Chapter 3
Text Preprocessing and Vectorization

Er. Shiva Ram Dam

Assistant Professor

Gandaki University

Content:

3.1 Accessing text from the web and local disk

3.1 Accessing text from the web and local disk

- In NLP, the first step is often to **load text data** from various sources.
- Text may come from many sources:
  - Local computer files
  - Websites
  - Online APIs (Wikipedia API, Twitter API, etc.)
  - Databases
  - Structured datasets (CSV, JSON)
- Two common sources are:
  - **Local files** (disk)
  - **Web pages** (internet)
- We handle them differently depending on the format (plain text, HTML, PDF, etc.).

1. Accessing Text from Local Disk

- Local files may contain plain text, CSV, JSON, or other formats.
- Python provides easy ways to read text files.
- Always use "utf-8" encoding for text files to avoid errors with special characters.
- Example: Reading a plain text file

```
# Open and read a file in Python
file_path = "…/filepath/sample.txt"

with open(file_path, "r", encoding="utf-8") as f:
    text = f.read()

print(text[:200])  # print first 200 characters
```

- Line-by-Line Reading

```
# Open and read a file in Python
with open("/content/Natural_Language_Processing.txt",
"r", encoding="utf-8") as f:
    for line in f:
        print("LINE:", line.strip())
```

- **Reading CSV Files (useful for datasets like sentiment analysis)**
- Let the sample.csv has this content:

```
id,review
1,The movie was excellent!
2,It was a boring film.
```

```
import csv

with open("/content/sample.csv", "r", encoding="utf-8") as f:
    reader = csv.DictReader(f)
    for row in reader:
        print(row["review"])
```

Output:

```
The movie was excellent!
It was a boring film.
```

- Reading JSON Files (common for labeled NLP datasets)
- Let sample.json be a file with

```
{
  "text": "I love NLP.",
  "label": "positive"
}
```

```
import json

with open("/content/sample.json", "r", encoding="utf-8") as f:
    data = json.load(f)

print(data["text"], data["label"])
```

Output:

```
I love NLP. positive
```

2. Accessing Text From the Web

- To download pages or text from the web, we commonly use:
  - **requests** library (to access webpages/API)
  - **BeautifulSoup** (to parse HTML content)
  - **API requests** (Wikipedia API, HuggingFace datasets, etc.)

1. Using requests to fetch web text

- Example: Fetching a webpage

```
import requests

url = "https://www.gutenberg.org/files/1342/1342-0.txt"    # Pride and Prejudice
response = requests.get(url)

print("Status:", response.status_code)

text = response.text
print(text[:1000])  # print first 1000 characters
```

2. Extracting Text from HTML Using BeautifulSoup

- Webpages contain **HTML tags**, so we need to extract only the useful text.

```
import requests
from bs4 import BeautifulSoup

url = "https://en.wikipedia.org/wiki/Natural_language_processing"

headers = {"User-Agent": "Mozilla/5.0"}
response = requests.get(url, headers=headers)

soup = BeautifulSoup(response.text, "html.parser")
paragraphs = soup.find_all("p")

for p in paragraphs[:10]:
    print(p.get_text(), "\n")
```

Assignments

- **Create a Dataset**
  - Choose any online content source.
  - Extract **title + body**.
  - Store in CSV.

3.2 Regular Expressions for detecting word pattern

What are Regular Expressions?

- Regular expressions (Regex) are **special patterns used to match text**.
- They are widely used in **Natural Language Processing (NLP)** to:
  - Find words
  - Detect patterns
  - Extract information from text
- Python provides the **re** module to work with regex.
- **Practical Applications in NLP**
  - **Tokenization**: splitting sentences into words
  - **Pattern-based filtering**: extracting emails, URLs, hashtags
  - **Text cleaning**: remove unwanted characters or digits
  - **Data preprocessing**: normalize numbers, remove punctuation

Basic Regex Syntax

Symbol	Meaning	Example
.	Any character	a.c *matches* abc, a1c
*	0 or more repetitions	ab* *matches* a, ab, abb, abbb
+	1 or more repetitions	ab+ *matches* ab, abb, but not a
?	0 or 1 repetition	colou?r *matches* color or colour
[]	Character class	[aeiou] *matches any vowel*
^	Start of string	^Hello *matches text starting with* Hello
$	End of string	world$ *matches text ending with* world
\d	Digit [0-9]	\d+ *matches one or more digits*
\w	Word character [a-zA-Z0-9_]	\w+ *matches words*
\s	Whitespace	\s+ *matches spaces, tabs, newlines*
\b	Word boundary	\bword\b matches "word" but not "sword

Python Regex Functions

Function	Description
re.match(pattern, string)	Check if pattern matches at the **beginning**
re.search(pattern, string)	Search pattern **anywhere** in the string
re.findall(pattern, string)	Returns **all occurrences** of pattern in string
re.sub(pattern, repl, string)	Replace matched pattern with repl
re.split(pattern, string)	Split string by pattern

Format of a Regex Pattern

- A regex pattern is a **string of characters** that defines a **search rule**.

  pattern = r"\b[A-Z]\w*\b"

  - **r** before the string → raw string (tells Python not to treat \ as escape)
  - **\b** → word boundary
  - **[A-Z]** → any uppercase letter
  - **\w*** → zero or more word characters (letters, digits, underscore)
  - **\b** → end of word boundary
- This pattern matches all **words starting with a capital letter**

Detecting Word Patterns

- Use **\b** for word boundaries
- **\w** matches letters, numbers, underscore
- **\d** matches digits
- **+** means 1 or more repetitions
- ***** means 0 or more repetitions
- **?** makes previous token optional

a) Detect all email addresses

```
import re
text = "Contact us at support@example.com or hr@company.org"

emails = re.findall(r'\b[\w.-]+@[\w.-]+\.\w+\b', text)
print("Emails found:", emails)
```

- Output:
  - Emails found: ['support@example.com', 'hr@company.org']

b) Detect hashtags

```
tweet = "Learning #NLP with #Python is fun! #AI #ML"

hashtags = re.findall(r'#\w+', tweet)
print("Hashtags:", hashtags)
```

- Output:
  - Hashtags: ['#NLP', '#Python', '#AI', '#ML']

c) Detect dates (dd/mm/yyyy or dd-mm-yyyy)

```
text = "Today is 05/12/2025, and tomorrow will be 06-12-2025."

dates = re.findall(r'\b\d{2}[-/]\d{2}[-/]\d{4}\b', text)
print("Dates found:", dates)
```

