Natural Language Processing
Course code: CSE3015

Module 1
Introduction to NLP

Prepared by

Dr. Venkata Rami Reddy Ch

SCOPE

Syllabus

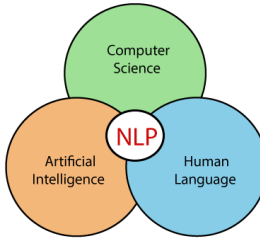
- Overview:
  - Origins and challenges of NLP
  - Need of NLP
- Preprocessing techniques-
  - Text Wrangling, Text cleansing, sentence splitter, tokenization, stemming, lemmatization, stop word removal, rare word removal, spell correction.
- Word Embeddings, Different Types :

  One Hot Encoding, Bag of Words (BoW), TF-IDF
- Static word embeddings:

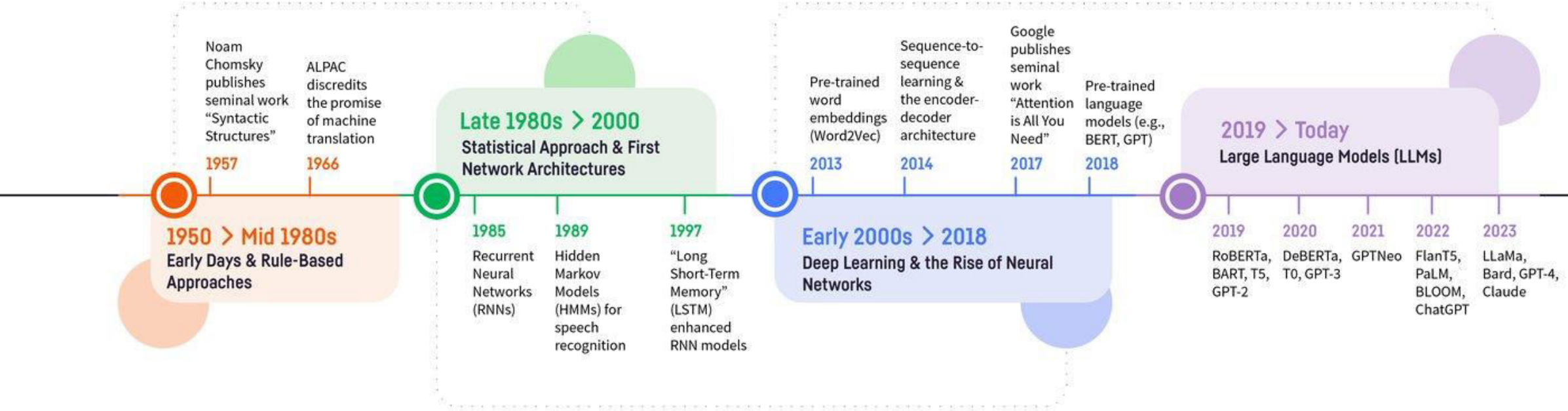
  Word2vec, GloVe, FastText

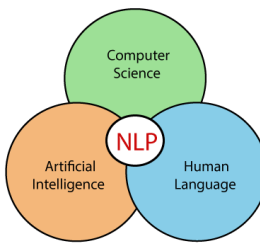
Introduction

- NLP stands for **Natural Language Processing**, which is a part of **Computer Science, Human language,** and **Artificial Intelligence**.
- It is the technology that is used by machines to understand, analyse, manipulate, and interpret human's languages.
- The goal is to enable machines to understand, interpret, generate, and respond to human language in a way that is both meaningful and useful.
- NLP combines concepts from linguistics, computer science, and machine learning to bridge the gap between human communication and machine understanding



The History of NLP





1950s: Beginnings of NLP

- **1950**: Alan Turing publishes *"Computing Machinery and Intelligence"*, introducing the Turing Test to evaluate a machine's ability to exhibit intelligent behavior equivalent to or indistinguishable from that of a human.
- **1954**: The Georgetown-IBM experiment demonstrates machine translation, translating 60 Russian sentences into English. This is one of the earliest NLP experiments.
- **1957**: Noam Chomsky introduces transformational grammar in *"Syntactic Structures"*, laying the theoretical foundation for many linguistic models.

1960s: Rule-Based Systems

- **1964-1966**: ELIZA, one of the first chatbots, is developed by Joseph Weizenbaum, demonstrating simple natural language understanding through pattern matching.
- **1966**: The ALPAC Report criticizes machine translation efforts, leading to reduced funding for NLP in the U.S.

1970s: Emergence of Parsing and Semantics

- **1970**: William A. Woods develops the Augmented Transition Network (ATN) for parsing natural language.
- **1972**: SHRDLU, developed by Terry Winograd, showcases NLP capabilities in a virtual blocks world, integrating syntax, semantics, and reasoning.
- **1980s: Statistical Methods**
- **1980**: Introduction of the concept of probabilistic language models, moving beyond purely rule-based systems.
- **1983**: The development of WordNet by George Miller begins, creating a semantic network for English.
- **1990s: Statistical and Machine Learning Approaches**
- **1990s**: Hidden Markov Models (HMMs) and n-gram models become widely used for tasks like speech recognition and part-of-speech tagging.
- **1996**: IBM develops BLEU, a metric for evaluating machine translation quality.
- **1999**: Latent Semantic Analysis (LSA) emerges for information retrieval and document similarity.

2000s: Rise of Probabilistic Models and Tools

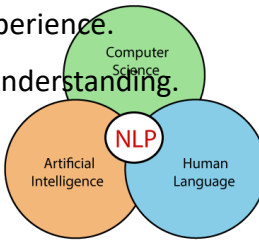
- **2001**: Conditional Random Fields (CRFs) are introduced for sequence labeling tasks.
- **2003**: Stanford Parser is released, providing tools for syntactic analysis.

2010s: Deep Learning Revolution

- **2013**: Word2Vec, introduced by Google, revolutionizes word embeddings using neural networks.
- **2014**:
  - GloVe (Global Vectors for Word Representation) is introduced by Stanford.
  - Seq2Seq models, a foundation for machine translation, gain popularity.
- **2017**:
  - The Transformer architecture is introduced in the paper *"Attention is All You Need"*, laying the foundation for modern NLP models.
  - ELMo (Embeddings from Language Models) demonstrates contextualized embeddings.
- **2018**:
  - OpenAI introduces GPT (Generative Pre-trained Transformer).
  - BERT (Bidirectional Encoder Representations from Transformers) is introduced by Google, setting new state-of-the-art results for many NLP tasks.

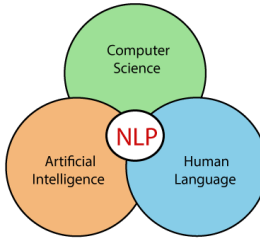
2020s: Large Language Models and Multimodal Systems

- **2020**: GPT-3, a 175-billion parameter model by OpenAI, demonstrates unprecedented language generation capabilities.
- **2021**: Multimodal models like CLIP and DALL-E combine text and images.
- **2022**: ChatGPT, based on GPT-3.5, provides a conversational AI experience.
- **2023**: Models like GPT-4 improve multi-modality, reasoning, and understanding.



Need of NLP

- Bridging the Gap Between Humans and Machines
- NLP enables interaction between these two entities by allowing machines to process, understand, and respond to human language.
- Examples: Virtual assistants like Siri and Alexa, customer service chatbots.



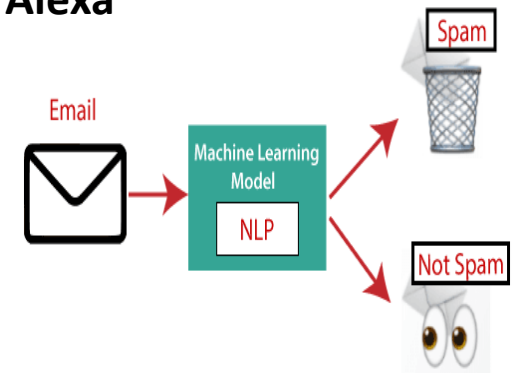
Need of NLP/Application of NLP

- Email platforms, such as Gmail, Outlook, etc., use NLP extensively to provide a range of product features, such as spam classification,calendar event extraction, auto-complete, etc.
- Voice-based assistants, such as Apple Siri, Google Assistant, Microsoft Cortana, and Amazon Alexa rely on a range of NLP techniques to interact with the user, understand user commands, and respond accordingly.
- Modern search engines, such as Google and Bing, use NLP heavily for various subtasks, such as query understanding, query expansion, question answering, information retrieval, and grouping of the results, etc.
- Machine translation services, such as Google Translate, Bing Microsoft Translator, and Amazon Translate are used in to solve a wide range of scenarios and business use cases.
- NLP forms the backbone of spelling- and grammar-correction tools, such as Grammarly and spell check in Microsoft Word and Google Docs.

