentence)

# Perform POS tagging
pos_tags = nltk.pos_tag(tokens)

print("POS Tags:", pos_tags)
```

Output:

In the output:

- `DT` = Determiner
- `JJ` = Adjective
- `NN` = Noun
- `VBZ` = Verb (3rd person singular present)
- `IN` = Preposition

Named Entity Recognition (NER)

Definition

: Identifies and classifies named entities (names, locations, organizations) in text.

When to Use

: After tokenization and POS tagging to extract structured information.

Advantages

:

- Helps in extracting valuable information from unstructured text.
- Facilitates tasks like information retrieval and question answering.

Disadvantages

:

- Can be sensitive to the quality of training data.
- May misclassify entities in ambiguous contexts.

2. Named Entity Recognition (NER)

NER is the process of identifying entities such as names of persons, organizations, locations, expressions of times, quantities, monetary values, percentages, etc., in text.

```
pythonCopy code
import nltk
from nltk import word_tokenize, pos_tag, ne_chunk

# Download necessary NLTK resources (if you haven't already)
nltk.download('maxent_ne_chunker')
nltk.download('words')

# Example sentence
sentence = "Apple is looking at buying U.K. startup for $1 billion."

# Tokenize the sentence
tokens = word_tokenize(sentence)

# Perform POS tagging
pos_tags = pos_tag(tokens)

# Perform Named Entity Recognition
named_entities = ne_chunk(pos_tags)

print("Named Entities:", named_entities)
```

Detailed Explanation

- **POS Tagging**: Each word in the sentence is tagged with its corresponding part of speech.
- **NER**: The `ne_chunk` function in NLTK takes POS-tagged tokens and identifies named entities, which could be organizations, people, locations, etc.

Detailed Explanation:

1. One-Hot Encoding

Color	Color_Yellow	Color_Blue	Color_Green
Yellow	1	0	0
Blue	0	1	0
Green	0	0	1

Description:

One-Hot Encoding represents each word in the text as a binary vector. In this vector, each dimensabulary, with a `1` in the position of the word and `0`s elsewhere.

Why Use It:

- **Simple Representation:** It provides a straightforward method for converting text into numerical form.
- **Preprocessing Step:** Useful for creating input features for machine learning models that require numerical input.

When to Use:

- **Small Vocabulary:** Suitable when dealing with a limited number of unique words.
- **Initial Model Building:** Often used as a starting point in text classification tasks.

Advantages:

- **Simplicity:** Easy to understand and implement.
- **Clear Mapping:** Directly maps each word to a unique position in the vector space.

Disadvantages:

- **High Dimensionality:** Vocabulary size directly affects vector size, leading to high-dimensional vectors if the vocabulary is large.
- **Sparsity:** Most of the vector elements are zero, resulting in sparse representations.
- **No Semantic Meaning:** Fails to capture any meaning or relationships between words.

- out of vocabulary
- new data we cannot add or extra data we cannot handle

Example:

Vocabulary:

```
vocabulary = ["I", "love", "machine", "learning"]
```

One-Hot Encoding for "machine":

```
import numpy as np

def one_hot_encode(word, vocab):
    vector = np.zeros(len(vocab))
    vector[vocab.index(word)] = 1
    return vector

vocab = ["I", "love", "machine", "learning"]
one_hot = one_hot_encode("machine", vocab)
print(one_hot)  # Output: [0. 0. 1. 0.]
```

Example: One-Hot Encoding with Pandas

Dataset:
Let's start with a simple dataset containing a categorical column:

```
pythonCopy code
import pandas as pd

# Sample dataset with a categorical column
data = {'Color': ['Red', 'Blue', 'Green', 'Red', 'Green']}
df = pd.DataFrame(data)

print("Original DataFrame:")
print(df)
```

Performing One-Hot Encoding:

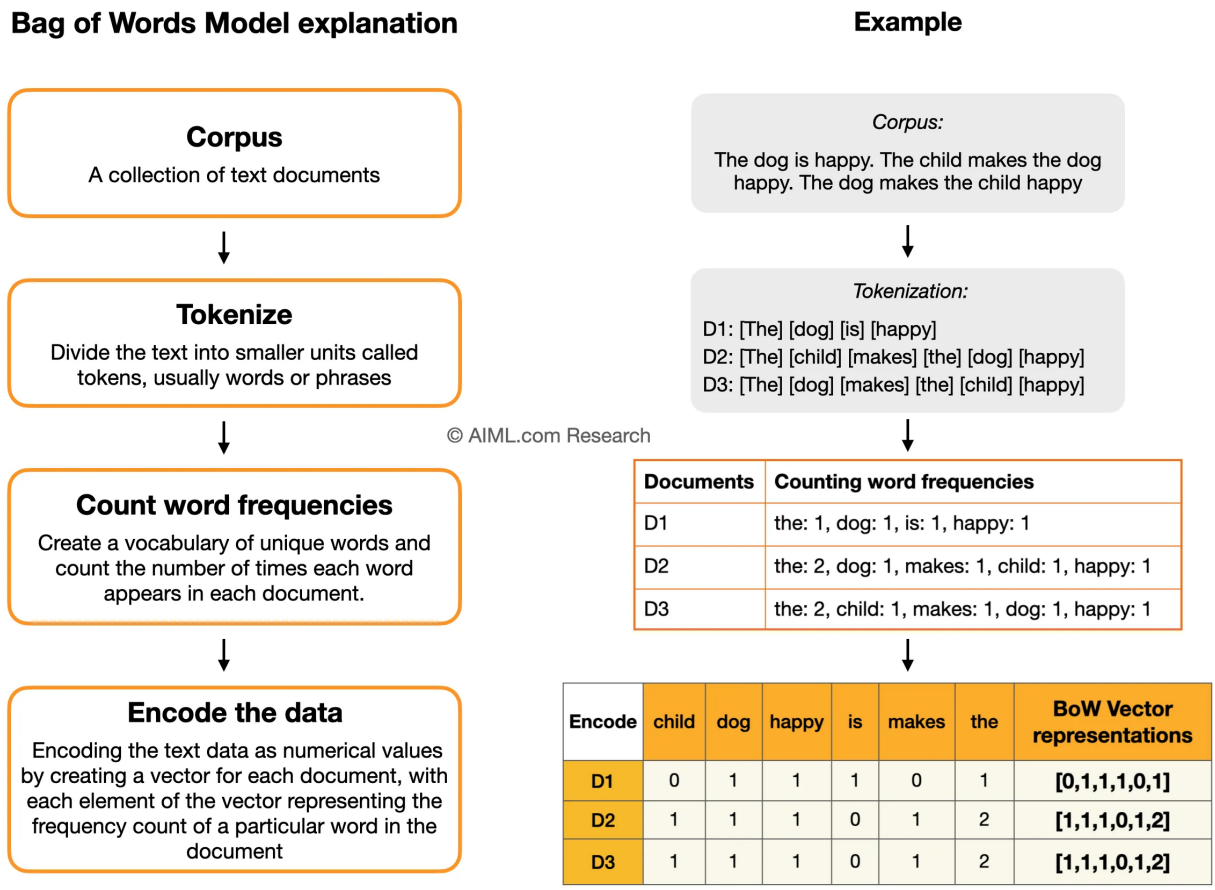
Using Pandas, you can perform one-hot encoding with the `pd.get_dummies()` function:

```
pythonCopy code
# Perform one-hot encoding using Pandas
one_hot_encoded = pd.get_dummies(df, columns=['Color'])

print("\nOne-Hot Encoded DataFrame:")
print(one_hot_encoded)
```

Code Example:

```
pythonCopy code
from sklearn.preprocessing import OneHotEncoder
import numpy as np