- Output:
  - Dates found: ['05/12/2025', '06-12-2025']

d) Detect words ending with “ing”

```
text = "I am learning, playing, and coding in Python."

ing_words = re.findall(r'\b\w+ing\b', text)
print("Words ending with 'ing':", ing_words)
```

- Output:
  - Words ending with 'ing': ['learning', 'playing', 'coding']

Exercise:

- Extract all phone numbers from a text like +977-9801234567, 9801234567
- Find all capitalized words in a paragraph
- Replace all URLs in text with <URL>
- Detect all words containing numbers, e.g., B2B, C3PO

3.3 Tokenization: Sentence, word

3.3 Tokenization: Sentence, word

- **Tokenization** is the process of **splitting text into smaller units called tokens**.
- Tokens can be:
  - **Words** → "I love Python" → ["I", "love", "Python"]
  - **Sentences** → "I love Python. NLP is fun." → ["I love Python.", "NLP is fun."]
- It is the **first step in most NLP tasks** like text analysis, sentiment analysis, and language modeling.
- **Word tokenization**: splits text into words, punctuation can be removed using regex.
- **Sentence tokenization**: splits text into sentences using punctuation.
- Python tools:
  - split(), re.findall(), nltk.word_tokenize(), nltk.sent_tokenize().
- Tokenization transforms raw text into structured units that can be processed, analyzed, and modeled. Without it, NLP tasks would be extremely difficult and inefficient.

Types of Tokenization

Type	Description	Example
Word Tokenization	Split text into words	"I love NLP" → ["I", "love", "NLP"]
Sentence Tokenization	Split text into sentences	"Hello. How are you?" → ["Hello.", "How are you?"]

Word Boundaries in Tokenization

- Word boundaries are represented by **\b in regex**.
- \bword\b ensures you **match the word "word" exactly**, not substrings like "sword" or "words".
- This is very useful in **pattern-based tokenization**

```
import re

text = "I like word, sword, and words."
# Match the exact word "word"
matches = re.findall(r'\bword\b', text)
print(matches)
```

- **Output:**
  - ['word']
- Only "word" is matched, "sword" and "words" are ignored.

a) Word Tokenization

```
# Using simple split
text = "I love NLP and Python."
words = text.split()  # Splits by whitespace
print(words)

# Using regex to remove punctuation
words = re.findall(r'\b\w+\b', text)
print(words)
```

- Output:
  - ['I', 'love', 'NLP', 'and', 'Python.']

b) Sentence Tokenization

```
#using Regex
text = "Hello there! How are you? I am fine."
sentences = re.split(r'(?<=[.!?]) +', text)
print(sentences)
```

- Output:
  - ['Hello there!', 'How are you?', 'I am fine.']

Using NLTK

```
import nltk

nltk.download('punkt_tab')

from nltk.tokenize import sent_tokenize, word_tokenize

text = "Hello there! How are you? I am fine."

words = word_tokenize(text)
print("Words:", words)

sentences = sent_tokenize(text)
print("Sentences:", sentences)
```

- Output:
  - Words: ['Hello', 'there', '!', 'How', 'are', 'you', '?', 'I', 'am', 'fine', '.']
  - Sentences: ['Hello there!', 'How are you?', 'I am fine.']

Significance of Tokenization

- It converts raw text into manageable units called **tokens** (words, sentences, or subwords).
- Provides foundation for NLP tasks: Tokenization **transforms unstructured text into structured data** that algorithms can process.
- Enables Word-Level Analysis: Word tokens allow you to:
  - Count word frequency (**Bag-of-Words**)
  - Create **TF-IDF vectors**
  - Perform **word embeddings** for deep learning (Word2Vec, GloVe, BERT)
- Tokenization is essential for **cleaning and normalizing text**, such as:
  - Removing punctuation
  - Converting to lowercase
  - Removing stopwords
  - Lemmatization or stemming
- Reduces Complexity
- Enables Pattern Matching and Information Extraction

Significance	Explanation
Foundation for NLP tasks	Tokenized text is required for almost all NLP applications
Word-level analysis	Frequency counts, embeddings, and vectorization
Sentence-level analysis	Summarization, sentiment, QA systems
Text preprocessing	Cleaning, stopword removal, normalization
Pattern matching	Extract emails, hashtags, numbers using regex
Reduces complexity	Converts raw text into manageable units
Supports advanced models	RNNs, LSTMs, Transformers need tokenized input

Exercise:

- You have a CSV file news.csv with 5 rows and 3 columns:

id	title	body
1	"AI in Healthcare"	"AI is transforming healthcare. It helps diagnose diseases faster."
2	"SpaceX Launch"	"SpaceX launched another rocket today. The mission was successful."
3	"Climate Change"	"Global warming is real. Scientists warn of rising sea levels."
4	"Python Tips"	"Python is popular. Learning Python improves job prospects."
5	"Sports Update"	"The football match ended in a draw. Fans are excited."

- **Tasks:**
- **Read the CSV** into a DataFrame.
- **Build a function tokenize_text** that:
  - Accepts a **text string**.
  - Returns **two lists**:
    - sentences → sentence-tokenized
    - words → word-tokenized (punctuation removed)
- Apply the function to **both title and body columns**.
- Store the results in a **new DataFrame** with columns:
  - | id | title_tokens | body_tokens | title_sentences | body_sentences |

Solution:

- Create a csv file as : **news.csv**

id	title	body
1	"AI in Healthcare"	"AI is transforming healthcare. It helps diagnose diseases faster."
2	"SpaceX Launch"	"SpaceX launched another rocket today. The mission was successful."
3	"Climate Change"	"Global warming is real. Scientists warn of rising sea levels."
4	"Python Tips"	"Python is popular. Learning Python improves job prospects."
5	"Sports Update"	"The football match ended in a draw. Fans are excited."

```
import pandas as pd
import re
from nltk.tokenize import sent_tokenize
import nltk

# Step 1: Load the CSV file
#df = pd.read_csv('news.csv')
print("Original DataFrame:\n", df.head(), "\n")

# Step 2: Build tokenization function
def tokenize_text(text):
    #Input: text string Output: tuple (word_tokens, sentence_tokens)
    # Sentence tokenization
    sentences = sent_tokenize(text)

    # Word tokenization (remove punctuation)
    words = re.findall(r'\b\w+\b', text)

    return words, sentences
```

```
# Step 3: Apply the function to 'title' and 'body' columns
df['title_tokens'], df['title_sentences'] = zip(*df['title'].apply(tokenize_text))
df['body_tokens'], df['body_sentences'] = zip(*df['body'].apply(tokenize_text))

# Step 4: Create a new DataFrame with tokenized results
tokenized_df = df[['id', 'title_tokens', 'body_tokens', 'title_sentences',
'body_sentences']]
print("Tokenized DataFrame:\n", tokenized_df.head(), "\n")

# Step 5: Save the tokenized DataFrame to a new CSV file
tokenized_df.to_csv('news2.csv', index=False)
print("Tokenized data saved to 'news2.csv'")
```

3.4 Stop words and removal

What are Stop Words?