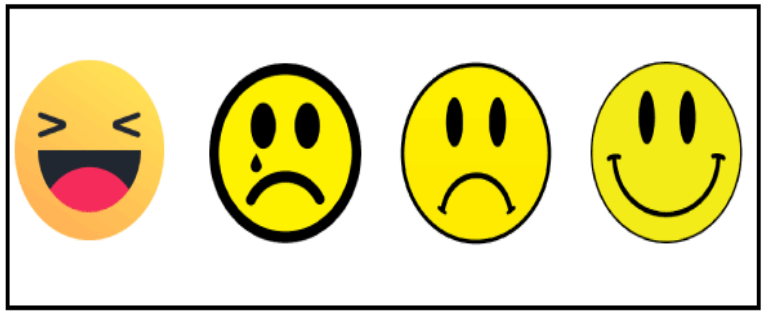
Need of NLP/Application of NLP

Google Home , Alexa

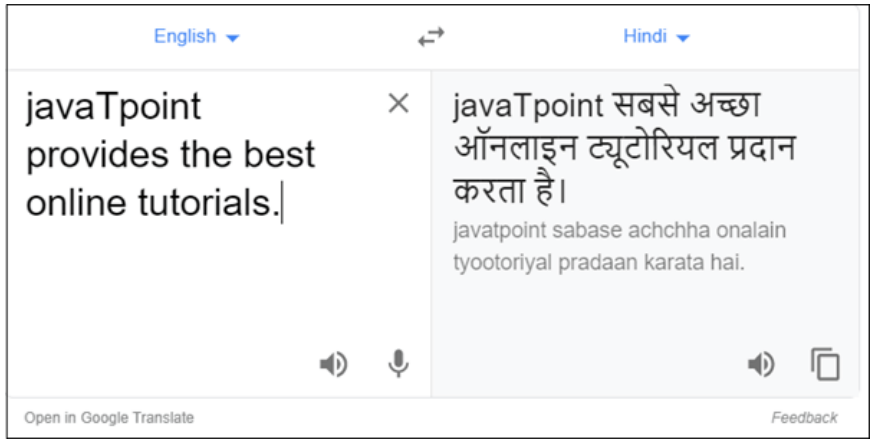
Question Answering



Spam Detection

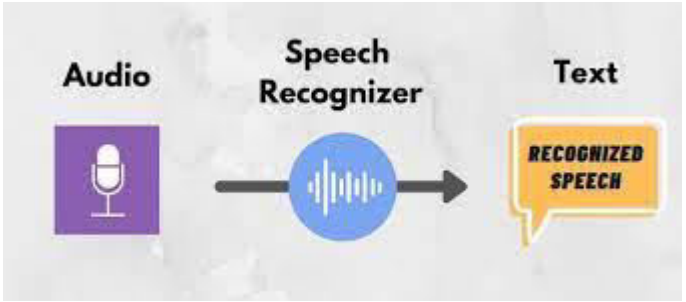


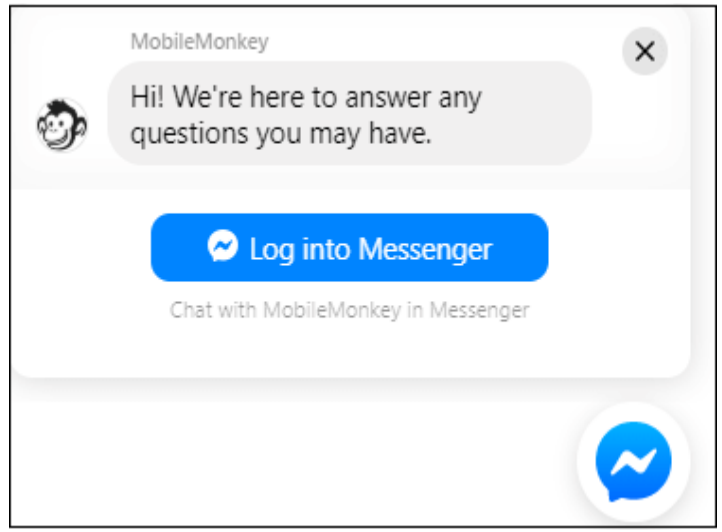
Sentiment Analysis



Machine Translation

Spelling correction

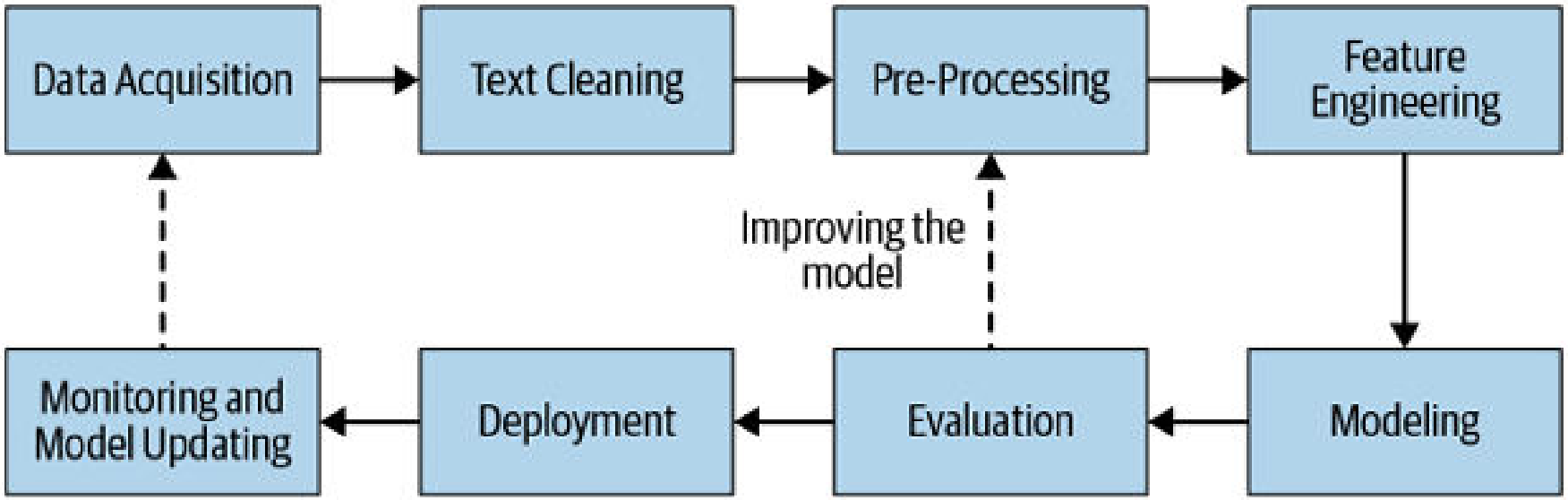




Chatbot

NLP Pipeline

Main components of a generic pipeline NLP system



NLP Pipeline

Data acquisition:

- Data acquisition involves obtaining raw textual data from various sources to create a dataset for NLP tasks.
- various sources like Documents, Emails, Social media posts, Transcribed speech, Application logs, Public Dataset, Web Scrapping, Image to Text, pdf to Text ,Data augmentation.

Text Cleaning :

- Sometimes our acquired data is not very clean.
- it may contain HTML tags, spelling mistakes, or special characters.
- So, use some techniques to clean our text data.

NLP Pipeline

Text Preprocessing:

- Preprocessing prepares the text for further analysis by cleaning and structuring it.

Steps in Preprocessing:

Tokenization: Splitting text into smaller units like words or sentences.

- Example: "I love NLP!" → ["I", "love", "NLP", "!"]

Lowercasing: Converting all text to lowercase for consistency.

- Example: "Natural Language Processing" → "natural language processing"

Stop-word Removal: Eliminating common, non-informative words.

- Example: Removing "the," "is," "and."

Lemmatization/Stemming: Reducing words to their root or base forms.

- Lemmatization: "running" → "run"
- Stemming: "flies" → "fli"

Punctuation and Special Character Removal: Removing unnecessary symbols or noise.

Part-of-Speech (POS) Tagging: POS tagging involves assigning a part of speech tag to each word in a text.

Example: "I love NLP." → [("I", Pronoun), ("love", Verb), ("NLP", Noun)]

NLP Pipeline

Feature Engineering/Feature Extraction:

- The goal of feature engineering is to represent/convert the text into a numeric vector that can be understood by the ML algorithms.

In this step, we use multiple techniques to convert text to numerical vectors.

1. One Hot Encoder
2. Bag Of Word(BOW)
3. n-grams
4. Tf-Idf
5. Word2vec

Modelling/Model Building

- In the modeling step, we try to make a model based on data.
- Here also we can use multiple approaches to build the model based on the problem statement.

Approaches to building model –

- Machine Learning Approach
- Deep Learning Approach

NLP Pipeline

Model Evaluation:

- In this evaluation, we use multiple metrics to check our model such as Accuracy, Recall, Confusion Metrics, etc.

Deployment

- In the deployment step, we have to deploy our model on the cloud/Server for the users and users can use this model.
- Deployment has three stages deployment, monitoring, and update.

Challenges in NLP

Ambiguity

- **Lexical Ambiguity**: Words can have multiple meanings depending on context (e.g., "bank" could mean a financial institution or a riverbank).
- **Syntactic Ambiguity**: Sentences can have multiple valid grammatical interpretations (e.g., "I saw the man with a telescope").

Misspellings

- Misspellings, can be more difficult for a machine to detect.
- You'll need to employ a natural language processing (NLP) technology that can identify and progress beyond typical misspellings of terms.

Multilingual and Cross-Language Challenges

- Developing systems that handle multiple languages or translate between them accurately is hard due to varying syntax, grammar, and idiomatic expressions.

