Please note : This is just only a reference material . Please read the prescribed books in the syllabus and any other resources for knowledge enhancement in this subject.(NLP)

Text Similarity

It is the process of comparing a piece of text with another and finding the similarity between them. It’s basically about determining the degree of closeness of the text.

Cosine Similarity

Cosine similarity measures the similarity between two vectors of an inner product space in general. It measures the cosine of the angle between two embeddings and determines whether they are pointing in roughly the same direction or not. When the embeddings are pointing in the same direction the angle between them is zero so their cosine similarity is 1. When the embeddings are perpendicular to each other the angle between them is 90 degrees and the cosine similarity is 0 finally when the angle between them is 180 degrees the cosine similarity is -1.

$$similarity(A,B) = \frac{A \cdot B}{\|A\| \times \|B\|} = \frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \times \sqrt{\sum_{i=1}^{n} B_i^2}}$$

Cosine Similarity ranges from 1 to -1, where 1 represents most similar and -1 represents least similar.

In natural language processing (NLP), cosine similarity is a mathematical metric that measures the similarity between two vectors in a multi-dimensional space. It's used to understand the semantic relationships between different pieces of text, such as documents, articles, or customer reviews.

Here are some ways cosine similarity is used in NLP:

- Summarizing documents: Cosine similarity helps understand the context and content similarity between different textual data points.
- Comparing articles: Cosine similarity helps understand the semantic similarity between documents, datasets, or images.
- Determining sentiment: Cosine similarity helps determine the sentiment of customer reviews.

Cosine similarity is used in other applications as well, including: Search algorithms, Recommendation systems, Text mining, Sentiment analysis, and Document clustering.

Example of cosine similarity

Here are two very short texts to compare:

1. `Julie loves me more than Linda loves me`
2. `Jane likes me more than Julie loves me`

We want to know how similar these texts are, purely in terms of word counts (and ignoring word order). We begin by making a list of the words from both texts:

me Julie loves Linda than more likes Jane

Now we count the number of times each of these words appears in each text:

me	2	2
Jane	0	1
Julie	1	1
Linda	1	0
likes	0	1
loves	2	1
more	1	1
than	1	1

So the two vectors are :

A: [2, 0, 1, 1, 0, 2, 1, 1]
B: [2, 1, 1, 0, 1, 1, 1, 1]

The cosine of the angle between them is nearly about 0.822 by applying the formula for cosine similarity given above.

Jaccard Similarity also called as Jaccard Index or Jaccard Coefficient is a simple measure to represent the similarity between data samples. The similarity is computed as the ratio of the length of the intersection within data samples to the length of the union of the data samples.

It is represented as –

`J(A, B) =  |A ∩ B| / |A U B|`

It is used to find the similarity or overlap between the two binary vectors or numeric vectors or strings. It can be represented as J. There is also a closely related term associated with Jaccard Similarity which is called Jaccard Dissimilarity or Jaccard Distance. Jaccard Distance is a measure of dissimilarity between data samples and can be represented as (1 – J) where J is Jaccard Similarity.

The Jaccard Similarity score ranges from 0 to 1, where 1 represents most and 0 represents least similar.

For example :

Let us say d1 and d2 are vectors.

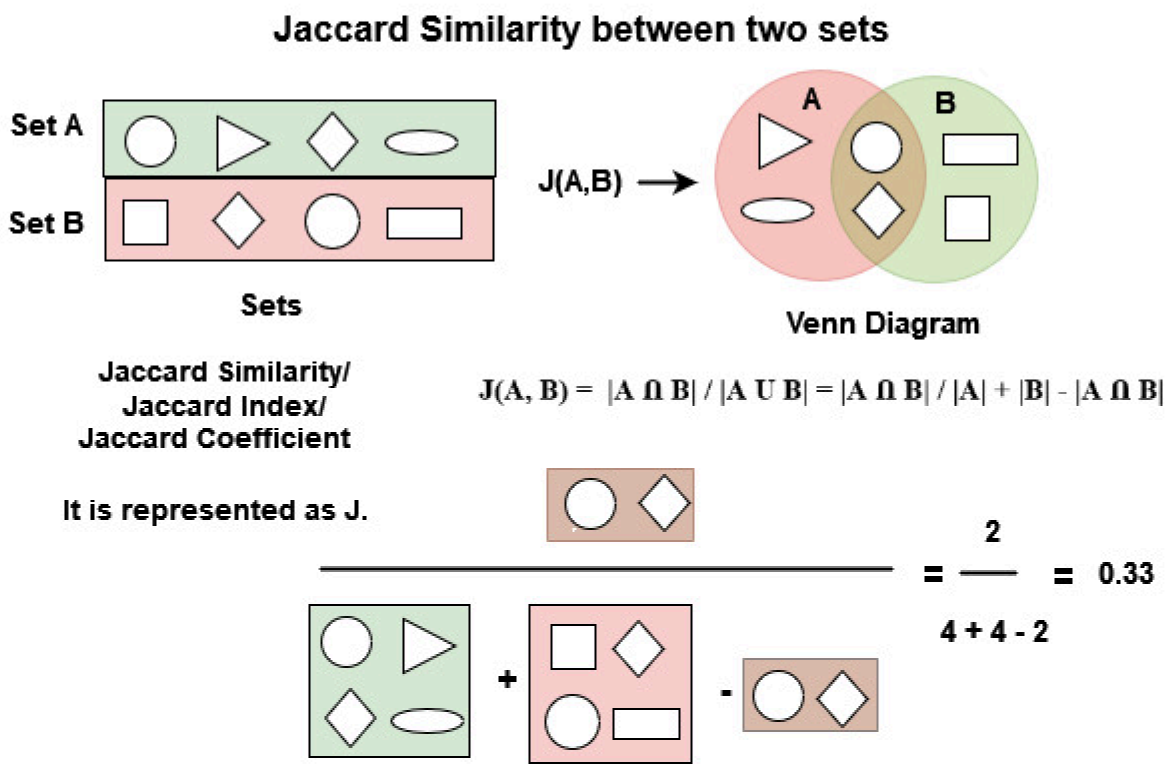
d1 = [1 3 2]