- **Stop words** are common words in a language that **do not carry significant meaning** for NLP tasks.
- Examples in English:

  a, an, the, in, on, is, are, of, and, to
- They are usually **removed during preprocessing** to reduce noise and improve performance.
- **Example**:
  - "The cat is sitting on the mat.“
- After removing stop words:
  - "cat sitting mat“
- Only the **meaningful words** remain.

Why Remove Stop Words?

- **Reduce Dimensionality**: Stop words increase feature space without adding useful information.
- **Improve Model Accuracy**: They do not contribute to text meaning and may confuse models.
- **Faster Computation**: Less words → smaller data → faster algorithms.
- **Focus on Meaningful Words**: Helps extract **keywords**, **named entities**, and relevant features.

Common Sources of Stop Words

- **NLTK** library in Python has a built-in stop word list for multiple languages.
- You can also create a **custom stop word list** depending on your dataset.
- Notes:
  - Stop words removal is **optional**, depending on the task:
    - Useful for **text classification, topic modeling**
    - Not always needed for **language generation tasks** (e.g., chatbots, summarization)
  - Always **lowercase words** before comparing with stop words list.

Examples: Stopword Removal

a) Using NLTK Stop Words
b) Using Custom Stop Words
c) Removing Stop Words from a CSV Column

a) Using NLTK Stop Words

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

# Sample text
text = "This is an example showing the removal of stop words from a sentence."
# Word tokenization
words = word_tokenize(text)
# Load English stop words
stop_words = set(stopwords.words('english'))

# Remove stop words
filtered_words = [w for w in words if w.lower() not in stop_words]

print("Original words:", words)
print("After stop word removal:", filtered_words)
```

- Output:
  - Original words: ['This', 'is', 'an', 'example', 'showing', 'the', 'removal', 'of', 'stop', 'words', 'from', 'a', 'sentence', '.']
  - After stop word removal: ['example', 'showing', 'removal', 'stop', 'words', 'sentence', '.']

b) Using Custom Stop Words

```
custom_stopwords = ['example', 'showing', 'of']

filtered_custom = [w for w in words if w.lower() not in custom_stopwords]
print("After custom stop word removal:", filtered_custom)
```

- Output:
  - After custom stop word removal: ['This', 'is', 'an', 'the', 'removal', 'stop', 'words', 'from', 'a', 'sentence', '.']

c) Removing Stop Words from a CSV Column

- Assume a CSV news.csv with a **column** body:

```
import pandas as pd

# Load CSV
df = pd.read_csv('news.csv')

# Remove stop words from 'body'
df['body_clean'] = df['body'].apply(lambda x: ' '.join([w for w in
word_tokenize(x) if w.lower() not in stop_words]))

print(df[['body', 'body_clean']])
```

Exercises for Students

- Remove stop words from the title and body columns of news.csv.
- Compare **text length before and after stop word removal**.
- Create a **custom stop word list** for your dataset and remove them.
- Apply stop word removal and **tokenize words**, then store in a new DataFrame.

3.5 Word Stemming

What is Word Stemming?

- **Word stemming** is the process of **reducing a word to its root or base form**.
- The root form is called the **stem**, which may not always be a valid word in the language.
- Stemming is used to **group words with the same meaning** together in NLP tasks.
- Example:

Original Words	Stemmed Word
playing, played, plays	play
running, runner, runs	run
happiness, happy	happi

Why is Stemming Important?

- **Reduces dimensionality**: Groups similar words → smaller feature set.
- **Improves NLP tasks**: Useful in search engines, text classification, information retrieval.
- **Helps in generalization**: Words like “run”, “running”, “runs” are treated as the **same concept**.

- **Example:**
  - Query: "running shoes"
  - Documents: "run shoes", "runner shoes"
  - After stemming, all words → "run shoes" → better match.

Applications of Stemming

- **Search engines**: match different forms of a word
- **Text classification**: reduce feature space
- **Information retrieval**: query expansion
- **Text mining**: preprocessing for NLP pipelines

Common Stemming Algorithms

Stemmer	Description
Porter Stemmer	Most popular, simple and fast, sometimes over-stems
Lancaster Stemmer	More aggressive than Porter, may cut words too much
Snowball Stemmer	Improved version of Porter Stemmer, supports multiple languages

Python Examples Using NLTK

a) Using Porter Stemmer
b) Using Lancaster Stemmer
c) Stemming Words in a Sentence

a) Using Porter Stemmer

```
import nltk
from nltk.stem import PorterStemmer

stemmer = PorterStemmer()

words = ["playing", "played", "plays", "runner", "running", "happiness"]
stemmed_words = [stemmer.stem(word) for word in words]

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

- Output:
  - Original words: ['playing', 'played', 'plays', 'runner', 'running', 'happiness']
  - Stemmed words: ['play', 'play', 'play', 'runner', 'run', 'happi']

b) Using Lancaster Stemmer

```
from nltk.stem import LancasterStemmer

lancaster = LancasterStemmer()
words = ["playing", "played", "plays", "runner", "running", "happiness"]
stemmed_lanc = [lancaster.stem(word) for word in words]

print("Original words:", words)
print("Lancaster stemmed words:", stemmed_lanc)
```

- Output:
  - Original words: ['playing', 'played', 'plays', 'runner', 'running', 'happiness']
  - Lancaster stemmed words: ['play', 'play', 'play', 'run', 'run', 'happy']

c) Stemming Words in a Sentence

```
from nltk.tokenize import word_tokenize

sentence = "The runners are running and playing in the playground"
tokens = word_tokenize(sentence)

stemmer = PorterStemmer()

stemmed_tokens = [stemmer.stem(token) for token in tokens]
print("Original tokens:", tokens)
print("Stemmed tokens:", stemmed_tokens)
```

- Output:
  - Original tokens: ['The', 'runners', 'are', 'running', 'and', 'playing', 'in', 'the', 'playground']
  - Stemmed tokens: ['the', 'runner', 'are', 'run', 'and', 'play', 'in', 'the', 'playground']

Exercises for Students

- Stem all words in the following list:
  ["cats", "caring", "happily", "connection", "connected"]
- Compare the output of **Porter Stemmer** vs **Lancaster Stemmer**.
- Apply stemming to the body column of news.csv and store in a new column body_stemmed.
- Use stemming to preprocess text before counting word frequency.