# Define the vocabulary
vocabulary = ["apple", "banana", "orange"]

# Convert the vocabulary into a format suitable for OneHotEncoder
vocab_array = np.array(vocabulary).reshape(-1, 1)

# Initialize the OneHotEncoder
one_hot_encoder = OneHotEncoder(sparse=False)

# Fit and transform the vocabulary
one_hot_encoded = one_hot_encoder.fit_transform(vocab_array)

# Display the One-Hot Encoded vectors
for word, vec in zip(vocabulary, one_hot_encoded):
    print(f"{word}: {vec}")
```

Bag-of-Words (BOW) Model: Detailed Explanation



What is Bag-of-Words?

The Bag-of-Words (BOW) model is a method for representing text data in Natural Language Processing (NLP). It transforms text into a numerical format by counting the frequency of each word in a document, disregarding grammar and word order.

Key Characteristics:

- **Unordered:** The model does not consider the sequence or structure of words in the text.
- **Frequency-Based:** It captures how often each word appears in the document.

Why Use Bag-of-Words?

- **Simplicity:** BOW is straightforward to implement and understand, making it a good starting point for text representation.
- **Effectiveness:** Despite its simplicity, BOW can perform well for various NLP tasks, such as text classification and sentiment analysis.
- **Compatibility:** The BOW representation can be easily used with many machine learning algorithms.

When to Use Bag-of-Words?

- **Initial Text Representation:** BOW is often used as an initial method for text representation before applying more complex models.
- **Text Classification:** It is suitable for tasks like spam detection, sentiment analysis, and topic classification.
- **Small to Medium-Sized Datasets:** BOW works well when the dataset is not too large, as it can become computationally expensive with larger vocabularies.

Importance of Bag-of-Words

- **Feature Extraction:** BOW transforms text into a numerical format that can be fed into machine learning models.
- **Baseline Comparison:** It serves as a baseline for evaluating the performance of more sophisticated models like TF-IDF or word embeddings.

How to Implement Bag-of-Words

1. **Text Preprocessing:**
    - **Tokenization:** Split text into words.
    - **Normalization:** Convert text to lowercase and remove stop words.
2. **Vocabulary Collection:**
    - Create a set of unique words from the text.
3. **Vectorization:**
    - Convert text into numerical vectors based on word frequency.

Disadvantages:

1. sparsity
2. ordering
3. out of vocabulary issue

Example of Bag-of-Words

Documents:

1. Document 1: "I love machine learning"
2. Document 2: "Machine learning is fun"

Step 1: Text Preprocessing

- **Tokenization:**
  - Doc 1: ["I", "love", "machine", "learning"]
  - Doc 2: ["Machine", "learning", "is", "fun"]
- **Normalization:**
  - Doc 1: ["love", "machine", "learning"]
  - Doc 2: ["machine", "learning", "fun"]

Step 2: Vocabulary Collection

The vocabulary from the documents is:

```
pythonCopy code
["love", "machine", "learning", "is", "fun"]
```

Step 3: Vectorization

Create a document-term matrix based on word frequency:

Document	love	machine	learning	is	fun
Document 1	1	1	1	0	0
Document 2	0	1	1	1	1

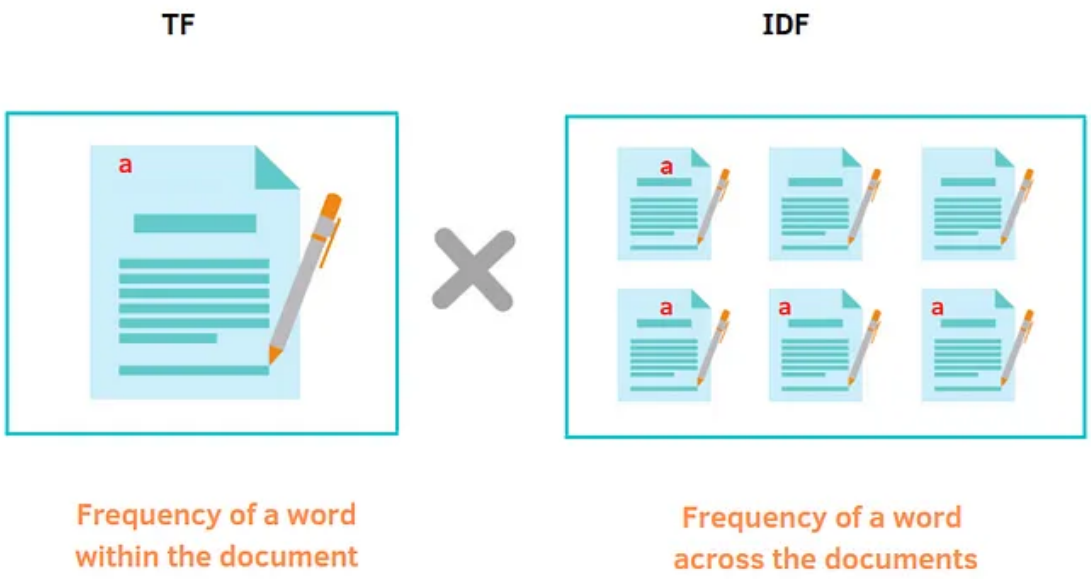
Code Implementation

Here's how to implement the Bag-of-Words model using Python with the scikit-learn library:

```
from sklearn.feature_extraction.text import CountVectorizer
import pandas as pd

# Sample documents
corpus = [
    "I love machine learning",
    "Machine learning is fun"
]

# Create the Bag-of-Words vectorizer
vectorizer = CountVectorizer()

# Fit and transform the corpus
X = vectorizer.fit_transform(corpus)
```

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

# Convert the Bag-of-Words matrix to a dense format and display it
dense = X.todense()
denselist = dense.tolist()

# Display the Bag-of-Words representation
df_bow = pd.DataFrame(denselist, columns=feature_names)
print(df_bow)
```

TF -IDF :



Standard TF-IDF

It’s obtained by combining two terms:

$$\text{TF-IDF}(t,d) = \text{TF}(t,d) \times \text{IDF}(t)$$

Image credit: Author

where **Term Frequency** (TF) is the frequency of the word `t` within the document d. In other words, it’s the ratio between the count of the word within the document and the total number of words:

$$\text{TF}(t,d) = \frac{\text{number of times t appears in d}}{\text{total number of words in d}}$$

As we said before, the term frequency is not enough to provide efficient measures. We also need to combine it with another term, called **Inverse Document Frequency**. It’s a logarithmic transformation of a fraction, calculated as the total number of documents in the corpus divided by the number of documents containing the word.

$$\text{IDF}(t) = \log \frac{\text{Total number of documents}}{\text{Number of documents that contain t}}$$

What is TF-IDF?

TF-IDF stands for Term Frequency-Inverse Document Frequency. It is a numerical statistic used in information retrieval and text mining to evaluate the importance of a term (word or phrase) in a document relative to a collection of documents (corpus). TF-IDF helps identify terms that are significant in specific documents while discounting terms that appear frequently across many documents.

Components of TF-IDF

1. **Term Frequency (TF)**:
    - **Definition**: Measures how frequently a term appears in a document.
    - **Calculation**:

    - **Raw Count**: Number of times the term appears in the document.
    - **Boolean Frequency**: 1 if the term appears, 0 otherwise.

2. **Inverse Document Frequency (IDF)**:
    - **Definition**: Measures how common or rare a term is across all documents in the corpus.
    -
3. **TF-IDF Score**:
    - **Definition**: Combines TF and IDF to provide a measure of a term's importance in a document relative to the entire corpus.

When to Use TF-IDF

- **Document Classification**: To determine the importance of terms in classifying text documents.
- **Information Retrieval**: To rank documents based on their relevance to a search query.
- **Text Mining**: To extract meaningful terms for further analysis or modeling.

importance of TF-IDF

- **Highlighting Unique Words**: TF-IDF helps identify words that are significant in a specific document but not common across all documents. This is particularly useful for information retrieval and text mining.
- **Feature Representation**: It provides a way to convert text into a numerical format that can be used in machine learning algorithms.