d2 = [5 0 3]

In this case, d1 ∩ d2 is: [3] and d1 ∪ d2 is [1 3 2 5 0]

Jaccard similarity between d1 and d2 is 1/5 = 0.2

Another example of Jaccard similarity :



Note : Cosine similarity measures the similarity between two vectors, while Jaccard similarity Is normally used to measures the similarity between two sets or two binary vectors.

TF-IDF:

The BoW method is simple and works well, but it creates a problem because it treats all words equally. As a result, we can’t distinguish very common words or rare words when BoW is used. TF-IDF comes into play at this stage, to solve this problem.

Unlike the bag of words model, the TF-IDF representation takes into account the importance of each word in a document. The term **TF** stands for **term frequency**, and the term **IDF** stands for **inverse document frequency**.

To understand TF-IDF, firstly we should cover the two terms separately:

- Term frequency (TF)

- Inverse document frequency (IDF)

Term Frequency (TF)

Term frequency refers to the frequency of a word in a document. For a specified word, it is defined as the ratio of the number of times a word appears in a document to the total number of words in the document.

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

— t is the word or token.

— d is the document.

Inverse document frequency (IDF)

Inverse document frequency measures the importance of the word in the corpus. It measures how common a particular word is across all the documents in the corpus.

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

TF-IDF Score

The TF-IDF score for a term in a document is calculated by multiplying its TF and IDF values. This score reflects how important the term is within the context of the document and across the entire corpus. Terms with higher TF-IDF scores are considered more significant.

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

How to compute TF-IDF (Example of TF - IDF)

Suppose we are looking for documents using the query *Q* and our database is composed of the documents *D1*, *D2,* and *D3*.

- *Q*: The cat.
- *D1*: The cat is on the mat.
- *D2*: My dog and cat are the best.
- *D3*: The locals are playing.

There are several ways of calculating TF, with the simplest being a raw count of instances a word appears in a document. We'll compute the TF scores using the ratio of the count of instances over the length of the document.

TF(word, document) = “number of occurrences of the word in the document” / “number of words in the document”

Let’s compute the TF scores of the words “the” and “cat” (i.e. the query words) with respect to the documents *D1*, *D2,* and *D3*.

TF(“the”, D1) = 2/6 = 0.33

TF(“the”, D2) = 1/7 = 0.14

TF(“the”, D3) = 1/4 = 0.25

TF(“cat”, D1) = 1/6 = 0.17

TF(“cat”, D2) = 1/7 = 0.14

TF(“cat”, D3) = 0/4 = 0

IDF can be calculated by taking the total number of documents, dividing it by the number of documents that contain a word, and calculating the logarithm. If the word is very common and appears in many documents, this number will approach 0. Otherwise, it will approach 1.

IDF(word) = log(number of documents / number of documents that contain the word)

Let’s compute the IDF scores of the words “the” and “cat”.

IDF(“the”) = log(3/3) = log(1) = 0

IDF(“cat”) = log(3/2) = 0.18

Multiplying TF and IDF gives the TF-IDF score of a word in a document. The higher the score, the more relevant that word is in that particular document.

TF-IDF(word, document) = TF(word, document) * IDF(word)

Let’s compute the TF-IDF scores of the words “the” and “cat”.

TF-IDF(“the”, D1) = 0.33 * 0 = 0

TF-IDF(“the, D2) = 0.14 * 0 = 0

TF-IDF(“the”, D3) = 0.25 * 0 = 0

TF-IDF(“cat”, D1) = 0.17 * 0.18= 0.0306

TF-IDF(“cat, D2) = 0.14 * 0.18= 0.0252

TF-IDF(“cat”, D3) = 0 * 0 = 0

We can use the average TF-IDF word scores over each document to get the ranking of *D1*, *D2,* and *D3* with respect to the query *Q*.

Average TF-IDF of D1 = (0 + 0.0306) / 2 = 0.0153

Average TF-IDF of D2 = (0 + 0.0252) / 2 = 0.0126

Average TF-IDF of D3 = (0 + 0) / 2 = 0

Looks like the word “the” does not contribute to the TF-IDF scores of each document. This is because “the” appears in all of the documents and thus it is considered a not-relevant word.

As a conclusion, when performing the query “The cat” over the collection of documents *D1*, *D2,* and *D3*, the ranked results would be:

1. *D1*: The cat is on the mat.
2. *D2*: My dog and cat are the best.
3. *D3*: The locals are playing.