3.6 Lemmatizer

What is Lemmatization?

- **Lemmatization** is the process of **reducing a word to its dictionary or base form**, called a **lemma**.
- It often uses **part-of-speech (POS) tagging** to select the correct lemma. Requires **POS tagging** to get correct base form.
- Unlike stemming, lemmatization **produces meaningful words**. Lemmatization is **more accurate** than stemming.
- Often used **after stop word removal** and **tokenization**.
- Slower than stemming but better for **meaning-preserving NLP tasks**.
- Example:

Original Word	Lemma
running	run
better	good
cats	cat
leaves	leaf

Why Lemmatization is Important

- Produces **real words**, making results more interpretable.
- Reduces **dimensionality** of the text.
- Improves **accuracy of NLP tasks** like search, classification, and information retrieval.
- Essential for **POS-aware NLP applications**

Lemmatization vs Stemming

Feature	Stemming	Lemmatization
Output	Root/stem (may not be valid word)	Dictionary form (valid word)
Accuracy	Less accurate	More accurate
Example	“happiness” → “happi”	“happiness” → “happiness”
Uses	Quick preprocessing	Applications needing precise meaning
POS aware	No	Yes

Python Examples Using NLTK

- a) Lemmatization with WordNetLemmatizer
- b) Lemmatization with POS Tagging
- c) Applying Lemmatization to a CSV Column

a) Lemmatization with WordNetLemmatizer

```
import nltk
from nltk.stem import WordNetLemmatizer
from nltk.tokenize import word_tokenize
# Download required resources
nltk.download('wordnet')
lemmatizer = WordNetLemmatizer()

words = ["running", "cats", "better", "leaves", "geese"]
lemmas = [lemmatizer.lemmatize(word) for word in words]
print("Words:", words)
print("Lemmas:", lemmas)
```

- Output:
  - [nltk_data] Downloading package wordnet to /root/nltk_data...
  - Words: ['running', 'cats', 'better', 'leaves', 'geese']
  - Lemmas: ['running', 'cat', 'better', 'leaf', 'goose']
- Notice: Without POS tagging, "running" stays "running".

b) Lemmatization with POS Tagging

```
import nltk
nltk.download('punkt_tab')
nltk.download('averaged_perceptron_tagger_
eng')
from nltk.corpus import wordnet
from nltk import pos_tag

# Function to map NLTK POS tags to WordNet
POS
def get_wordnet_pos(treebank_tag):
    if treebank_tag.startswith('J'):
        return wordnet.ADJ
    elif treebank_tag.startswith('V'):
        return wordnet.VERB
    elif treebank_tag.startswith('N'):
        return wordnet.NOUN
    elif treebank_tag.startswith('R'):
        return wordnet.ADV
    else:
        return wordnet.NOUN  # Default to
noun
```

```
sentence = "The cats are running faster
than the geese."
tokens = word_tokenize(sentence)
pos_tags = pos_tag(tokens)

lemmas = [lemmatizer.lemmatize(word,
get_wordnet_pos(pos)) for word, pos in
pos_tags]
print("Original:", tokens)
print("Lemmatized:", lemmas)
```

c) Applying Lemmatization to a CSV Column

```
import pandas as pd

# Load news.csv
df = pd.read_csv('news.csv')

# Lemmatize 'body' column
def lemmatize_text(text):
    tokens = word_tokenize(text)
    pos_tags = pos_tag(tokens)
    lemmas = [lemmatizer.lemmatize(word, get_wordnet_pos(pos)) for word, pos
in pos_tags]
    return ' '.join(lemmas)

df['body_lemmatized'] = df['body'].apply(lemmatize_text)
print(df[['body', 'body_lemmatized']].head())
```

- Output:
- [nltk_data] Downloading package punkt_tab to /root/nltk_data...
- [nltk_data] Package punkt_tab is already up-to-date!
- [nltk_data] Downloading package averaged_perceptron_tagger_eng to
- [nltk_data] /root/nltk_data... [nltk_data] Unzipping taggers/averaged_perceptron_tagger_eng.zip.
- Original: ['The', 'cats', 'are', 'running', 'faster', 'than', 'the', 'geese', '.']
- Lemmatized: ['The', 'cat', 'be', 'run', 'faster', 'than', 'the', 'geese', '.']

Exercise for students:

- Lemmatize the following words: ["running", "better", "flies", "leaves", "studies"]
- Apply lemmatization to title and body columns of news.csv.
- Compare the results of **stemming vs lemmatization** on the same text.
- Create a **function preprocess_text(text)** that performs:
  - Lowercasing
  - Tokenization
  - Stop word removal
  - Lemmatization

3.7 Text vectorization: One-hot encoding, BOW, TFIDF, Word Embedding

Text Vectorization

- Text vectorization is the process of **converting text into numerical form** so that machine learning algorithms can process it.
- Why Do We Need Text Vectorization?
  - Machines cannot understand text directly; they need numbers.
  - Vectorization converts text into **numeric vectors**.
  - Used in NLP tasks like **classification, clustering, sentiment analysis, topic modeling**, etc.

One-Hot Encoding

- Each unique word is represented as a **vector** where:
  - The position for that word = **1**
  - All other positions = **0**
- **Advantages**
  - Very simple
  - Easy to implement
- **Disadvantages**
  - **High dimensionality** for large vocabularies
  - **No meaning captured** (dog and cat are unrelated)
- **Example:**
  - Vocabulary = {“cat”, “dog”, “rat”}

Word	Vector
Cat	[1, 0, 0]
dog	[0, 1, 0]
rat	[0, 0, 1]

Python example

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

words = np.array(["cat", "dog", "rat"]).reshape(-1, 1)
enc = OneHotEncoder(sparse_output=False)
print(enc.fit_transform(words))
```

- Output:

```
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

Bag of Words (BOW)

- BOW represents a document by **counting how many times each word appears**.
- **Advantages**
  - Simple and effective for many tasks
  - Good for document classification
- **Disadvantages**
  - High dimensionality
  - Counts common words heavily, even if unimportant

- **Example**
  - Documents:
    - D1: “dog barks”
    - D2: “cat meows”
  - Vocabulary = {barks, cat, dog, meows}