Challenges in NLP

Data Quality and Bias

- Training data may contain biases, inaccuracies, or imbalances that result in biased or unfair NLP systems.
- Poor-quality datasets can lead to models misunderstanding or misrepresenting input.

Dynamic and Evolving Language

- Language constantly changes, with new words, slang, and phrases emerging, requiring models to stay updated.
- Handling code-switching (switching between languages or dialects in a conversation) remains challenging.

Domain Adaptation

- NLP models trained on general data may not perform well in specialized domains like medicine, law, or engineering, requiring fine-tuning with domain-specific data.

Low-Resource Languages

- Many languages lack large, high-quality datasets, making it challenging to build robust NLP systems for these languages.

Introduction to NLTK

- **NLTK (Natural Language Toolkit)** is a powerful and widely-used **Python library** for processing and analyzing human language data (text).
- It provides tools and methods for text processing, such as tokenization, stemming, lemmatization, parsing, classification, and more.
- To install
  - pip install nltk
- A variety of tasks can be performed using NLTK are
  - Tokenization
  - Lower case conversion
  - Stop Words removal
  - Stemming
  - Lemmatization
  - Parse tree or Syntax Tree generation
  - POS Tagging

Preprocessing techniques

- Preprocessing in **NLP** refers to the steps taken to clean and transform raw text data into a format suitable for further analysis.
- Since raw text often contains noise, inconsistencies, or irrelevant details, preprocessing ensures better performance of NLP tasks.
- Preprocessing techniques in NLP involve a series of steps to clean, transform, and prepare raw text for further analysis or modeling.
- These techniques ensure that text data is in a suitable format for machine learning algorithms or statistical models.

Text Wrangling

- Text wrangling, also known as text preprocessing or data cleaning, is the process of transforming raw, unstructured, and noisy text data into a clean and structured format that can be used effectively in NLP tasks.

Why is Text Wrangling Important?

1. **Raw Text is Noisy**: Raw data often contains irrelevant information such as HTML tags, emojis, misspellings, or special characters that can distort the results of NLP algorithms.
2. **Standardization**: It ensures that the text follows a consistent structure, making it easier to process and analyze.
3. **Improves Model Performance**: Properly cleaned and preprocessed data can significantly improve the accuracy and efficiency of machine learning models.

Text Wrangling/Text Cleaning Techniques

sentence splitter

- Spit the text into sentences.

Word Tokenization

split a sentence into words.

stop word removal

- Removal of most common words

rare word removal

- Removal of less important words.(low distribution)

Stemming

- words to their root forms

Lemmatization

- words to their root forms with preserving the meaning.

spell correction

- Correcting of spelling in given word

sentence splitter

- Sentence Splitting (or Sentence Segmentation) in NLP is the task of dividing a stream of text into individual sentences.
- In NLTK, you can use the built-in sent_tokenize() function to split text into sentences.

```
import nltk

from nltk.tokenize import sent_tokenize

text = "Hello world. NLP is amazing. Let's split this."
sentences = sent_tokenize(text)
print(sentences)
```

['Hello world.', 'NLP is amazing.', "Let's split this."]

Word Tokenization

- **Word Tokenization** is the process of splitting a text into individual words, which are the basic units for many NLP tasks, such as part-of-speech tagging, named entity recognition, and text classification.
- The word_tokenize() function from NLTK is a simple and effective way to split text into individual words, considering punctuation as separate tokens.

```
import nltk
from nltk.tokenize import word_tokenize

text = "Hello world! NLP is amazing. Let's tokenize this text, it's fun."

# Tokenize the text into words
tokens = word_tokenize(text)

# Print the tokens (words)
print("Tokens:", tokens)
```

```
Tokens: ['Hello', 'world', '!', 'NLP', 'is', 'amazing', '.', 'Let', "'s", 'tokenize', 'this', 'text', ',', 'it', "'s", 'fun', '.']
```

Stop word removal

- **Stop word removal** is a preprocessing step in NLP, where common words (like "the," "is," "in," etc.) are removed from a text because they do not contribute much meaningful information for many NLP tasks like text classification, sentiment analysis, and topic modeling.

Why Remove Stop Words?

- **Reduces Noise**: Stop words are frequent and usually carry little or no meaningful information, so removing them can help reduce the "noise" in the text.
- **Improves Efficiency**: Reducing the number of words in the dataset can speed up downstream processes like training machine learning models or performing text analysis.
- **Focus on Important Words**: It helps the model focus on words that carry more meaning and are more likely to affect the outcome of the analysis.

Common Stop Words:

- **English stop words**: "the", "is", "at", "which", "in", "on", "of", "for", "and", "or", "a", "an", etc.

Stop word removal

```
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
nltk.download('stopwords')

text = "This is a sample sentence, and it contains some stop words."
tokens = word_tokenize(text)
# Get the list of stop words in English
stop_words = set(stopwords.words('english'))
filtered_tokens=[]
# Remove stop words from the tokenized text
for word in tokens:
    if word.lower() not in stop_words:
        filtered_tokens.append(word)

print("Original Text:")
print(text)
print("\nFiltered Text (without stop words):")
print(filtered_tokens)
```

```
Original Text:
This is a sample sentence, and it contains some stop words.

Filtered Text (without stop words):
['sample', 'sentence', ',', 'contains', 'stop', 'words', '.']
```

Stop word removal

Custom Stop Word Removal:

- If you want to define your own list of stop words or extend the default list, you can create a custom stop word list.

```
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
nltk.download('stopwords')

# Custom stop words
custom_stop_words = set(['this', 'sample', 'contains'])
text = "This is a sample sentence, and it contains some stop words."
tokens = word_tokenize(text)
filtered_tokens=[]

for word in tokens:
    if word.lower() not in custom_stop_words:
        filtered_tokens.append(word)

filtered_text_custom = " ".join(filtered_tokens)

print("Filtered Text (with custom stop words):")
print(filtered_text_custom)
```

```
Filtered Text (with custom stop words):
is a sentence , and it some stop words .
```

Rare word removal

- **Rare word removal** is a technique in NLP where words that occur infrequently in a dataset (i.e., words with low frequency) are removed.

```
import nltk
from nltk.probability import FreqDist
from nltk.tokenize import word_tokenize

text = "This is a sample text with some rare words like xylophone, and other common words."
tokens = word_tokenize(text)
# Calculate word frequency distribution
fdist = FreqDist(tokens)
# Set a frequency threshold (e.g., remove words that appear less than 2 times)
threshold = 2

filtered_tokens = []
for word in tokens:
    if fdist[word] >= threshold:
        filtered_tokens.append(word)

print("Original Text:")
print(text)
print("\nFiltered Text (after removing rare words):")
print(filtered_tokens)
```

```
Original Text:
This is a sample text with some rare words like xylophone, and other common words.

Filtered Text (after removing rare words):
['words', 'words']
```

Rare word removal

```
import nltk
from nltk.probability import FreqDist
from nltk.tokenize import word_tokenize
doc = [
    "This is the first document.",
    "This document is the second document.",
    "And this is the third one.",
    "Is this the first document?"
]
# Tokenize all documents in the corpus
all_tokens = []
for line in doc:
    all_tokens.extend(word_tokenize(line))

# Calculate word frequencies across the entire corpus
fdist_all = FreqDist(all_tokens)
threshold = 2
filtered_tokens_doc = []
for word in all_tokens:
    if fdist_all[word] >= threshold:
        filtered_tokens_doc.append(word)

filtered_text_doc = " ".join(filtered_tokens_doc)
print("Filtered Text (after removing rare words in the corpus):")
print(filtered_text_doc)
```

Filtered Text (after removing rare words in the corpus):
This is the first document . This document is the document . this is the . this the first document

stemming

- **Stemming** in **NLP** is the process of reducing a word to its root form (or base form) by stripping off prefixes, suffixes, and other derivational affixes.

For example: **"running," "runs," "runner"** → **"run"**

- However, stemming does not always produce an actual dictionary word; it simply trims words based on predefined rules.

Why is Stemming Important?

1. **Reduces Redundancy**: Different forms of the same word are treated as one, helping improve efficiency.
2. **Simplifies Text Analysis**: Reduces the total number of unique words, leading to faster and easier analysis.