Term Document Incidence Matrix

The term-document incidence matrix is one of the basic techniques to represent text data where,

> *We get the* ***unique words*** *across all the documents.*

> *For each document, we* ***add 1 if the term exists*** *in the document otherwise fill 0 in the cell.*

Consider below sentences,

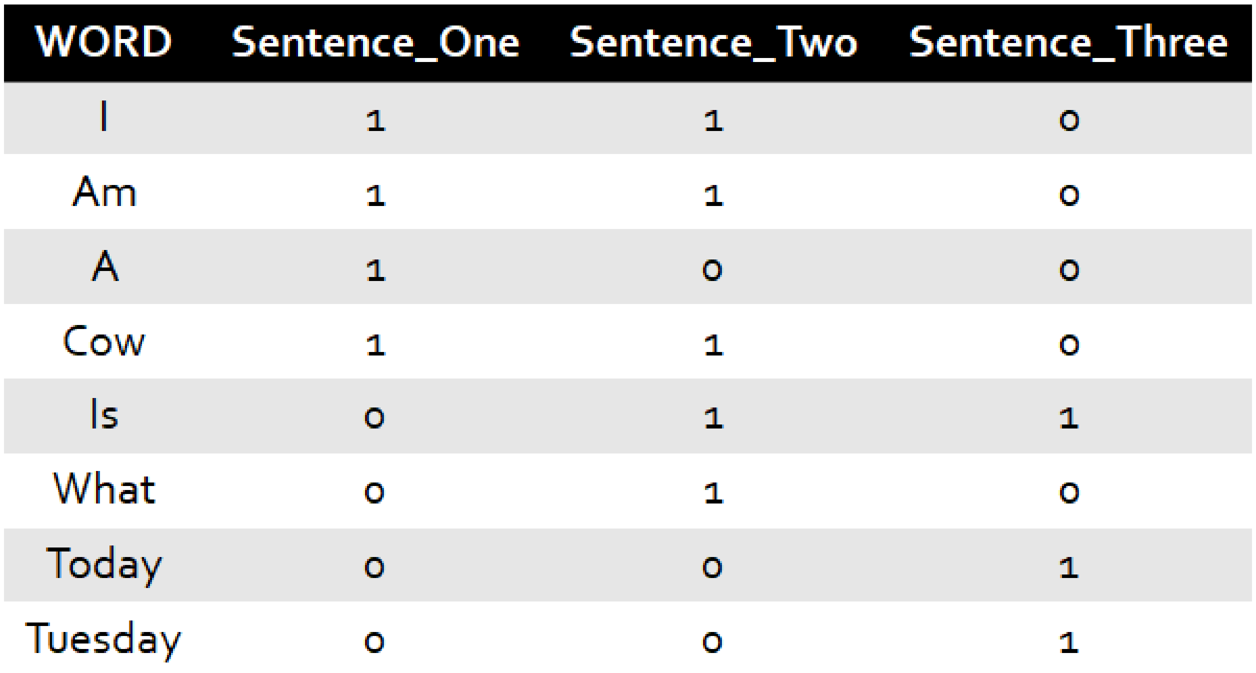
1. I am a cow.
2. Cow is what I am.
3. Today is Tuesday.

Now, if I ask you a question — Can you tell the sentences which contain the term 'cow' but not 'Tuesday'?

As a human, it is easy for us to say that the answer will be sentence 1 and sentence 2.

But how to model this problem mathematically so that it can be solved by a machine?

For the sentences, which we took in our problem statement, Term-Document Incidence Matrix will look something like this :



WORD	Sentence_One	Sentence_Two	Sentence_Three
I	1	1	0
Am	1	1	0
A	1	0	0
Cow	1	1	0
Is	0	1	1
What	0	1	0
Today	0	0	1
Tuesday	0	0	1

Term-Document Incidence Matrix for the sentences — 1, 2 and 3.

Note : Words are normalized i.e. same word is not considered twice across all the documents/sentences.

Boolean Retrieval Model

It is one of the application of this matrix where we can answer any query which is in the form of a ***Boolean expression*** of terms, that is, in which terms are combined with the operators ***and, or,*** and ***not***.

For our query i.e. get the sentences which contain the term ‘cow’ but not ‘tuesday’,

> We will ***get the term vector***, which is basically, the values from the row containing the term in Term-Document Matrix. Example — For Cow, the vector will be [1,1,0].
> Perform a ***Bitwise AND operation*** between the vectors of the terms provided in the input query.

Let’s apply the algorithm and see if we get the right answer.