Document	barks	cat	dog	meows
D1	1	0	1	0
D2	0	1	0	1

Example:

```
from sklearn.feature_extraction.text import
CountVectorizer

docs = ["dog barks", "cat meows"]
vectorizer = CountVectorizer()
X = vectorizer.fit_transform(docs)

print(vectorizer.get_feature_names_out())
print(X.toarray())
```

- Output:

```
['barks' 'cat' 'dog' 'meows']
[[1 0 1 0]
 [0 1 0 1]]
```

Numerical

- Represent below documents with BOW vectorization.
  - **D1:** "data mining techniques"
  - **D2:** "data science and data mining"
  - **D3:** "big data analytics"

- **Step 1: Build the Vocabulary**
  - O: data, 1: mining, 2: techniques, 3: science, 4: and, 5: big, 6: analytics
  - So the BOW vectors will have **7 dimensions**.
- **Step 2: Count the frequency of each word in each document**

Doc	data	mining	techniques	science	and	big	analytics
D1	1	1	1	0	0	0	0
D2	**2**	**1**	0	**1**	**1**	0	0
D3	1	0	0	0	0	1	1

  - Matrix notation:

$$\begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 2 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

TF-IDF (Term Frequency – Inverse Document Frequency)

- TF-IDF shows **how important a word is** in a document **relative** to a whole collection.
  - **TF:** Number of occurrences of a word
  - **IDF:** log(total documents / documents containing the word)
  - **TFIDF= TF x IDF**
- **Why TF-IDF is better than BOW?**
  - Reduces the weight of common words like “the”, “is”
  - Highlights important words
- **Example:**
  - If “**data**” occurs in many documents, its weight is low.
  - If “**entropy**” occurs in only one document, weight is high.
- **Advantages**
  - Reduces noise from common words
  - Better for information retrieval and ranking
- **Disadvantages**
  - Still bag-of-words based
  - Does NOT understand meaning or context

```
from sklearn.feature_extraction.text import TfidfVectorizer

docs = ["data mining techniques", "mining is fun"]
tfidf = TfidfVectorizer()
X = tfidf.fit_transform(docs)

print(tfidf.get_feature_names_out())
print(X.toarray())
```

- Output:
  - ['data‘ 'fun' 'is‘ 'mining' 'techniques']
  - [[0.6316672 0. 0. 0.44943642 0.6316672]
  - [0. 0.6316672 0.6316672 0.44943642 0.]]

Numerical:

- Represent below documents with TFIDF vectorization.
  - **D1:** "data mining techniques"
  - **D2:** "data science and data mining"
  - **D3:** "big data analytics"

- Step 1: Vocabulary Construction:
  - [and, analytics, big, data, mining, science, techniques]
  - Vocabulary size = **7**

- Step 2: TF computation
  - **Document lengths:**
    - D1: 3 words
    - D2: 5 words
    - D3: 3 words
- TF for D1 (“data mining techniques”)

Word	Count	TF
and	0	0
analytics	0	0
big	0	0
data	1	1/3 = 0.333
mining	1	0.333
science	0	0
techniques	1	0.333

- TF for D2 (“data science and data mining”)

Word	Count	TF
and	1	1/5 = 0.20
analytics	0	0
big	0	0
data	2	2/5 = 0.40
mining	1	0.20
science	1	0.20
techniques	0	0

- TF for D3 (“big data analytics”)

Word	Count	TF
and	0	0
analytics	1	1/3 = 0.333
big	1	0.333
data	1	0.333
mining	0	0
science	0	0
techniques	0	0

- **STEP 3 — IDF Calculation**
  - Total documents: **N = 3**
  - Calculate document frequency **df**:

Word	df
and	1 (D2)
analytics	1 (D3)
big	1 (D3)
data	3 (D1, D2, D3)
mining	2 (D1, D2)
science	1 (D2)
techniques	1 (D1)

- Compute:

$$IDF - \log\left(\frac{N}{df}\right)$$

Word	**Idf**
and	log(3/1) = **1.0986**
analytics	log(3/1) = **1.0986**
big	log(3/1) = **1.0986**
data	log(3/3) = **0**
mining	log(3/2) = **0.4055**
science	log(3/1) = **1.0986**
techniques	log(3/1) = **1.0986**

- **STEP 4 — TF–IDF for Each Document**
- TFIDF=TF×IDF

Word	IDF	Document D1		Document D2		Documents D3	
		TF	TFIDF= TF x IDF	TF	TFIDF= TF x IDF	TF	TFIDF= TF x IDF
and	**1.0986**	0	**0**	0.20	**0.2197**	0	**0**
analytics	**1.0986**	0	**0**	0	**0**	0.333	**0.366**
big	**1.0986**	0	**0**	0	**0**	0.333	**0.366**
data	**0**	0.333	**0**	0.40	**0**	0.333	**0**
mining	**0.4055**	0.333	**0.135**	0.20	**0.0811**	0	**0**
science	**1.0986**	0	**0**	0.20	**0.2197**	0	**0**
techniques	**1.0986**	0.333	**0.366**	0	**0**	0	**0**

- Final TF–IDF Vectors (Complete Table)

Word →	and	analytics	big	data	mining	science	techniques
D1	0	0	0	0	**0.135**	0	**0.366**
D2	**0.2197**	0	0	0	**0.0811**	**0.2197**	0
D3	0	**0.366**	**0.366**	0	0	0	0

Word Embedding (Dense Vector Representation)

- Word Embeddings convert words into **dense, low-dimensional vectors** that capture **meaning and relationships**.
- Embeddings are **learned** using deep learning models.
- **Popular Word Embedding Models**
  - **Word2Vec (Google)**
  - **GloVe (Stanford)**
  - **FastText (Facebook)**
- **Advantages**
  - Captures **semantic meaning**
  - Words with similar meaning have **similar vectors**
  - Low dimensional representation (50–300 dimensions)
- **Disadvantages**
  - Requires heavy training or pretrained models
  - Not easily interpretable

- Word embeddings are a **way to represent words as vectors of numbers**, so that **similar words have similar representations**. Unlike Bag-of-Words or TF-IDF, embeddings **capture semantic meaning**, not just word frequency.
- **1. Motivation**
  - Traditional methods:
    - **Bag-of-Words**: counts words, ignores word order and meaning.
    - **TF-IDF**: gives importance to words based on frequency and rarity, but still **no semantic meaning**.
- Example: "king" - "man" + "woman" ≈ "queen"
  - BOW/TF-IDF cannot do this.
  - Word embeddings can, because similar words have vectors close in a high-dimensional space.