Common Stemming Algorithms

- Porter Stemmer
- Lancaster Stemmer
- Snowball Stemmer
- Regexp Stemmer

Porter Stemmer Example

```
from nltk.stem import PorterStemmer
from nltk.tokenize import word_tokenize

# Initialize the stemmer
porter = PorterStemmer()

words = ["running", "flies", "easily", "played"]

# Apply stemming
stemmed_words = []
for word in words:
    stemmed_words.append(porter.stem(word))

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

```
Original Words:  ['running', 'flies', 'easily', 'played']
Stemmed Words:  ['run', 'fli', 'easili', 'play']
```

Lancaster Stemmer Example

```
from nltk.stem import LancasterStemmer

# Initialize the stemmer
lancaster = LancasterStemmer()

# Example words
words = ["running", "flies", "easily", "played",]

stemmed_words = []
for word in words:
    stemmed_words.append(lancaster.stem(word))

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

```
Original Words:  ['running', 'flies', 'easily', 'played', 'running']
Stemmed Words:  ['run', 'fli', 'easy', 'play', 'run']
```

Snowball Stemmer Example

```
from nltk.stem import SnowballStemmer

# Initialize the Snowball Stemmer for English
snowball = SnowballStemmer("english")

words = ["running", "flies", "easily", "played", "running"]

stemmed_words = []
for word in words:
    stemmed_words.append(snowball.stem(word))
print("Original Words: ", words)
print("Stemmed Words: ", stemmed_words)
```

```
Original Words:  ['running', 'flies', 'easily', 'played', 'running']
Stemmed Words:  ['run', 'fli', 'easili', 'play', 'run']
```

RegexpStemmer Stemmer Example

```
from nltk.stem import RegexpStemmer

# Define a RegexpStemmer to remove common suffixes like 'ing', 'ly', 'ed', 's'
regstemmer = RegexpStemmer(r'(ing$|ly$|ed$|s$)')

words = ["running", "played", "happily", "studies", "cars", "faster"]

# Stem each word
stemmed_words = []
for word in words:
    stemmed_words.append(regstemmer.stem(word))

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

```
Original Words:  ['running', 'played', 'happily', 'studies', 'cars','faster']
Stemmed Words:  ['runn', 'play', 'happi', 'studie', 'car', 'faster']
```

lemmatization

- **Lemmatization** in **NLP** is the process of reducing a word to its **base form** (known as the *lemma*) by considering its **meaning** and **part of speech (POS)**.
- Unlike stemming, lemmatization produces valid dictionary words.
- It produces meaningful root words, unlike stemming, which can create non-existent words.
- For example:
  - **"studies" → "study“**

Why is Lemmatization Important?

1.Reduces Redundancy.

2.Improves Text Processing

lemmatization

- NLTK provides the **WordNet Lemmatizer** for this task, which uses the **WordNet** lexical database to find the lemma of a word.

```
from nltk.stem import WordNetLemmatizer
import nltk

# Download WordNet if not done yet
nltk.download('wordnet')

# Initialize the WordNet Lemmatizer
lemmatizer = WordNetLemmatizer()

words = ["plays", "flies", "studies", "better", "cars"]

lemmatized_words = []
stemmed_words = []
for word in words:
    lemmatized_words.append(lemmatizer.lemmatize(word))

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

```
Original Words:  ['plays', 'flies', 'studies', 'better', 'cars']
Lemmatized Words:  ['play', 'fly', 'study', 'better', 'car']
```

spell correction using Edit distance method

- Spelling corrections are important phase of text cleaning process, since misspelled words will leads a wrong prediction during machine learning process.
- Edit Distance measures the minimum number of edits (insertions, deletions, substitutions, or transpositions) required to transform one word into another.
- Words with small edit distances to known words in the dictionary can be suggested as corrections.

```
import nltk
from nltk.metrics import edit_distance
from nltk.corpus import words
# Download the words dataset
nltk.download("words")
# Get the list of valid words
valid_words = set(words.words())
input_words = ["exampl", "runnig", "crickt"]
corrected_words = []

# Process each word
for word in input_words:
    min_distance = float('inf')
    best_match = None

    # Compare with each valid word
    for valid_word in valid_words:
        distance = edit_distance(word, valid_word)
                if distance < min_distance:
            min_distance = distance
            best_match = valid_word

    # Append the best match to the corrected words
    corrected_words.append(best_match)
print("Original Words:", input_words)
print("Corrected Words:", corrected_words)
```

```
Original Words: ['exampl', 'runnig', 'crickt']
Corrected Words: ['example', 'running', 'cricket']
```

Punctuation, Special Characters, HTML Tags, stopword and numbers Removal and Lemmatization

```
import nltk
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
import re
import string
nltk.download('stopwords')
nltk.download('wordnet')

text = """  <html><body><h1>Welcome to NLP 101!</h1></body></html>
   This is an example text!, full of #special characters & numbers like 12345. """
# Step 1: Remove HTML tags
text = re.sub(r'<.*?>', '', text)
# Step 2: # Keeps only letters and spaces
text = re.sub(r'[^A-Za-z\s]', '', text)

words = word_tokenize(text)
lemmatizer = WordNetLemmatizer()
stop_words = set(stopwords.words('english'))
lemmatized_words = []
for word in words:
   if word not in stop_words:
         lemmatized_words.append(lemmatizer.lemmatize(word))

lemmatized_words=' '.join(lemmatized_words)
print(lemmatized_words)
```

Welcome NLP This example text full special character number like

Text Representation in NLP

- Text representation is a foundational aspect of NLP that involves converting raw text into numerical vectors that algorithms can process.
- Since machine learning models and algorithms work with numerical data, text must be transformed into a mathematical representation.

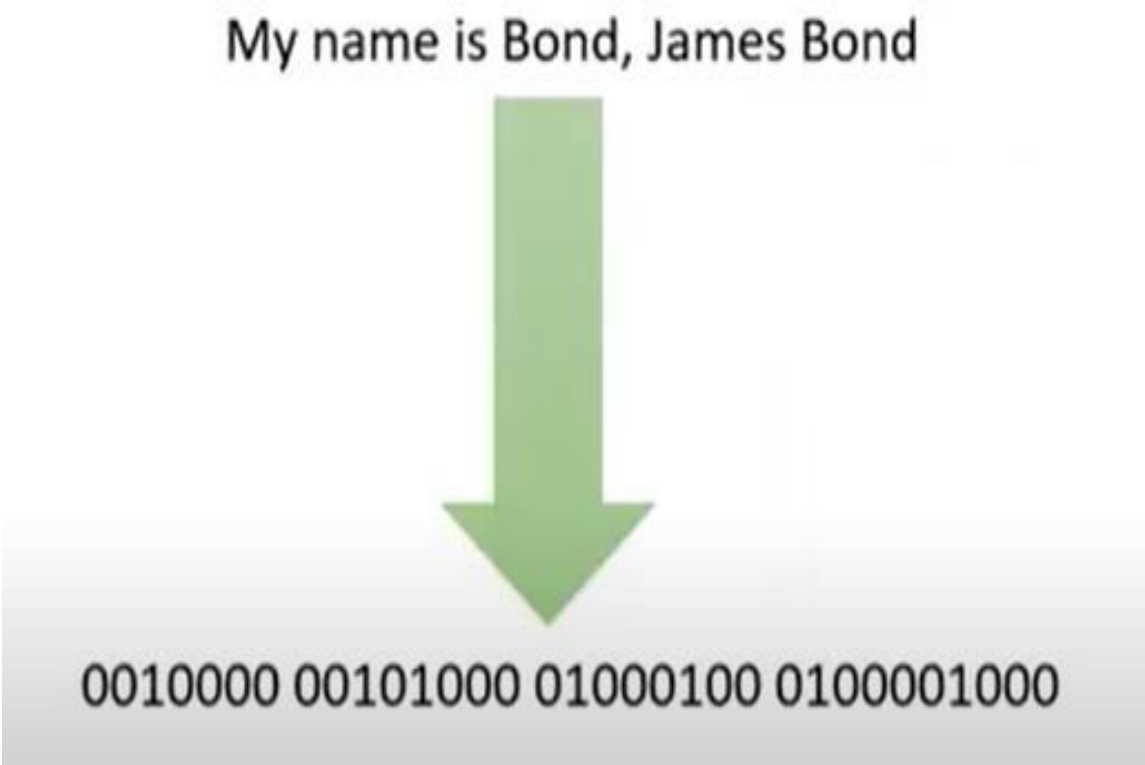
Common Terms Used While Representing Text in NLP

Corpus(C): All the text data or records of the dataset together are known as a corpus.

Vocabulary(V): This consists of all the unique words present in the corpus.

Document(D): One single text record of the dataset is a Document.

Word(W): The words present in the vocabulary.



Text Representation Techniques

1. One Hot Encoding
2. Bag of Words (BoW)
3. TF-IDF
4. word embeddings
    a. Word2vec
    b. GloVe
    c. FastText

One Hot Encoding

- One-Hot Encoding is a technique used to represent words as binary vectors, where each word in the vocabulary is mapped to a unique vector with a single "1" and the rest of the entries as "0".