1. Cow Vector = [1,1,0]
2. Tuesday Vector = [0,0,1].
3. Not Tuesday Vector = [1,1,0]. ***Not*** *vector can be obtained by taking* **compliment** *of the original vector.*

Perform BITWISE AND OPERATION :

[1,1,0] & [1,1,0] => **[1,1,0]**

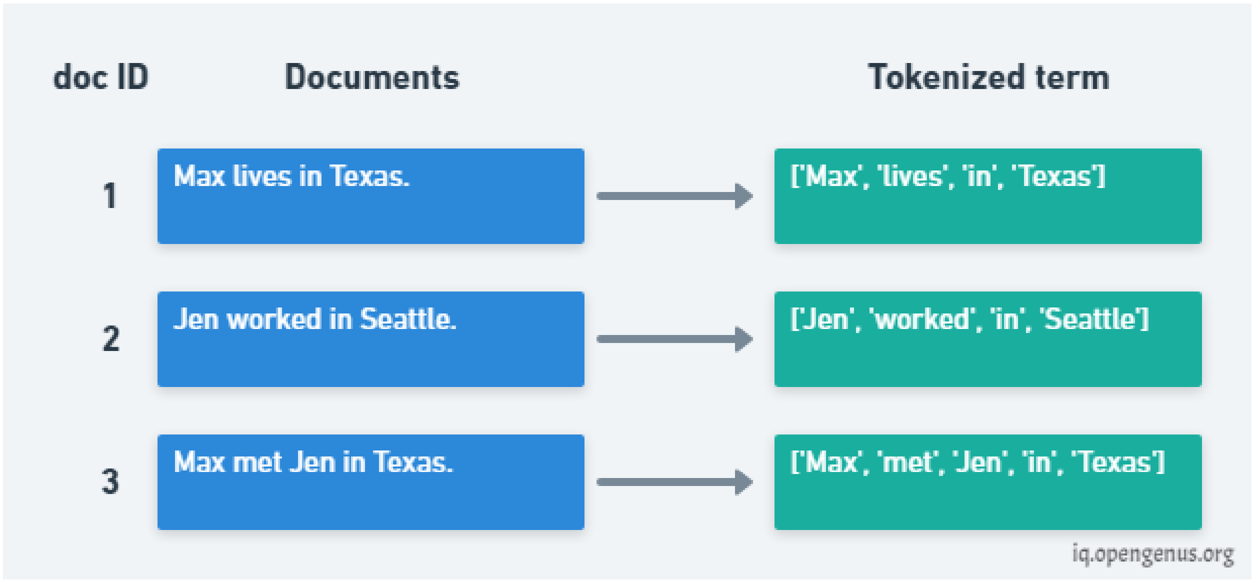
Inference from the result :

In the result obtained from BITWISE AND operation, **the indices for which 1 is present**, those sentence satisfy the input query. Hence, sentence one and two contain the word ‘cow’ but not ‘tuesday’ and will be returned as result for the query.