- **Word2Vec**: Learns embeddings from your corpus. OOV words are not handled.
- **GloVe**: Uses co-occurrence matrix. Pre-trained vectors are more common in practice.
- **FastText**: Similar to Word2Vec but **can generate embeddings for unseen words** using subwords.

How Word Embeddings Work

- Word embeddings map each word w to a vector $v_w \in R^d$, where d is the embedding dimension (usually 50–300).
- Words with **similar meaning** → vectors close in space.
- Captures **contextual relationships**.
- **Examples:**
- Vectors for:

Word	Vector (example)
king	[0.21, 0.45, -0.12, …]
queen	[0.18, 0.51, -0.10, …]
man	[0.12, 0.40, -0.15, …]
woman	[0.10, 0.44, -0.13, …]

Methods to Generate Word Embeddings

- **Predictive Models**
  - Learn embeddings by predicting a word from context.
  - Examples:
    - **Word2Vec**: CBOW (predict word from context) / Skip-gram (predict context from word)
    - **GloVe**: counts-based matrix factorization embedding
- **Contextual Embeddings**
  - Consider context in sentences (same word has different vectors in different contexts).
  - Examples:
    - **BERT**, **GPT**, **ELMo**

Properties of Word Embeddings

- **Semantic similarity**
  - “dog” and “cat” → close vectors
  - “king” and “queen” → close vectors
- **Arithmetic relationships**
  - king – man + woman ≈ queen
  - Paris – France + Italy ≈ Rome
- **Dense vector**
  - Unlike BOW (sparse, high-dimensional), embeddings are **dense** and low-dimensional (50–300 dims).

Applications

- Text classification (spam detection, sentiment analysis)
- Named entity recognition (NER)
- Machine translation
- Question-answering and chatbots
- Recommendation systems

Python Example:

```
from gensim.models import Word2Vec

# Sample corpus
sentences = [
    ["data", "mining", "techniques"],
    ["machine", "learning", "models"],
    ["big", "data", "analytics"]
]

# Train Word2Vec model
model = Word2Vec(sentences, vector_size=50, window=2, min_count=1, workers=4)

# Vector for a word
vector_data = model.wv['data']
print("Vector for 'data':", vector_data)

# Find similar words
similar = model.wv.most_similar('data', topn=3)
print("Words similar to 'data':", similar)
```

Output:

- Vector for 'data': [-1.0724545e-03 4.7286271e-04 1.0206699e-02 1.8018546e-02 -1.8605899e-02 -1.4233618e-02 1.2917745e-02 1.7945977e-02 -1.0030856e-02 -7.5267432e-03 1.4761009e-02 -3.0669428e-03 -9.0732267e-03 1.3108104e-02 -9.7203208e-03 -3.6320353e-03 5.7531595e-03 1.9837476e-03 -1.6570430e-02 -1.8897636e-02 1.4623532e-02 1.0140524e-02 1.3515387e-02 1.5257311e-03 1.2701781e-02 -6.8107317e-03 -1.8928028e-03 1.1537147e-02 -1.5043275e-02 -7.8722071e-03 -1.5023164e-02 -1.8600845e-03 1.9076237e-02 -1.4638334e-02 -4.6675373e-03 -3.8754821e-03 1.6154874e-02 -1.1861792e-02 9.0324880e-05 -9.5074680e-03 -1.9207101e-02 1.0014586e-02 -1.7519170e-02 -8.7836506e-03 -7.0199967e-05 -5.9236289e-04 -1.5322480e-02 1.9229487e-02 9.9641159e-03 1.8466286e-02]
- Words similar to 'data': [('mining', 0.1320439279079437 3), ('big', 0.12670072913169 86), ('analytics', 0.04237301647663 1165)]

1. Word2Vec (Google, 2013)

- Predictive model: learns embeddings by predicting words from their context.
- Introduced by Google in 2013.
- Generates **dense vector representations** of words.
- Each word is mapped to a fixed-dimensional vector.
- **Key Features**

- Key Features:
- **Two architectures**:
  - **CBOW (Continuous Bag of Words)**: Predicts the target word from surrounding context words.
  - context→target
  - Example: context = ["data", "techniques"] → predict target = "mining"
- **Skip-gram**: Predicts surrounding words from the target word.
  - target→context
  - Example: target = "mining" → predict context = ["data", "techniques"]
- Produces **dense embeddings** (50–300 dimensions).
- Captures semantic relationships:
  - king - man + woman ≈ queen
  - Paris - France + Italy ≈ Rome
- **Pros**
  - Efficient, fast to train on large corpus.
  - Captures semantic similarity and vector arithmetic.
- **Cons**
  - Does not handle **out-of-vocabulary (OOV)** words.
  - Only single-word embeddings (no subword information).

2. GloVe (Stanford, 2014)

- **Overview**
  - **Global Vectors for Word Representation**.
  - Developed by Stanford.
  - Counts-based model: learns embeddings using **word co-occurrence matrix**.
  - Captures global statistical information about words.

How it works

- Build a **co-occurrence matrix** X, where X[i, j] = number of times word i appears in context of word j.
- Factorize the matrix to generate word vectors W.
- Optimizes the following loss:

$$J = \sum_{i,j=1}^{V} f(X_{ij}) \left(w_i^T \tilde{w}_j + b_i + \tilde{b}_j - \log X_{ij} \right)^2$$

- W_i , $\widetilde{W}_j$ = word vectors
- b_i , $\widetilde{b}_j$ = bias terms
- $f(X_{ij})$ = weighting function to reduce influence of very frequent words

- **Pros**
  - Captures **global statistical information** (better than Word2Vec for rare words sometimes).
  - Works well on large corpora.
  - Pre-trained embeddings widely available.
- **Cons**
  - Still cannot handle OOV words directly.
  - Requires building huge co-occurrence matrix for large vocabularies.