Steps in One-Hot Encoding:

1.Tokenize the Text:

- The first step is to break down the text into words. For example:
  Text: "I love NLP"
  words: ['I', 'love', 'NLP']

2.Create a Vocabulary:

- Identify the unique words in the dataset (corpus). This list is called the vocabulary. For a larger dataset, you might apply additional preprocessing steps, such as removing stopwords or punctuation.
  Vocabulary: ['I', 'love', 'NLP']

One Hot Encoding

3. Assign Indices:

- Assign a unique integer index to each word in the vocabulary. This is essentially creating a mapping from each word to a position in a vector.
  - Example mapping:
    - 'I' → 0
    - 'love' → 1
    - 'NLP' → 2

4.Create the One-Hot Vectors:

- For each word in a given sentence, create a binary vector with the length equal to the size of the vocabulary.
- Set the position of the word (based on the index mapping) to 1, and all other positions to 0.
  - 'I' → Index 0 → One-hot vector: [1, 0, 0]
  - 'love' → Index 1 → One-hot vector: [0, 1, 0]
  - 'NLP' → Index 2 → One-hot vector: [0, 0, 1]

5. Represent the Entire Sentence or Document:

- After encoding each word, you can represent the entire sentence or document by concatenating the one-hot vectors of all words in the sentence. For example:
  - Sentence: "I love NLP"
  - One-hot vectors: [1, 0, 0], [0, 1, 0], [0, 0, 1]
  - Combined representation: [1, 0, 0, 0, 1, 0, 0, 0, 1]

corpus

D1:"I love NLP"

D2:"NLP is amazing"

D3:"I enjoy learning NLP"

vocabulary :

[I

love

NLP

is

amazing

enjoy

learning]

Assign Indices

I -> 0

love-> 1

NLP -> 2

is -> 3

amazing -> 4

enjoy -> 5

learning -> 6

Convert Words to One-Hot Vectors

I -> [1, 0, 0, 0, 0, 0, 0]

love-> [0, 1, 0, 0, 0, 0, 0]

NLP-> [0, 0, 1, 0, 0, 0, 0]

is -> [0, 0, 0, 1, 0, 0, 0]

amazing-> [0, 0, 0, 0, 1, 0, 0]

enjoy-> [0, 0, 0, 0, 0, 1, 0]

learning -> [0, 0, 0, 0, 0, 0, 1]

D1: [[1, 0, 0, 0, 0, 0, 0][0, 1, 0, 0, 0, 0, 0][0, 0, 1, 0, 0, 0, 0]]

D2:[[0, 0, 1, 0, 0, 0, 0][0, 0, 0, 1, 0, 0, 0][0, 0, 0, 0, 1, 0, 0]]

D3: [[1, 0, 0, 0, 0, 0, 0][0, 0, 0, 0, 0, 1, 0][0, 0, 0, 0, 0, 0, 1]

[0, 0, 1, 0, 0, 0, 0]]

```python
import nltk
from nltk.tokenize import word_tokenize

corpus = "I Love NLP"
# Tokenize the text into words
tokens = word_tokenize(corpus.lower())  # Tokenizing and converting to lowercase
# Remove punctuation from the tokens
tokens = [word for word in tokens if word.isalpha()]
# Create a vocabulary (list of unique words)
vocabulary = list(set(tokens))
print("Vocabulary:", vocabulary)

# Manually assigning index to each word in the vocabulary
word_to_index = {word: idx for idx, word in enumerate(vocabulary)}

# Create a one-hot encoding for each word in the tokenized text
one_hot_encoded = []

for word in vocabulary:
    encoding = [0] * len(vocabulary)  # Initialize a zero vector of length of vocabulary
    encoding[word_to_index[word]] = 1  # Set 1 at the index of the word
    one_hot_encoded.append(encoding)

# Print one-hot encoded vectors for the words
for word, encoding in zip(vocabulary, one_hot_encoded):
    print(f"Word: '{word}' -> One-hot encoding: {encoding}")
```

```
Vocabulary: ['i', 'nlp', 'love']
Word: 'i' -> One-hot encoding: [1, 0, 0]
Word: 'nlp' -> One-hot encoding: [0, 1, 0]
Word: 'love' -> One-hot encoding: [0, 0, 1]
```

disadvantages

High Dimensionality:

- One-hot encoding creates a large vector for each word, If the vocabulary is large (as in real-world text data), this leads to very high-dimensional vectors, which can be computationally expensive and inefficient.

Sparse Representation:

- Most of the elements in a one-hot encoded vector are zero, leading to sparse vectors. Sparse vectors are inefficient in terms of memory and computational resources, especially when dealing with large datasets.

Lack of Semantic Information:

- One-hot encoding does not capture any semantic relationships between words. For example, "cat" and "dog" would be represented as distinct, orthogonal vectors, even though they are semantically related. This limitation makes it difficult for the model to understand context or word similarity.

No Contextual Information:

- Each word is treated as an independent token without considering its context. For instance, the word "bank" could refer to a financial institution or the side of a river, but one-hot encoding doesn’t provide any indication of the word's contextual meaning.

Bag of Words (BoW)

- The Bag of Words (BoW) model is a representation technique used in NLP to convert text documents into numerical vectors.
- we can represent a sentence as a bag of words vector (a string of numbers).

How does Bag of Words work?

Tokenization: Break the text into individual words or tokens.

Vocabulary Creation: Create a vocabulary (a list of unique words) from all the text.

Vector Representation: For each document, represent it as a vector.

The vector's size will be equal to the number of unique words in the vocabulary, and each element in the vector represents the frequency of the corresponding word in the document.

Example-1:

Corpus:
- "I love programming."
- "I love NLP."
- "Programming is fun."

Tokenize the sentences
Sentence 1: ['I', 'love', 'programming’]
Sentence 2: ['I', 'love', 'NLP’]
Sentence 3: ['Programming', 'is', 'fun']

Vocabulary:
['I', 'love', 'programming', 'NLP', 'is', 'fun']

Represent each sentence as a vector

	I	love	programming	NLP	is	fun
Sentence 1	1	1	1	0	0	0
Sentence 2	1	1	0	1	0	0
Sentence 3	0	0	1	0	1	1

Sentence 1: "I love programming."
- Vector: [1, 1, 1, 0, 0, 0]

Sentence 2: "I love NLP."
- Vector: [1, 1, 0, 1, 0, 0]

Sentence 3: "Programming is fun."
- Vector: [0, 0, 1, 0, 1, 1]

Example-2:

Corpus:
“The cat sat on the mat.”
“The dog played in the yard.”

Vocabulary:
[“the”, “cat”, “sat”, “on”, “mat”, “dog”, “played”, “in”, “yard”]

Represent each sentence as a vector

	the	cat	sat	on	mat	dog	played	in	yard
D1	2	1	1	1	1	0	0	0	0
D2	2	0	0	0	0	1	1	1	1

Representing these sentences using BoW:
- Sentence 1: [2, 1, 1, 1, 1, 0, 0, 0, 0]
- Sentence 2: [2, 0, 0, 0, 0, 1, 1, 1, 1,]

Limitations

High-dimensionality

- As the number of words increases, the feature space grows, making models computationally expensive.

Sparsity

- The vector representation can be sparse, especially with a large vocabulary.

Ignoring Word Order

- BoW does not consider the word order in a sentence, which may lead to losing important context.

Ignores Context and Semantic Relationships

- It does not capture relationships between words or their meanings, leading to limitations in understanding context.

```
from sklearn.feature_extraction.text import CountVectorizer

documents = [
    "This movie is very scary and long",
    "This movie is not scary and is slow",
    "This movie is spooky and good"
]

# Initialize the CountVectorizer
vectorizer = CountVectorizer()

# Fit and transform the documents into the Bag of Words representation
bow_matrix = vectorizer.fit_transform(documents)

# Get the feature names (unique words in the corpus)
feature_names = vectorizer.get_feature_names_out()

# Convert the BoW matrix to an array for better visualization
bow_array = bow_matrix.toarray()

# Display the BoW matrix
print("Feature Names:", feature_names)
print("\nBag of Words Matrix:")
print(bow_array)
```

```
Feature Names: ['and' 'good' 'is' 'long' 'movie' 'not' 'scary' 'slow' 'spooky' 'this'
 'very']

Bag of Words Matrix:
[[1 0 1 1 1 0 1 0 0 1 1]
 [1 0 2 0 1 1 1 1 0 1 0]
 [1 1 1 0 1 0 0 0 1 1 0]]
```

TF-IDF

- In all the two approaches, all the words in the text are treated as equally important—there is no notion of some words in the document being more important than others.
- TF-IDF, or term frequency–inverse document frequency, addresses this issue.
- It aims to quantify the importance of a given word relative to other words in the document and in the corpus.
- This creates a numerical representation where higher scores indicate greater relevance.
- Mathematically, this is captured using two quantities: TF and IDF.
- TF (term frequency): measures how often a term or word occurs in a given document.
- TF of a term t in a document d is defined as: $$\mathrm{TF}(t,d) = \frac{(\text{Number of occurrences of term } t \text{ in document } d)}{(\text{Total number of terms in the document } d)}$$

- IDF (inverse document frequency): measures the importance of the term across a corpus.
- IDF of a term t is calculated as follows: $$\mathrm{IDF}(t) = \log_e \frac{(\text{Total number of documents in the corpus})}{(\text{Number of documents with term } t \text{ in them})}$$

- The TF-IDF score is a product of these two terms. Thus, TF-IDF score = TF * IDF.