Example of TF-IDF Calculation

Corpus

1. **Document 1**: "I love machine learning"
2. **Document 2**: "Machine learning is fun"
3. **Document 3**: "I love programming"

Calculate TF-IDF for the Term "machine" in Document 1

Step 1: Calculate Term Frequency (TF)

- **Term Frequency (TF)**: TF for "machine" in Document 1 = (Number of times "machine" appears in Document 1) / (Total number of words in Document 1)
- Document 1: "I love machine learning" (Total words = 4)
- "Machine" appears 1 time in Document 1.
- **TF("machine", Doc1)** = 1 / 4 = 0.25

Step 2: Calculate Inverse Document Frequency (IDF)

- **IDF**: IDF for "machine" = log(Total number of documents / Number of documents containing the term)
- "Machine" appears in 2 documents out of 3.
- **IDF("machine")** = log(3 / 2) ≈ 0.176

Step 3: Calculate TF-IDF

- **TF-IDF**: TF-IDF("machine", Doc1) = TF * IDF = 0.25 ∗ 0.176 ≈ 0.044

Corrected Calculation

Term Frequency (TF):

- **Document 1**: "I love machine learning" (Total words = 4)
- **TF("machine", Doc1)** = 1 / 4 = 0.25

Inverse Document Frequency (IDF):

- **IDF("machine")** = log(3 / 2) ≈ 0.176

TF-IDF Calculation:

- **TF-IDF("machine", Doc1)** = 0.25 ∗ 0.176 ≈ 0.044

Updated Python Code

Here's the corrected code implementation to reflect these calculations:

```
from sklearn.feature_extraction.text import TfidfVectorizer
import pandas as pd

# Sample corpus
corpus = [
    "I love machine learning",
    "Machine learning is fun",
    "I love programming"
]
```