Python Example:

```
!pip install gensim --quiet

import gensim.downloader as api

# Load pre-trained GloVe vectors (50-dimensional)
glove_model = api.load("glove-wiki-gigaword-50")

# Vector for 'data'
vector_data = glove_model['data']
print("GloVe vector for 'data':")
print(vector_data)

# Find similar words
similar_words = glove_model.most_similar('data', topn=5)
print("\nWords similar to 'data':")
print(similar_words)
```

```
[==================================================] 100.0% 66.0/66.0MB downloaded
GloVe vector for 'data':
[ 5.3101e-01 -5.5869e-01  1.7674e+00  4.4824e-01  2.2341e-01 -3.4559e-01
 -7.7679e-01 -9.6117e-01  1.1669e+00  7.4279e-02  8.1470e-01 -5.9428e-02
  6.4599e-02  1.5176e-03  9.9179e-02  3.6602e-01 -9.8724e-01 -8.3913e-01
  1.5917e-01 -7.7603e-01  7.3474e-01 -6.4861e-01  4.6174e-01  8.8162e-03
  5.1738e-01 -6.5976e-01 -7.4010e-01 -1.3928e-01  8.1094e-02  2.0657e-01
  3.5652e+00 -8.2264e-01  5.7360e-01 -1.7268e+00  6.2356e-03  6.7672e-02
 -2.3411e-01  3.5163e-02  2.6507e-01 -2.9966e-01  7.4323e-01 -4.5027e-01
  1.9406e-01  4.8611e-01 -4.3075e-01 -2.5210e-01  1.2774e+00  1.5815e+00
  6.5838e-01 -2.0978e-01]

Words similar to 'data':
[('information', 0.8329989314079285), ('tracking', 0.8124602437019348), ('database',
```

3. FastText (Facebook, 2016)

- **Overview**
  - Developed by Facebook AI.
  - Extension of Word2Vec.
  - Uses **subword information** (character n-grams) to create embeddings.
  - Handles **out-of-vocabulary words** naturally.
- **How it works**
  - A word is represented as the sum of its **n-gram vectors**.
    Example: "where" → <wh, whe, her, ere, re>
  - The final embedding = sum of all subword embeddings.
  - Trained similar to **Skip-gram** or **CBOW**.

- **Pros**
  - Handles **OOV words** → good for morphologically rich languages.
  - Captures **semantic and syntactic similarity**.
  - Pre-trained embeddings available in multiple languages.
- **Cons**
  - Slightly slower to train than Word2Vec.
  - Slightly larger model size due to n-grams.

Python Program:

```
# ================================
# Install required packages
# ================================
!pip install gensim --quiet

# ================================
# Import libraries
# ================================
from gensim.models import Word2Vec, FastText
import gensim.downloader as api
import numpy as np
```

```
# ================================
# 1. Prepare Corpus
# ================================
sentences = [
    ["data", "mining", "techniques"],
    ["data", "science", "and", "data", "mining"],
    ["big", "data", "analytics"]
]

# List of all unique words
vocab = sorted(set(word for sent in sentences for word in sent))
print("Vocabulary:", vocab)

# ================================
# 2. Word2Vec
# ================================
w2v_model = Word2Vec(sentences, vector_size=50, window=2, min_count=1,
workers=4)
```

```
print("\n--- Word2Vec Vectors ---")
for word in vocab:
    print(f"{word}: {w2v_model.wv[word][:5]} ...")  # show first 5
dimensions

# ================================
# 3. FastText
# ================================
ft_model = FastText(sentences, vector_size=50, window=2, min_count=1,
workers=4)

print("\n--- FastText Vectors ---")
for word in vocab:
    print(f"{word}: {ft_model.wv[word][:5]} ...")  # first 5 dimensions

# Example of OOV word with FastText
oov_word = 'analytics123'
print(f"\nVector for OOV word '{oov_word}':\n", ft_model.wv[oov_word][:5],
"...")
```

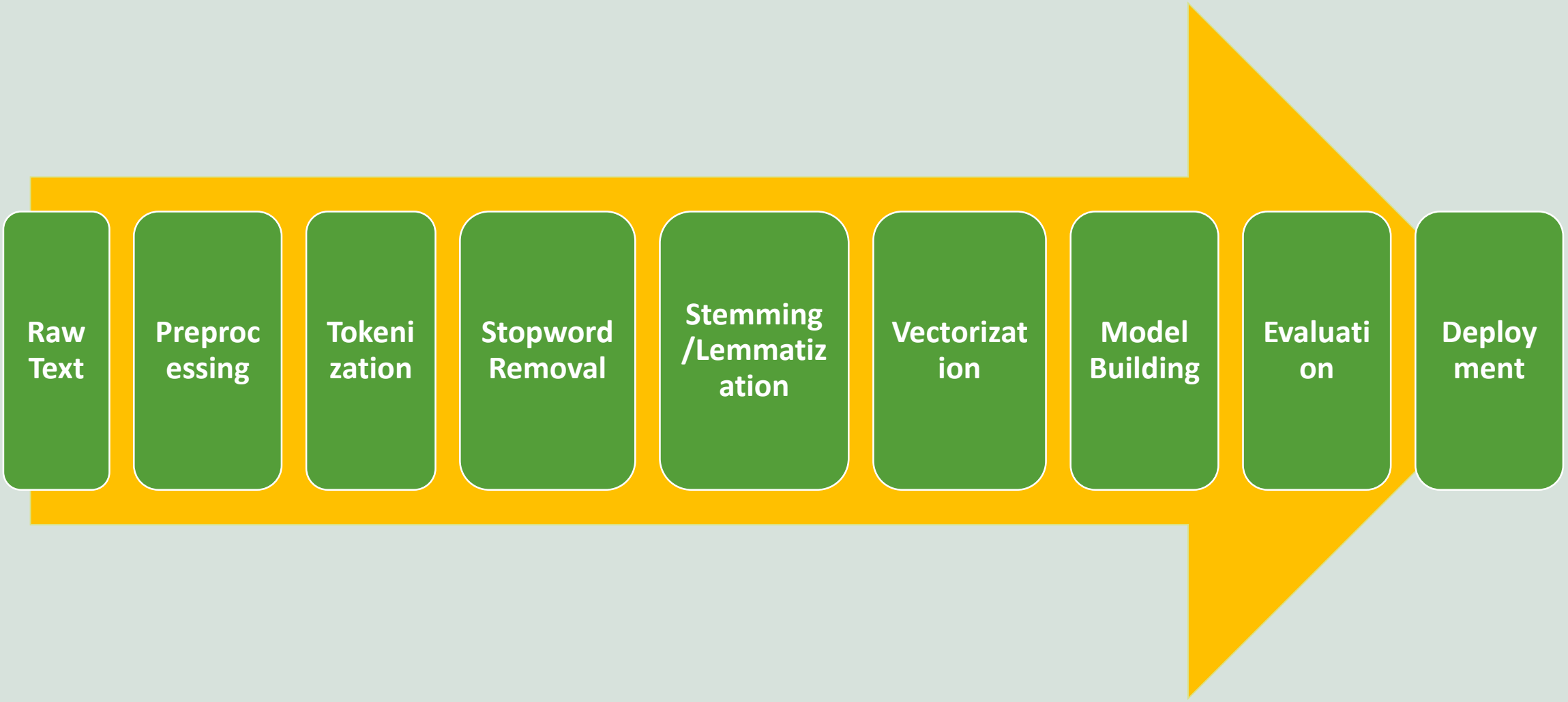
```
# ================================
# 4. GloVe (Pre-trained via gensim)
# ================================
glove_model = api.load("glove-wiki-gigaword-50")  # 50-dimensional vectors

print("\n--- GloVe Vectors ---")
for word in vocab:
    if word in glove_model.key_to_index:
        print(f"{word}: {glove_model[word][:5]} ...")
    else:
        print(f"{word}: Not in GloVe vocabulary")

# ================================
# 5. Find Similar Words (Example)
# ================================
word = 'data'
print("\nMost similar words to 'data':")
print("Word2Vec:", w2v_model.wv.most_similar(word, topn=3))
print("FastText:", ft_model.wv.most_similar(word, topn=3))
print("GloVe:", glove_model.most_similar(word, topn=3))
```