Corpus:

Inflation has increased unemployment

The company has increased its sales

Fear increased his pulse

Step 1: Data Pre-processing

After lowercasing and removing stop words the sentences are transformed as below:

S.No.	Sentences
1.	inflation increased unemployment
2.	company increased sales
3.	fear increased pulse

Step 2: Calculating Term Frequency

$$\mathrm{TF}(t,d) = \frac{(\text{Number of occurrences of term } t \text{ in document } d)}{(\text{Total number of terms in the document } d)}$$

	inflation	company	increased	sales	fear	pulse	unemployment
inflation increased unemployment	1/3	0/3	1/3	0/3	0/3	0/3	1/3
company increased sales	0/3	1/3	1/3	1/3	0/3	0/3	0/3
fear increased pulse	0/3	0/3	1/3	0/3	1/3	1/3	0/3

Step 3: Calculating Inverse Document Frequency

Words	Inverse Document Frequency (IDF)
inflation	log(3/1) = 0.477
company	log(3/1) = 0.477
increased	log(3/3) = 0
sales	log(3/1) = 0.477
fear	log(3/1) = 0.477
pulse	log(3/1) = 0.477
unemployment	log(3/1)=0.477

$$\mathrm{IDF}(t) = \log_e \frac{(\text{Total number of documents in the corpus})}{(\text{Number of documents with term } t \text{ in them})}$$

Step 4: Calculating Product of Term Frequency & Inverse Document Frequency

TF-IDF score = TF * IDF.

	inflation	company	increased	sales	fear	pulse	unemployment
inflation increased unemployment	1/3*0.477	0/3*0.477	1/3*0	0/3*0.477	0/3*0.477	0/3*0.477	1/3*0.477
company increased sales	0/3*0.477	1/3*0.477	1/3*0	1/3*0.477	0/3*0.477	0/3*0.477	0/3*0.477
fear increased pulse	0/3*0.477	0/3*0.477	1/3*0	0/3*0.477	1/3*0.477	1/3*0.477	0/3*0.477

After simplifying ,we will get the final TF-IDF matrix as follows:

	inflation	company	increased	sales	fear	pulse	unemployment
inflation increased unemployment	0.159	0	0	0	0	0	0.159
company increased sales	0	0.159	0	0.159	0	0	0
fear increased pulse	0	0	0	0	0.159	0.159	0

TF-IDF Matrix:

[[0.159 0 0 0 0 0 0.159]
 [0 0.159 0 0.159 0 0 0]
 [0 0 0 0 0.159 0.159 0]]

TF-IDF

```
from sklearn.feature_extraction.text import TfidfVectorizer
documents = [
"The cat sat on the mat.",
"The dog sat on the mat.",
 "The mat is warm."
]

# Initialize TF-IDF Vectorizer
vectorizer = TfidfVectorizer(stop_words='english',max_features=3000)

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

# Feature names
features = vectorizer.get_feature_names_out()
print("TF-IDF Matrix:")
print(tfidf_matrix.toarray())
print("\nFeatures (Terms):")
print(features)
```

```
Features (Terms):
['cat' 'dog' 'mat' 'sat' 'warm']
TF-IDF Matrix:
[[0.72033345 0.         0.42544054 0.54783215 0.        ]
 [0.         0.72033345 0.42544054 0.54783215 0.        ]
 [0.         0.         0.50854232 0.         0.861037  ]]
```

TF-IDF

High Dimensionality

- For large corpora with many unique words, the feature vectors generated by TF-IDF can be very high-dimensional, leading to computational inefficiency and storage issues.

Ignores Context and Semantic Relationships

- TF-IDF treats words as independent entities and does not capture the relationships or meanings of words.

Does Not Consider Word Order

- TF-IDF is a "bag of words" model, meaning it ignores the order of words in the document.
- As a result, sentences with completely different meanings but similar word distributions can produce similar TF-IDF representations.

Word embeddings

- Word embeddings in NLP are a type of word representation where words or phrases are mapped to numerical vectors in a continuous vector space.
- These vectors capture semantic and syntactic meanings of words such that similar words (in meaning or context) are represented by similar vectors.

Key Concepts in Word Embeddings

Representation: Each word is represented as a fixed-size dense vector, where semantically similar words are closer together in the vector space.

Example: "king" and "queen" have vectors that are closer to each other than "king" and "apple."

Dimensionality: The size of the vector is predefined, typically ranging from 50 to 300 dimensions, depending on the complexity of the language and the dataset.

Learning: Embeddings are typically learned from large text corpora, leveraging the context in which words appear.

Word Embedding Techniques/models

Static Embeddings:

1. Word2Vec (Google):
2. GloVe (Stanford)
3. FastText(Facebook)

Contextual Embeddings:

1. ELMo (Embeddings from Language Models):
2. BERT (Bidirectional Encoder Representations from Transformers)
3. GPT (Generative Pre-trained Transformer)

Word2Vec

- Word2Vec is a popular algorithm/model in NLP used to create vector representations of words.
- These vectors are called **word embeddings**.
- It was introduced by researchers at Google in 2013
- Words that occur in similar contexts to have similar vector representations.
- Word2Vec uses a neural network-based approach that transforms words into dense vector representations.

Two main architectures used in the Word2Vec model to learn word embeddings:

CBOW (Continuous Bag of Words): Predicts the target/center word based on its context (surrounding words).

Skip-gram: Predicts the context words(surrounding words) based on the target/center word.

Word2Vec with CBOW

- CBOW predicts a target word based on its **context words**.

How CBOW Works:

For example, if the window size is **2**, the model will look at two words before and two words after the target word in the sentence.

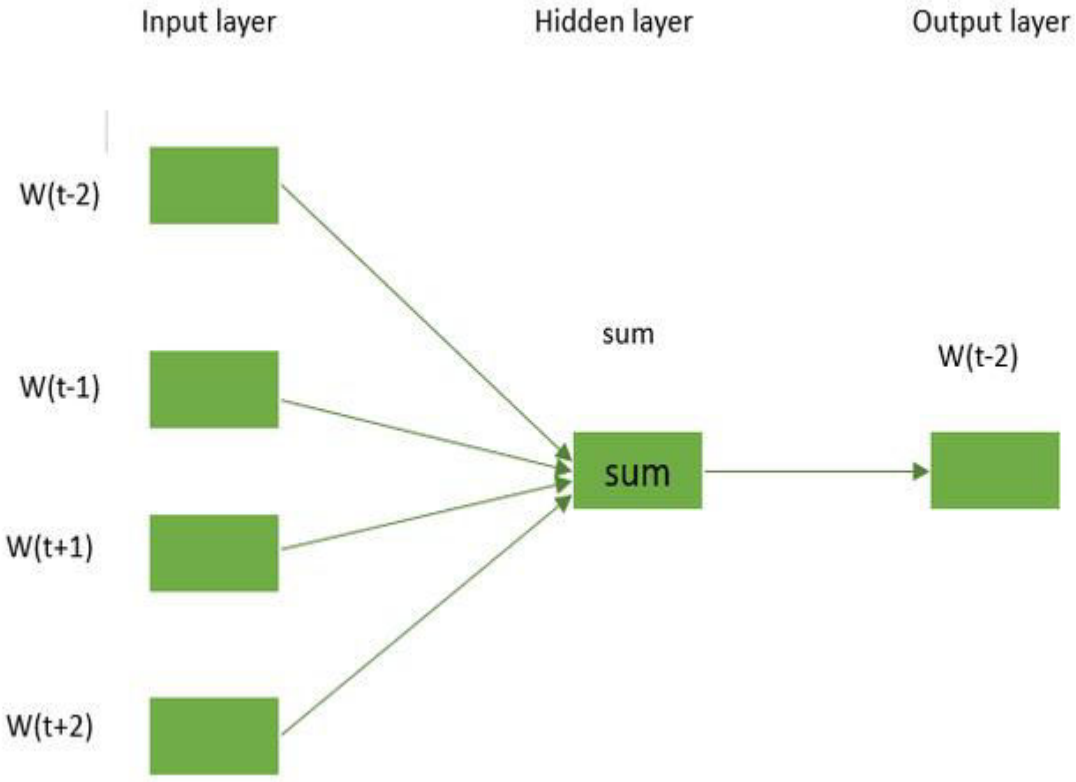
"The cat sat on the mat."

And the target word is "sat" with a window size of 2, the context would be:

•Context words: ["The", "cat", "on", "the"]

Output: Target Word

- •The model learns to predict the word sat using the context words.



1. Input Layer

The CBOW model takes a **context window** around a target word as input.

- **Input:** Surrounding words (context) of the target word.
- **Representation:** Words are one-hot encoded vectors (size = vocabulary size).

2. Embedding Layer

The one-hot encoded input is passed to an embedding layer, which is essentially a weight matrix W.

- **Weight Matrix:** W of size $V \times d$, where:
  - V: Vocabulary size
  - d: Embedding size (number of dimensions of word vectors)
- **Transformation:** Multiply each one-hot vector by W:

$$h_i = W^\top x_i$$

  - x_i: One-hot vector of the i-th context word
  - h_i: Embedding vector of the i-th context word

3. Hidden Layer

The embedding vectors of the context words are averaged to create a single vector.

- **Averaging:**

$$h = \frac{1}{n} \sum_{i=1}^{n} h_i$$

  - h: Hidden layer representation (context vector)

4. Output Layer

The context vector h is passed to the output layer to predict the target word.

- **Weight Matrix:** Another weight matrix W' of size $d \times V$ is used.
- **Transformation:** Compute scores for all words in the vocabulary:

$$z = W'^\top h$$

  - z: Scores for each word in the vocabulary.
- **Softmax Function:** Convert the scores into probabilities:

$$P(w_t|context) = \frac{\exp(z_t)}{\sum_{j=1}^{V} \exp(z_j)}$$

  - w_t: Target word
  - $P(w_t|context)$: Probability of the target word given the context.

5. Loss Function

The goal is to maximize the probability of the correct target word.

6. Training

The model is trained using backpropagation

- Update the weights to minimize the loss.