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

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

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

# Convert the TF-IDF matrix to a dense format and display it
dense = tfidf_matrix.todense()
denselist = dense.tolist()

# Create a DataFrame for better readability
df_tfidf = pd.DataFrame(denselist, columns=feature_names)

print(df_tfidf)
```

Advantages of TF-IDF

1. **Highlights Unique Words**: TF-IDF helps identify words that are significant in a specific document but not common across all documents. This is particularly useful for information retrieval and text mining.

2. **Provides Feature Representation**: It converts text into a numerical format that can be used in machine learning algorithms.

3. **Effective for Text Classification**: When building models to classify documents based on their content, TF-IDF can be used as a feature representation.

4. **Useful for Information Retrieval**: In search engines, TF-IDF helps rank documents based on their relevance to a query.

5. **Identifies Key Terms for Topic Modeling**: TF-IDF can help identify the main topics of a document by highlighting the most significant terms.

Disadvantages of TF-IDF

1. **Ignores Word Order and Grammar**: TF-IDF represents text as a bag of words, disregarding the order of words and the grammar of the text.

2. **Sensitive to Document Length**: TF-IDF scores can be biased towards longer documents, as they tend to have more unique words.

3. **Sparsity in High-Dimensional Spaces**: When dealing with a large vocabulary, the TF-IDF matrix becomes very sparse, which can lead to computational inefficiencies.

4. **Lack of Semantic Understanding**: TF-IDF does not capture the semantic relationships between words, which can be important for certain NLP tasks.

5. **Requires Careful Handling of Rare Words**: Rare words can have high TF-IDF scores, which may not always be meaningful. Proper handling of rare words is necessary to avoid overfitting.

How to Check Which Word is More Important

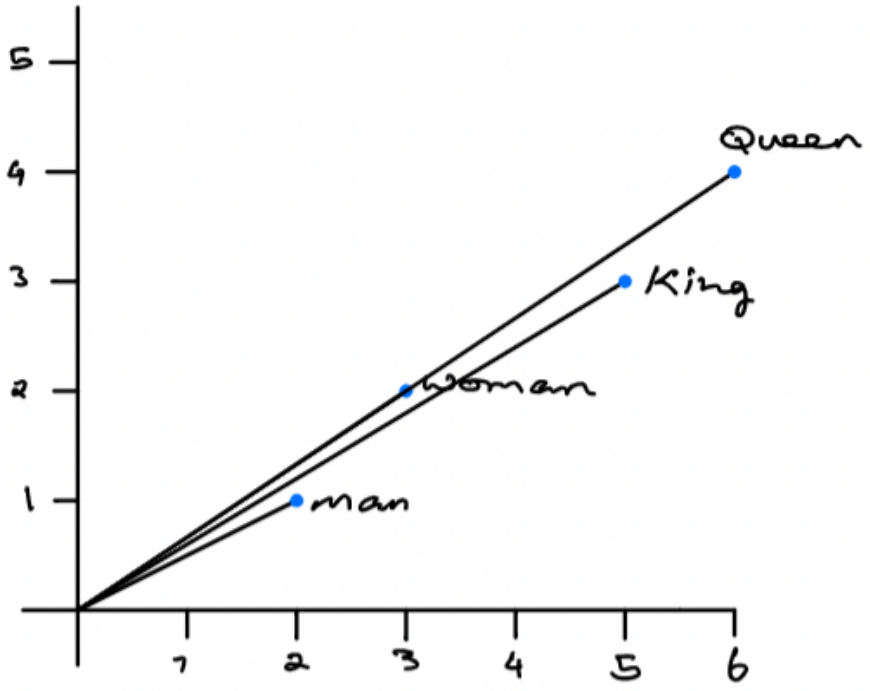
- **Importance in a Document**: The word with the highest TF-IDF score in a document is the most important word in that document.
- **Cross-Document Importance**: If a word consistently has a high TF-IDF score across multiple documents, it could be important across the entire corpus.

What is Important and What TF-IDF is Showing

- **TF-IDF Scores**: The TF-IDF score of a word in a document tells you how relevant that word is to the document, considering the word's frequency in other documents.
  - **High TF-IDF**: Indicates the word is relatively unique to the document.
  - **Low TF-IDF**: Indicates the word is common across many documents in the corpus, hence less significant.

Word2Vec: An Overview

```
King    -    Man    +    Woman    =    Queen
[5,3]   -    [2,1]  +    [3, 2]   =    [6,4]
```



Word2Vec is a technique used in natural language processing (NLP) to convert words into dense vectors of numbers. These vectors (or embeddings) capture the semantic meaning of words, meaning that words with similar meanings have similar vector representations.

Why Do We Need Word2Vec?

Traditional methods of representing words, like one-hot encoding, suffer from several limitations:

- **No Semantic Information**: Each word is treated as a distinct entity with no relationship to other words. For example, "king" and "queen" would be entirely different, with no inherent similarity.
- **High Dimensionality**: The vector size is equal to the size of the vocabulary, which can be enormous, leading to high-dimensional sparse vectors.
- **Lack of Generalization**: Since each word is represented independently, these representations don't generalize well to unseen data.

Word2Vec addresses these issues by capturing semantic relationships between words in a continuous vector space, where similar words are closer together.

Importance and Applications of Word2Vec

Word2Vec embeddings have numerous applications in NLP:

- **Text Classification**: Improve the accuracy of models by providing meaningful word representations.
- **Sentiment Analysis**: Capture the sentiment of words based on their context.
- **Machine Translation**: Assist in mapping words from one language to another by capturing their meaning.
- **Information Retrieval**: Improve search engines by understanding the context and meaning of queries.
- **Similarity Measures**: Identify words or documents that are similar in meaning.

How Word2Vec Works:

Word2Vec uses two main model architectures to generate word embeddings:

1. **Continuous Bag of Words (CBOW)**
2. **Skip-Gram**

Both architectures involve a neural network that learns word representations based on the context in which words appear.

Continuous Bag of Words (CBOW) Model

How CBOW Works:

- **Objective**: Given context words, predict the target word (center word).

- **Architecture**:
  - **Input**: Context words around a target word.
  - **Embedding Layer**: The context words are passed through an embedding layer, which converts them into dense vectors.
  - **Averaging**: The vectors are averaged to create a single vector representation.
  - **SoftMax Layer**: This averaged vector is passed through a SoftMax layer to predict the target word.

Example:
Consider the sentence: "The cat sat on the mat."

If the context window size is 2, and the target word is "sat", the context words would be ["The", "cat", "on", "the"].

The CBOW model will take these context words as input and try to predict the word "sat".

Code Example:

```
from gensim.models import Word2Vec

# Example sentence
sentences = [["the", "cat", "sat", "on", "the", "mat"],
             ["the", "dog", "barked", "at", "the", "cat"]]

# Train the CBOW model
cbow_model = Word2Vec(sentences, vector_size=100, window=2, min_count=
1, sg=0)

# Get the embedding for a word
print(cbow_model.wv['sat'])
```

- If `sg=1`, the model uses **Skip-Gram**.
- If `sg=0`, the model uses **CBOW**.

Skip-Gram Model

How Skip-Gram Works:

- **Objective**: Given a target word (center word), predict the context words around it.
- **Architecture**:
  - **Input**: The target word.
  - **Embedding Layer**: The target word is passed through an embedding layer, converting it into a dense vector.
  - **Output**: The model tries to predict the surrounding context words using this vector.

Example:
Using the same sentence, "The cat sat on the mat", if "sat" is the target word, the model will try to predict the context words ["The", "cat", "on", "the"].

Code Example:

```
pythonCopy code
from gensim.models import Word2Vec

# Example sentence
sentences = [["the", "cat", "sat", "on", "the", "mat"],
             ["the", "dog", "barked", "at", "the", "cat"]]

# Train the Skip-Gram model
skipgram_model = Word2Vec(sentences, vector_size=100, window=2, min_cou
nt=1, sg=1)

# Get the embedding for a word
print(skipgram_model.wv['sat'])
```

- If `sg=1`, the model uses **Skip-Gram**.
- If `sg=0`, the model uses **CBOW**.

Advantages of Word2Vec

1. **Captures Semantic Relationships**: Words with similar meanings are closer in vector space.

2. **Efficient and Scalable**: Word2Vec can handle large datasets and produce meaningful embeddings quickly.
3. **Versatile**: Can be used in various NLP tasks, from text classification to machine translation.
4. **Generalizes Well**: Can produce embeddings for words not seen during training through similarity with known words.

Disadvantages of Word2Vec

1. **Context Ignorance**: Word2Vec does not consider the order of words beyond a fixed window size.
2. **Fixed Embeddings**: Each word has a single embedding, regardless of its meaning in different contexts.
3. **Requires Large Corpus**: To generate good-quality embeddings, a large amount of text data is needed.
4. **Computational Complexity**: Training can be computationally expensive for very large datasets.

Word Embedding : (Summary Format)

Evolution of Word Embedding Techniques: A Sequential Overview

1. One-Hot Encoding (OHE)

What It Is:

- Represents each word as a unique binary vector where only one element is "1" (indicating the presence of the word), and all others are "0".

Advantages:

- Simple to implement and understand.
- Unambiguous representation of words.

Disadvantages:

- **High Dimensionality:** The size of vectors equals the vocabulary size, leading to sparse and high-dimensional vectors.
- **No Semantic Information:** No information about the meaning or relationships between words is captured (e.g., "king" and "queen" are as different as "king" and "car").
- **No Context Awareness:** Every occurrence of a word is treated the same, regardless of its context.

Why It Was Improved:

- The inability to capture semantic meaning or relationships between words led to the need for more sophisticated embeddings.

2. Bag of Words (BoW)

What It Is:

- Represents a text (, sentence, document) as a vector of word counts or binary indicators, ignoring word order.

Advantages:

- Simpler and faster to implement for small texts.
- Captures word frequency, which can be important for certain tasks.

Disadvantages:

- **Ignores Word Order:** Completely disregards the sequence of words, losing valuable syntactic information.
- **High Dimensionality:** Similar to OHE, leading to sparse representations.
- **No Context Awareness:** Fails to capture the meaning of words based on their context.

Why It Was Improved:

- BoW doesn’t consider word order or context, which are essential for understanding nuanced meanings in text.