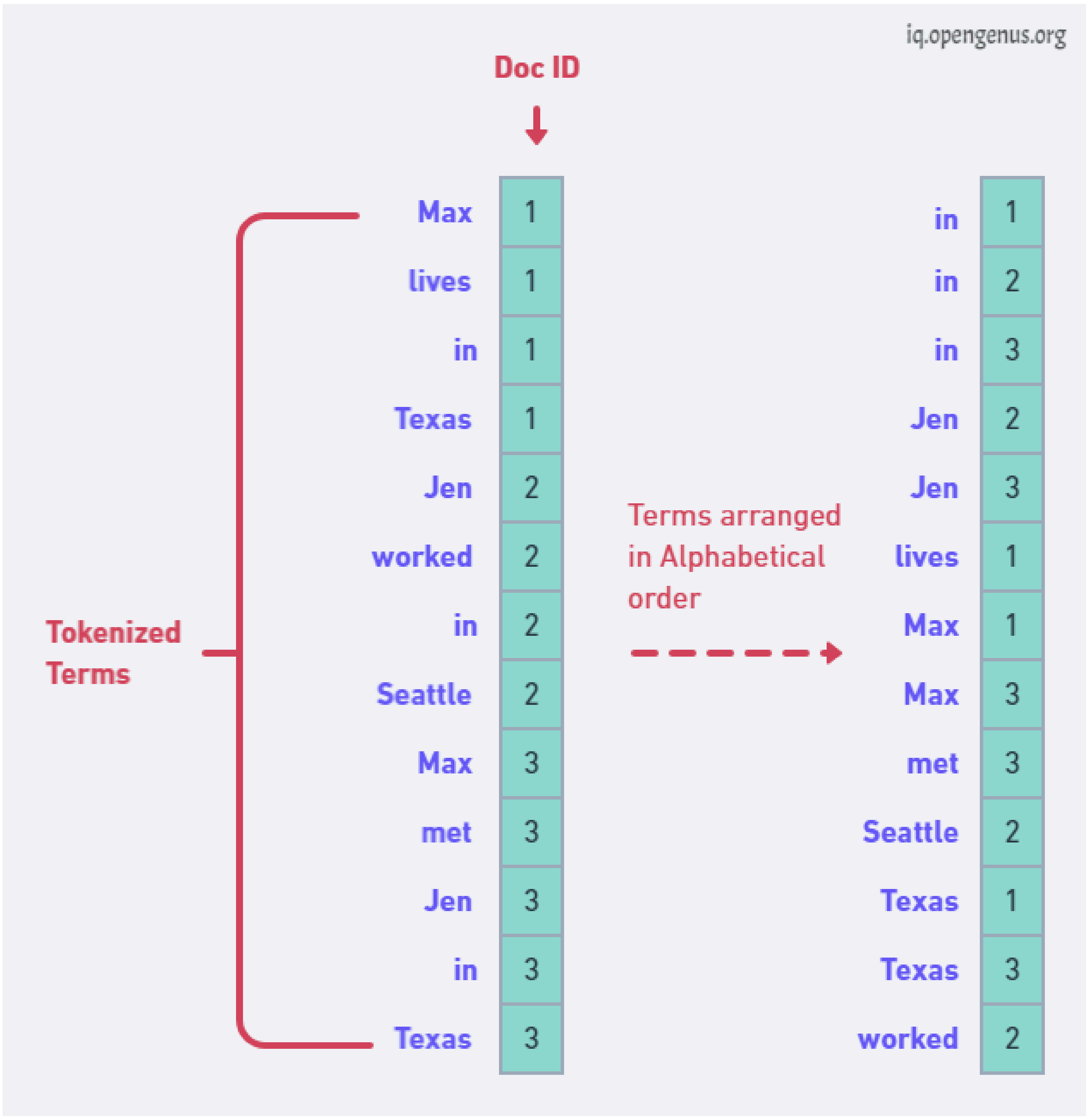
Inverted index

In this method, a vector is formed where each document is given a document ID and the terms act as pointers. Then sorting of the list is done in alphabetical order and pointers are maintained to their corresponding document ID.

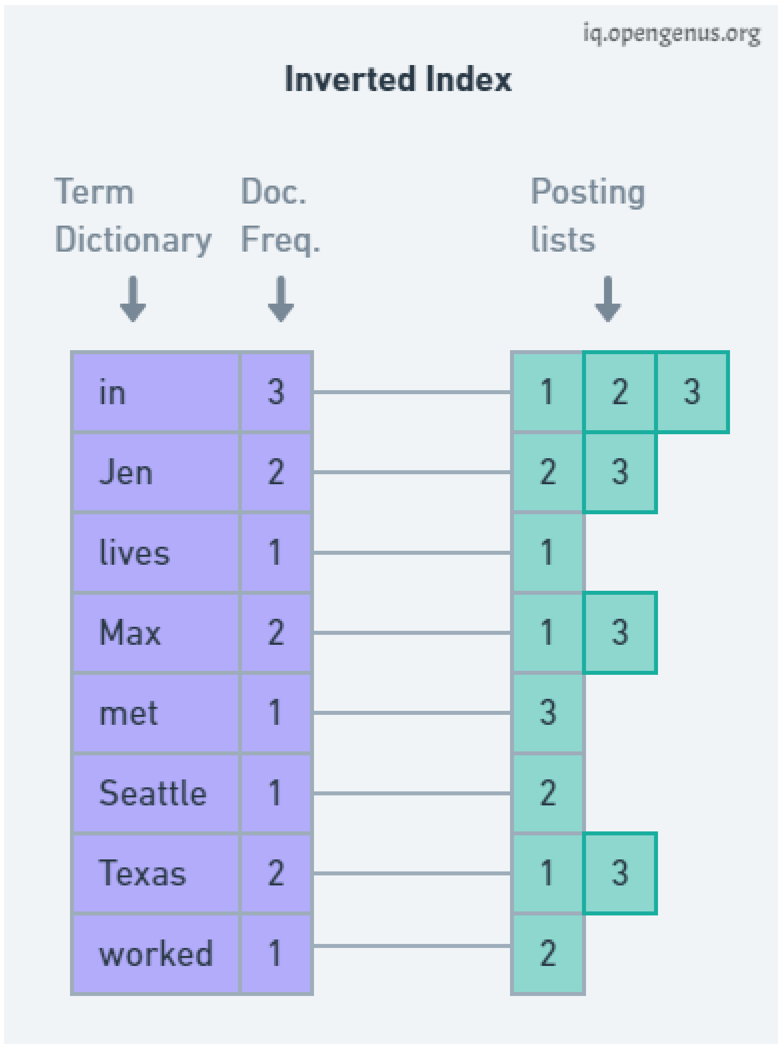
For example:-


doc ID
Documents
Tokenized term
1
Max lives in Texas.
['Max', 'lives', 'in', 'Texas']
2
Jen worked in Seattle.
['Jen', 'worked', 'in', 'Seattle']
3
Max met Jen in Texas.
['Max', 'met', 'Jen', 'in', 'Texas']
iq.opengenus.org

Formation of vector



Finally, an inverted index structure is created. Then an array-like structure is formed containing the doc ID and the terms grouped together.



Term Dictionary	Doc. Freq.	Posting lists
in	3	1 2 3
Jen	2	2 3
lives	1	1
Max	2	1 3
met	1	3
Seattle	1	2
Texas	2	1 3
worked	1	2