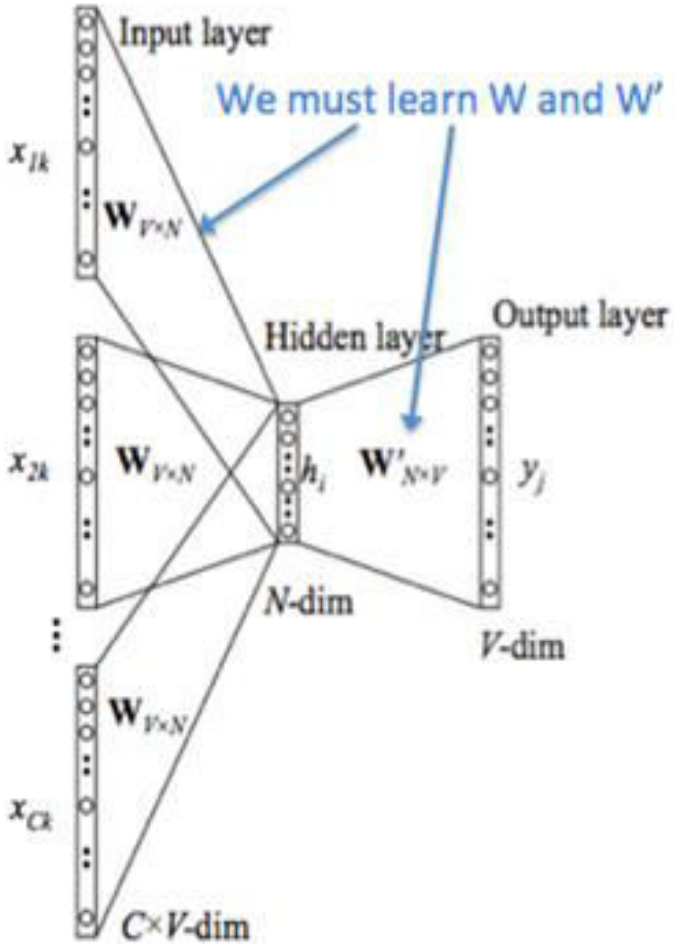
Once the training is done in the whole vocabulary, the weight matrix $\mathbf{W_{VxN}}$ contains the word embeddings, where each row corresponds to the vector representation of a word

We breakdown the way this model works in these steps:

$\mathcal{V}$ = W

$\mathcal{U}$ =W'

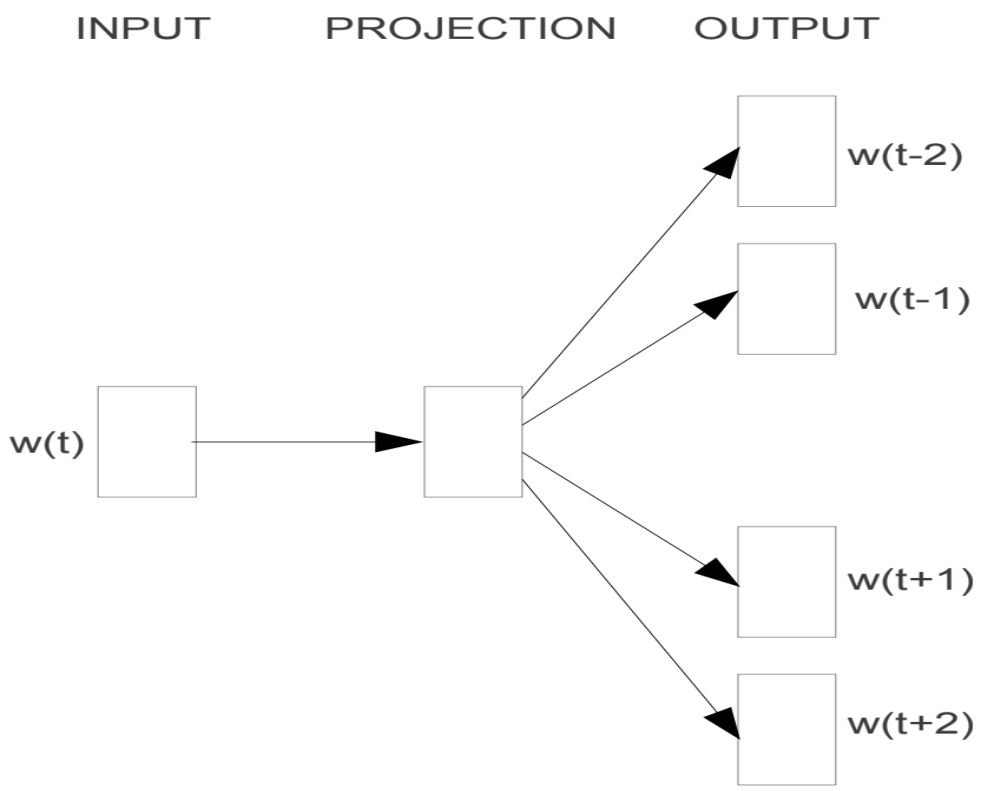
1. We generate our one hot word vectors for the input context of size $m : (x^{(c-m)}, \ldots, x^{(c-1)}, x^{(c+1)}, \ldots, x^{(c+m)} \in \mathbb{R}^{|V|})$.
2. We get our embedded word vectors for the context ($v_{c-m} = \mathcal{V}x^{(c-m)}, v_{c-m+1} = \mathcal{V}x^{(c-m+1)}, \ldots, v_{c+m} = \mathcal{V}x^{(c+m)} \in \mathbb{R}^n$)
3. Average these vectors to get $\hat{v} = \frac{v_{c-m} + v_{c-m+1} + \ldots + v_{c+m}}{2m} \in \mathbb{R}^n$
4. Generate a score vector $z = \mathcal{U}\hat{v} \in \mathbb{R}^{|V|}$. As the dot product of similar vectors is higher, it will push similar words close to each other in order to achieve a high score.
5. Turn the scores into probabilities $\hat{y} = \text{softmax}(z) \in \mathbb{R}^{|V|}$.
6. We desire our probabilities generated, $\hat{y} \in \mathbb{R}^{|V|}$, to match the true probabilities, $y \in \mathbb{R}^{|V|}$, which also happens to be the one hot vector of the actual word.



Once the training is done in the whole vocabulary, the weight matrix $\mathbf{W_{VxN}}$ contains the word embeddings, where each row corresponds to the vector representation of a word

Word2Vec with skip-gram

- The **Skip-gram** model in Word2Vec is used to predict the context words given a target word.
- It takes the target word and tries to predict the surrounding context words.



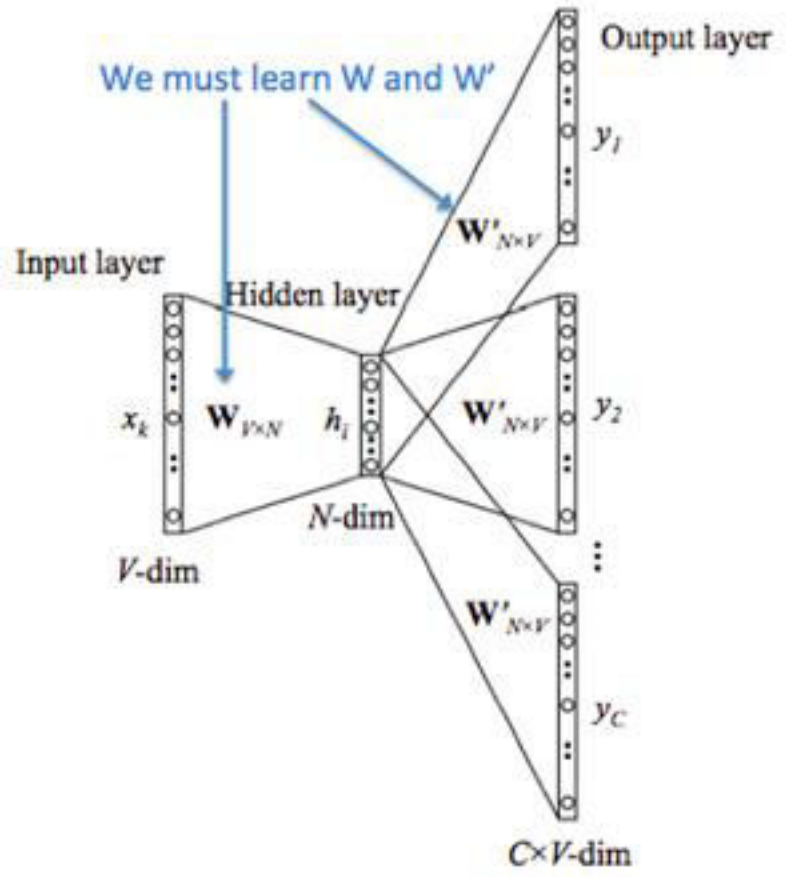
Skip-gram

$\mathcal{V}$ = W

$\mathcal{U}$ =W'

We breakdown the way this model works in these 6 steps:

1. We generate our one hot input vector $x \in \mathbb{R}^{|V|}$ of the center word.
2. We get our embedded word vector for the center word $v_c = \mathcal{V}x \in \mathbb{R}^n$
3. Generate a score vector $z = \mathcal{U}v_c$.
4. Turn the score vector into probabilities, $\hat{y} = \text{softmax}(z)$. Note that $\hat{y}_{c-m}, \ldots, \hat{y}_{c-1}, \hat{y}_{c+1}, \ldots, \hat{y}_{c+m}$ are the probabilities of observing each context word.
5. We desire our probability vector generated to match the true probabilities which is $y^{(c-m)}, \ldots, y^{(c-1)}, y^{(c+1)}, \ldots, y^{(c+m)}$, the one hot vectors of the actual output.