---

3. Term Frequency-Inverse Document Frequency (TF-IDF)

What It Is:

- Weights words based on their frequency in a document and their rarity across the entire corpus. The more unique a word is to a document, the higher its weight.

Advantages:

- **Emphasizes Rare but Important Words:** Important for distinguishing documents in tasks like document classification.

- **Reduces the Influence of Common Words:** More effective than BoW in highlighting the significance of less frequent terms.

Disadvantages:

- **No Semantic Meaning:** Still doesn't capture word relationships or context.
- **Sparse Vectors:** High-dimensional and sparse like BoW and OHE.
- **No Word Order Information:** Similar to BoW, word order is ignored.

Why It Was Improved:

- TF-IDF improves upon BoW by considering word importance but still lacks semantic understanding and context-awareness.

---

4. Word2Vec

What It Is:

- A predictive model that creates dense, low-dimensional vectors for words based on their context. Two architectures are used: Continuous Bag of Words (CBOW) and Skip-Gram.

Advantages:

- **Captures Semantic Relationships:** Words with similar meanings have similar vectors (e.g., "king" and "queen").
- **Dense Vectors:** More compact and informative than sparse representations.
- **Contextual Awareness:** Word2Vec captures the context of a word in a sentence, leading to better semantic understanding.

Disadvantages:

- **Single Sense per Word:** It assigns one vector per word, which doesn't account for polysemy (e.g., "bank" as a financial institution vs. riverbank).
- **No Global Context:** Word2Vec focuses on local context within a window size and might miss out on broader document-level information.

Why It Was Improved:

- Word2Vec was a breakthrough in capturing semantic meanings but lacked the ability to consider polysemy and broader context.

---

5. GloVe (Global Vectors for Word Representation)

What It Is:

- Combines the strengths of both matrix factorization and predictive methods by creating word vectors based on the co-occurrence matrix of words in a corpus.

Advantages:

- **Global Context:** Captures global statistical information from the corpus, giving a more holistic understanding of word relationships.
- **Dense Vectors:** Similar to Word2Vec, produces compact and meaningful vectors.
- **Captures Semantic and Syntactic Relationships:** Effectively captures relationships like analogies (e.g., "king" is to "queen" as "man" is to "woman").

Disadvantages:

- **Requires Large Corpus:** GloVe needs a substantial amount of text data for effective training.
- **Static Embeddings:** Like Word2Vec, GloVe generates one embedding per word, ignoring multiple meanings.

Why It Was Improved:

- Despite its advantages, GloVe couldn't handle polysemy or dynamically change word meanings based on context.

---

6. Contextual Embeddings (e.g., BERT, ELMo)

What It Is:

- Models like BERT (Bidirectional Encoder Representations from Transformers) and ELMo (Embeddings from Language Models) create dynamic word embeddings that change based on the context in which a word appears.

Advantages:

- **Contextual Understanding:** Words are represented differently depending on their surrounding context (e.g., "bank" in "river bank" vs. "financial bank").
- **Handles Polysemy:** Addresses the limitation of static embeddings by creating context-sensitive vectors.
- **Pre-trained Models:** Large pre-trained models are available, which can be fine-tuned for specific tasks, saving time and resources.

Disadvantages:

- **Computationally Intensive:** Training and using these models require significant computational resources.
- **Complexity:** These models are complex and require a deeper understanding of NLP and deep learning to implement effectively.

Why It Was Improved:

- To overcome the limitations of static embeddings and enhance the model's ability to understand nuanced meanings in varied contexts.