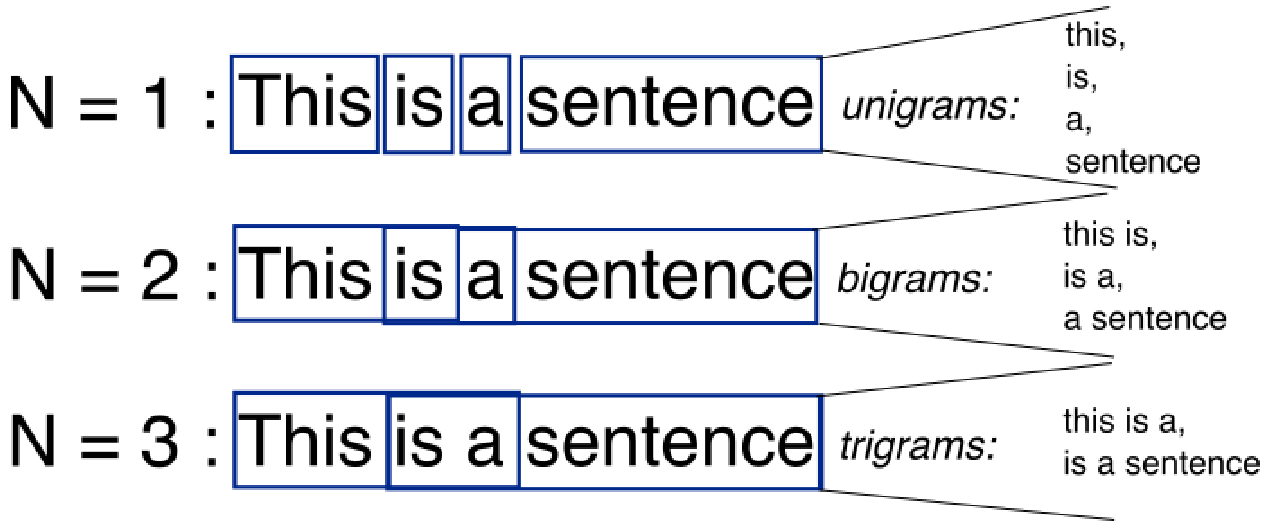
N-Gram :

N-grams are contiguous sequences of 'n' items, typically words in the context of NLP. These items can be characters, words, or even syllables, depending on the granularity desired. The value of 'n' determines the order of the N-gram.

Examples:

- **Unigrams (1-grams): Single words, e.g., "cat," "dog."**
- **Bigrams (2-grams): Pairs of consecutive words, e.g., "natural language," "deep learning."**

- **Trigrams (3-grams): Triplets of consecutive words, e.g., "machine learning model," "data science approach."**
- **4-grams, 5-grams, etc.: Sequences of four, five, or more consecutive words.**



Significance of N-grams in NLP:

1. Capturing Context and Semantics:

- **N-grams help capture the contextual information and semantics within a sequence of words, providing a more nuanced understanding of language.**

2. Improving Language Models:

- **In language modeling tasks, N-grams contribute to building more accurate and context-aware models, enhancing the performance of applications such as machine translation and speech recognition.**

3. Enhancing Text Prediction:

- **N-grams are essential for predictive text applications, aiding in the prediction of the next word or sequence of words based on the context provided by the preceding N-gram.**

4. Information Retrieval:

- **In information retrieval tasks, N-grams assist in matching and ranking documents based on the relevance of N-gram patterns.**

5. Feature Extraction:

- **N-grams serve as powerful features in text classification and sentiment analysis, capturing meaningful patterns that contribute to the characterization of different classes or sentiments.**

Applications of N-grams in NLP:

1. Speech Recognition:

- **N-grams play a crucial role in modeling and recognizing spoken language patterns, improving the accuracy of speech recognition systems.**

2. Machine Translation:

- **In machine translation, N-grams contribute to understanding and translating phrases within a broader context, enhancing the overall translation quality.**

3. Predictive Text Input:

- **Predictive text input on keyboards and mobile devices relies on N-grams to suggest the next word based on the context of the input sequence.**

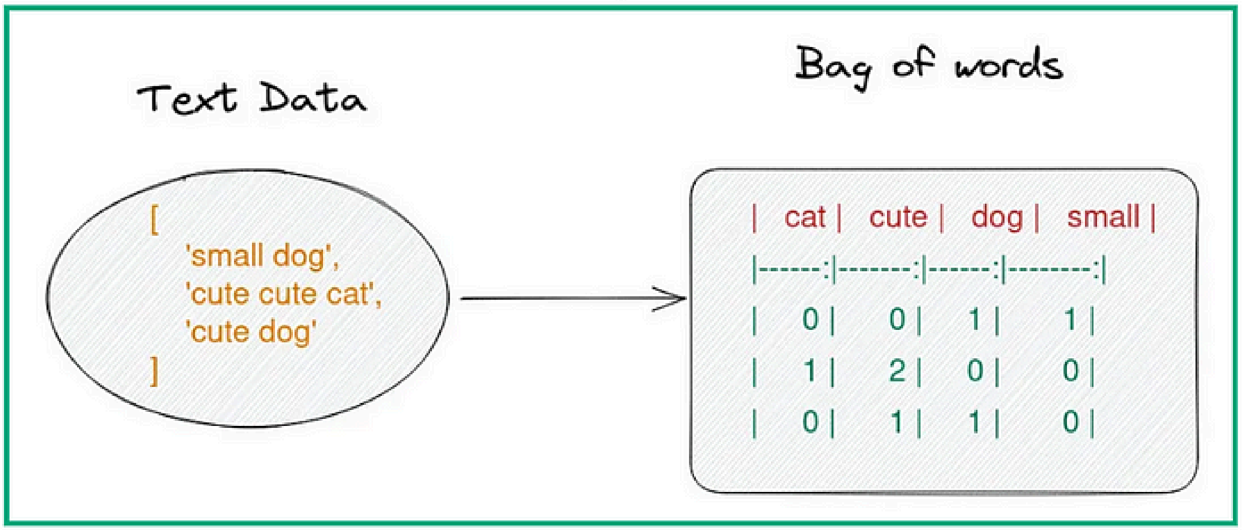
4. Named Entity Recognition (NER):