Once the training is done in the whole vocabulary, the weight matrix $\mathbf{W}_{VxN}$ contains the word embeddings, where each row corresponds to the vector representation of a word

Word2Vec Model

```
from gensim.models import Word2Vec

sentences = [
    ['the', 'quick', 'brown', 'fox'],
    ['jumps', 'over', 'the', 'lazy', 'dog']
]

# Train Word2Vec model
model = Word2Vec(sentences, vector_size=10, window=5, min_count=1, sg=0)  # sg=1 for Skip-gram, sg=0 for CBOW

#Get the embedding
model.wv['fox']
```

```
array([-0.08157917,  0.04495798, -0.04137076,  0.00824536,  0.08498619,
       -0.04462177,  0.045175  , -0.0678696 , -0.03548489,  0.09398508],
      dtype=float32)
```

```
# Find similar words
model.wv.most_similar('over')
```

```
[('quick', 0.2941223680973053),
 ('the', 0.1049435512314224),
 ('fox', 0.0926730707287884),
 ('lazy', -0.105510175228189),
 ('brown', -0.1138749942183494),
 ('dog', -0.2113374471664287),
 ('jumps', -0.3662713766098025)]
```

Word2Vec Model Limitations

Contextual Limitations

- Word2Vec represents words as fixed vectors, meaning each word has a single representation regardless of its meaning in different contexts.

OOV (Out-of-Vocabulary) Problem

- If a word does not appear in the training data (out-of-vocabulary), Word2Vec cannot generate its embedding.

Memory and Computation Complexity

- Training a Word2Vec model on large datasets with a large vocabulary may require considerable resources (memory and computation).

GloVe **Model**

- GloVe (Global Vectors for Word Representation) is an unsupervised learning algorithm for obtaining vector representations of words.
- It is widely used in NLP tasks to capture semantic meaning and relationships between words.
- GloVe was developed by researchers at Stanford University

How GloVe Works

Step 1: Build a Co-occurrence Matrix

•The co-occurrence matrix records how often words co-occur within a given context window in the corpus.

•Example Corpus: I like deep learning.
I like NLP.
I enjoy flying.

counts	I	like	enjoy	deep	learning	NLP	flying
I	0	2	1	0	0	0	0
like	2	0	0	1	0	1	0
enjoy	1	0	0	0	0	0	1
deep	0	1	0	0	1	0	0
learning	0	0	0	1	0	0	0
NLP	0	1	0	0	0	0	0
flying	0	0	1	0	0	0	0

GloVe **Model**

Step 2: Calculate Co-occurrence Ratios

•GloVe's key insight is that word meanings are encoded in the ratios of co-occurrence probabilities rather than raw frequencies.

•It uses these probabilities to express relationships between words mathematically.

Step 3: Train the Embeddings

- GloVe minimizes a weighted loss function to learn embeddings:

$$J = \sum_{i,j} f(X_{ij})(w_i^\top w_j + b_i + b_j - \log(X_{ij}))^2$$

  - X_{ij}: Co-occurrence count of words i and j.
  - w_i, w_j: Word vectors for i and j.
  - b_i, b_j: Bias terms for words i and j.
  - $f(X_{ij})$: Weighting function that reduces the impact of very frequent co-occurrences.

•The model uses an optimization technique, **stochastic gradient descent (SGD)** to minimize the loss function J.

•During training, the word vectors w_i and context vectors c_j are updated based on the gradients of the objective function with respect to each vector.

•This process is repeated until the model converges.

GloVe Model

Learned Word Vectors

- After training, the word vectors are stored in the matrix W (word vectors for each word in the vocabulary) and C (context vectors for each word in the vocabulary).

Final Embeddings

- GloVe uses the sum of the word vectors (word matrix W) and context vectors (context matrix C) to generate the final word embeddings for each word.

Pre-trained GloVe Model

```
import gensim
import gensim.downloader as api
my_vocab =
['apple','orange','shimla','banana','maruti','mumbai','china','india','husba
nd’]

glove_model = api.load('glove-wiki-gigaword-300’)

glove_model.most_similar('king',topn=5)
```

```
[('queen', 0.6336469054222107),
 ('prince', 0.6196622848510742),
 ('monarch', 0.5899620652198792),
 ('kingdom', 0.5791266560554504),
 ('throne', 0.5606487989425659)]
```

```
glove_model['king']
```

```
array([ 0.0033901, -0.34614 ,  0.28144 ,  0.48382 ,  0.59469 ,
        0.012965,  0.53982 ,  0.48233 ,  0.21463 , -1.0249  ,
       -0.34788 , -0.79001 , -0.15084 ,  0.61374 ,  0.042811,
        0.19323 ,  0.25462 ,  0.32528 ,  0.05698 ,  0.063253,
       -0.49439 ,  0.47337 , -0.16761 ,  0.045594,  0.30451 ,
       -0.35416 , -0.34583 , -0.20118 ,  0.25511 ,  0.091111,
```

Glove Model Limitations

Contextual Limitations

- Glove represents words as fixed vectors, meaning each word has a single representation regardless of its meaning in different contexts.

OOV (Out-of-Vocabulary) Problem

- If a word does not appear in the training data (out-of-vocabulary), Glove cannot generate its embedding.

Memory and Computation Complexity

- Training a Glove model on large datasets with a large vocabulary may require considerable resources (memory and computation).

FastText

- FastText is an open-source library for text representation and classification developed by Facebook's AI Research (FAIR) team.
- As Word2Vec and Glove models suffer with OOV (Out-of-Vocabulary) Problem. FastText overcomes this problem
- FastText breaks them down into smaller subword units, such as character n-grams.
  - For example, for the word "apple" with n-grams of size 3, the generated n-grams would be:
    - "<ap", "app", "ppl", "ple", "le>", "appl", "ppl>

- So, FastText will generate representations for such character n-grams and in turn, these will add up to form the embeddings of a complete word
- Suppose we train the FastText model on a corpus and the model is now familiar with a vocabulary.
- If we try to generate embeddings for a word that was absent in the vocabulary during training, the FastText model would still be able to generate embeddings for the unseen word as the n-grams information of the word would be present in the vocabulary and the model has already captured that information.

How FastText vectorizes text

Tokenization:

- FastText starts by splitting the input text into individual tokens (words).

Character n-grams:

- Each word is broken into character n-grams. For example, for the word "apple" with n-grams of size 3, the generated n-grams would be:
  - "<ap", "app", "ppl", "ple", "le>", "appl", "ppl>".
- The start (<) and end (>) characters are added to capture the boundaries of words.

Training:

- FastText uses the skip-gram or CBOW (Continuous Bag of Words) approach for training, similar to Word2Vec.

Word Representation:

- Each n-gram is assigned a vector (embedding) through the training process.
- A word's final vector is the sum (or average) of the embeddings of its n-grams.
- This ensures that words sharing similar characters or subword structures are close to each other in vector space.

Text Representation:

- Once the model is trained, any text can be represented as a vector by taking the word vectors of all words in the text and averaging them. This representation captures the overall meaning of the text.

Word2Vec Vs FastText

Aspect	Word2Vec	fastText
Handling of OOV Words	Struggles with OOV words	Handles OOV words efficiently using subword embeddings
Representation of Words	Generates embeddings solely based on words	Considers subword information for richer representations
Training Efficiency	Training speed is moderate	Exceptional speed and scalability, especially with large datasets
Use Cases	Well-suited for finding word relationships and semantic similarities	Preferred for handling OOV words, sentiment analysis, and understanding morphology
Word-Level vs. Subword-Level	Operates at the word level	Considers subword units for understanding word meanings and morphology

FastText Model

```
import nltk
import gensim
from gensim.models import FastText
from nltk.tokenize import word_tokenize


sentences =[
    "I love programming and machine learning",
   "natural language processing is fun"
       ]
tokenized_sentences = [word_tokenize(sentence)for sentence in sentences]
model=FastText(sentences=tokenized_sentences,vector_size=10,window=3,min_count=1, sg=1)
model.wv['fun’]
model.wv.most_similar('programming', topn=5)
model.wv['abc']
```

```
array([-0.00180062, -0.00018049,  0.0175008 ,  0.00461981,  0.00657197,
        0.00359357,  0.00191148, -0.00279698,  0.03218568, -0.00217876],
      dtype=float32)
```

```
[('love', 0.696203052997589),
 ('and', 0.5780276656150818),
 ('natural', 0.4464503824710846),
 ('is', 0.4181584417819977),
 ('I', 0.3748960494995117)]
```

```
array([-0.00109169,  0.00322612, -0.02925587,  0.01488084, -0.0150843 ,
       -0.00694802, -0.00084427,  0.02269463, -0.0037506 , -0.00589674],
      dtype=float32)
```