- **N-grams aid in identifying and extracting named entities from text, such as names of people, organizations, and locations.**

5. Search Engine Algorithms:

- **Search engines use N-grams to index and retrieve relevant documents based on user queries, improving the accuracy of search results.**

Bag of Words Model in NLP



cat	cute	dog	small
0	0	1	1
1	2	0	0
0	1	1	0

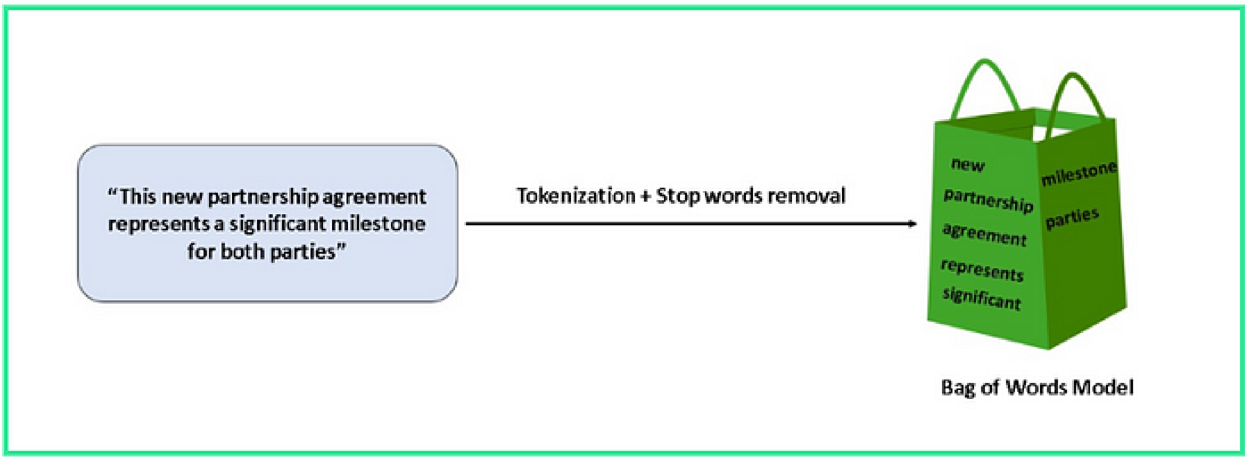
Bag of Words (BoW) is a Natural Language Processing strategy for converting a text document into numbers that can be used by a computer program. This method involves converting text into a vector based on the frequency of words in the text, without considering the order or context of the words.

Imagine a social media platform that aims to analyze customer reviews and understand the popularity of services among users. This platform decides to employ the **Bag of Words** method for processing customer reviews.

Data Collection: The first step involves collecting and storing customer reviews, which consist of text written by customers about various services.

Preprocessing: Text data is cleaned by removing punctuation marks, numbers and unnecessary whitespace.

Creating a Word List: A word list is created for BoW. This list includes all the unique words in the dataset.


"This new partnership agreement represents a significant milestone for both parties"
Tokenization + Stop words removal
new
partnership
agreement
represents
significant
milestone
parties
Bag of Words Model

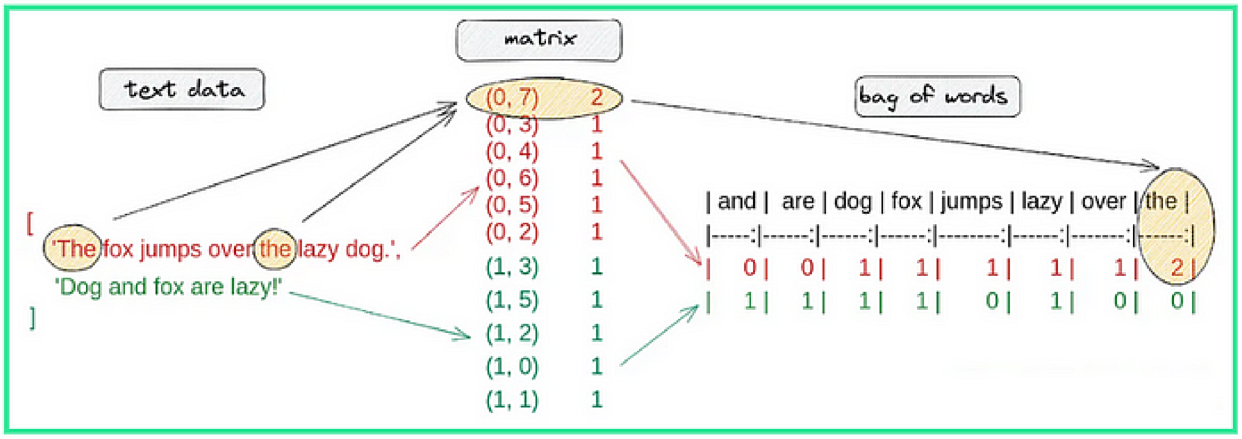
Text Representation: Each customer review is represented using the BoW method. The frequency of each word is recorded within a vector based on its position in the word list. For example, the BoW representation for the phrase **"great service"** could be as follows: [service: 1, great: 1, other_words: 0].

Analysis and Classification: With this representation method, the platform can analyze how popular services are among customers and identify which services receive positive or negative reviews. For instance, if a service’s BoW representation frequently includes positive terms like “**high quality**” and “**affordable**,” it can be inferred that the service receives positive feedback.

Improvement: Based on the results obtained, the platform can take steps to optimize its service portfolio and enhance the overall customer experience.

In this way, the BoW method enables the social media platform to analyze customer reviews, monitor service performance and make improvements effectively.

Let’s apply the above steps to a sample text.



Parts of Speech (POS) Tagging in Natural Language Processing.

Part-of-Speech (POS) tagging is a fundamental task in Natural Language Processing (NLP) that involves assigning a grammatical category (such as noun, verb, adjective, etc.) to each word in a sentence. The goal is to understand the syntactic structure of a sentence and identify the grammatical roles of individual words. POS tagging provides essential information for various NLP applications, including text analysis, machine translation, and information retrieval.

Key Concepts:

POS Tags:

- POS tags are short codes representing specific parts of speech. Common POS tags include:
- Noun (NN)
- Verb (VB)
- Adjective (JJ)
- Adverb (RB)
- Pronoun (PRP)
- Preposition (IN)
- Conjunction (CC)
- Determiner (DT)
- Interjection (UH)

There are other POS tags also.

Example:

Consider the sentence: “The quick brown fox jumps over the lazy dog.”

POS tagging might yield:

```
[('The', 'DT'), ('quick', 'JJ'), ('brown', 'NN'), ('fox',
'NN'), ('jumps', 'VBZ'), ('over', 'IN'), ('the', 'DT'),
('lazy', 'JJ'), ('dog', 'NN')]
```

In this example, each word is assigned a POS tag indicating its grammatical category in the sentence.

— —

Applications where POS tagging plays a crucial role:

1. **Syntactic Parsing:**

- **Application:** Understanding the grammatical structure of sentences.

- **Role of POS Tagging**: POS tags provide information about the syntactic role of each word, aiding in syntactic parsing and tree construction.

2. **Named Entity Recognition (NER):**

- **Application:** Identifying and classifying entities (e.g., persons, organizations, locations) in text.
- **Role of POS Tagging:** POS tags help in identifying proper nouns, which are often indicative of named entities.

3. **Information Retrieval:**

- **Application:** Improving search and retrieval of relevant documents or information.
- **Role of POS Tagging:** Using POS tags, one can prioritize or filter search results based on the grammatical category of words. For instance, focusing on nouns for certain queries.

4. **Text Summarization:**

- **Application:** Generating concise summaries of longer texts.
- **Role of POS Tagging:** Understanding the syntactic structure helps in identifying key elements and relationships in the text, aiding in the creation of coherent summaries.

5.**Machine Translation:**

- **Application:** Translating text from one language to another.
- **Role of POS Tagging:** POS tags provide information about the grammatical structure, aiding in accurate translation by preserving the syntactic and grammatical nuances of the source language.

6.**Sentiment Analysis:**

- **Application:** Determining the sentiment expressed in a piece of text (positive, negative, neutral).
- **Role of POS Tagging:** Identifying adjectives and verbs in particular helps in capturing the sentiment expressed by the author.

7.**Question Answering Systems:**

- **Application:** Generating accurate answers to user queries.
- **Role of POS Tagging:** Understanding the grammatical structure of questions helps in extracting key information and formulating appropriate answers.

8.**Text-to-Speech Synthesis:**

- **Application:** Converting written text into spoken language.

- **Role of POS Tagging:** POS tags guide the synthesis process, ensuring that the spoken output follows appropriate intonation and emphasis based on the grammatical structure.

9. **Speech Recognition:**

- **Application:** Converting spoken language into written text.
- **Role of POS Tagging:** POS tags contribute to language models used in speech recognition, aiding in predicting the likely sequence of words based on their grammatical roles.

10. **Grammar Checking:**

- **Application:** Identifying and correcting grammatical errors in written text.
- **Role of POS Tagging**: POS tags help in detecting errors related to word usage, agreement, and syntactic structure.

— -

Program for listing all POS Tags

```
import nltk
nltk.download('tagsets')
nltk.help.upenn_tagset()
```

Program for determining POS Tag in a sentence

```
# Parts of Speech Tagging
import nltk
from nltk.tokenize import word_tokenize
# Download NLTK tokenizer and POS tagging models
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')
def pos_tagging(text):
# Tokenize the text into words
words = word_tokenize(text)
# Perform POS tagging
tagged_words = nltk.pos_tag(words)
return tagged_words
# Example text
text = "NLTK is a leading platform for building Python programs to work
with human language data."
# Perform POS tagging
tagged_text = pos_tagging(text)
# Print POS tagged text
print(tagged_text)
```

Output :

```
[('NLTK', 'NNP'), ('is', 'VBZ'), ('a', 'DT'), ('leading', 'VBG'),
('platform', 'NN'), ('for', 'IN'), ('building', 'VBG'), ('Python', 'NNP'),
('programs', 'NNS'), ('to', 'TO'), ('work', 'VB'), ('with', 'IN'),
('human', 'JJ'), ('language', 'NN'), ('data', 'NNS'), ('.', '.')]
```