3.8 Fundamental algorithms for NLP, Model evaluation

NLP pipeline



Model Building

- Model building is the stage where we take the **processed, vectorized text** and train a **machine learning or deep learning model** to perform an NLP task.
- It is the **core step** where learning happens.

Purpose of Model Building

- Once text has been:
  - cleaned (preprocessed),
  - tokenized,
  - converted into numerical vectors (BOW, TF-IDF, embeddings),
  - …a model can **learn patterns** in the data to make predictions.
- **Examples of what the model can learn:**
  - Detect whether a review is *positive* or *negative*
  - Classify emails into *spam* or *not spam*
  - Predict the next word in a sentence
  - Identify named entities like *persons*, *locations*, *dates*

Types of NLP Models

- **A. Classical Machine Learning Models**
  - Used when the dataset is smaller and vectorization is simple (TF-IDF, BOW).
  - Widely used models:
    - **Naive Bayes** (fast, good for text classification)
    - **Logistic Regression**
    - **Support Vector Machine (SVM)**
    - **Decision Trees / Random Forest**
    - **K-Nearest Neighbors (KNN)**
  - **Example use-case:**
    TF-IDF + Logistic Regression → spam detection.

- **B. Deep Learning Models**
- These handle word sequences and context better.
  1. **RNN (Recurrent Neural Networks)**
     - Understand sequence of words.
     - Good for language modeling & sequence prediction.
  2. **LSTM / GRU**
     - Solve RNN's vanishing gradient problem.
     - Used in early chatbot and translation systems.
  3. **CNN for text**
     - Extract local patterns (similar to n-grams).
     - Good for sentence classification.
  4. **Transformer-based Models (SOTA)**
     - These are currently the most used and **industry standard**.
     - Examples:
       - **BERT** (Google)
       - **RoBERTa**
       - **ALBERT**
       - **GPT family (OpenAI)**
       - **T5 (Google)**
     - They use **attention mechanisms** to understand context in both directions.

Fundamental Algorithms in NLP

3.8 Fundamental algorithms for NLP

- NLP uses a combination of linguistic rules, statistical methods, and modern deep-learning techniques. Below are the core algorithms that form the building blocks of most NLP systems.
  1. Text Preprocessing Algorithms
  2. Text Representation Algorithms
  3. Statistical NLP Algorithms
  4. Machine Learning Algorithms
  5. Sequence Models
  6. Attention & Transformer Algorithms
  7. Text Generation Algorithms
  8. Similarity & Retrieval Algorithms
  9. Evaluation Algorithms

1. Text Preprocessing Algorithms

- These algorithms prepare raw text so that models can understand it.

a) Tokenization

- Splits text into smaller units (words, sentences, subwords).
- **Algorithms/Methods:**
  - Whitespace tokenization
  - Rule-based tokenizers
  - Byte Pair Encoding (BPE)
  - WordPiece (used in BERT)
  - SentencePiece (used in modern models)

b) Stemming

- Reduces words to their base form by chopping suffixes.
- **Algorithms:**
  - Porter Stemmer
  - Snowball Stemmer
  - Lancaster Stemmer

c) Lemmatization

- Returns the dictionary form of a word using grammar rules.
- **Algorithms:**
  - WordNet Lemmatizer

2. Text Representation Algorithms

- These convert text into numerical form (vectors).

a) Bag-of-Words (BoW)

- Represents text by word counts → simple but ignores order.

b) TF–IDF

- Improved BoW → reduces importance of common words.

c) Word Embeddings

- Represent words as dense vectors.
- **Algorithms:**
  - **Word2Vec** (Skip-gram, CBOW)
  - **GloVe** (Global Vectors)
  - **FastText** (word + subword embeddings)

d) Contextual Embeddings

- Meaning depends on context.
- **Algorithms:**
  - ELMo
  - BERT
  - GPT
  - RoBERTa

3. Statistical NLP Algorithms

- These use probability and linguistics.

a) N-grams

- Predicts next word based on previous *n–1* words.

b) Hidden Markov Models (HMM)

- Used for sequential tasks like:
  - Part-of-Speech tagging
  - Named Entity Recognition

c) Naïve Bayes Classifiers

- Common for text classification (spam detection).

4. Machine Learning Algorithms

a) Logistic Regression

- For binary/multi-class text classification.

b) Support Vector Machines (SVM)

- Works well with TF-IDF for sentiment analysis.

c) Decision Trees / Random Forest

- Simple models for classification tasks.

d) K-Means Clustering

- Used for topic grouping, document clustering.

5. Sequence Models

- These handle data with order (text, speech, etc.)

a) Recurrent Neural Networks (RNNs)

- Process sequences word by word.

b) LSTM (Long Short Term Memory)

- Handles long-range dependencies.

c) GRU (Gated Recurrent Unit)

- Simpler, faster than LSTM.

6. Attention & Transformer Algorithms

- **a) Attention Mechanism**
  - Allows the model to focus on important words.
- **b) Transformer Architecture**
  - Currently the foundation of advanced NLP.
  - Used in:
    - BERT
    - GPT series
    - T5
    - Transformer-XL
  - Key components:
    - Multi-head attention
    - Positional encoding
    - Feed-forward layers

7. Text Generation Algorithms

a) Language Models

- Predict next word or sentence.
- GPT
- LLaMA
- Transformer-based models

b) Beam Search

- Improves the quality of generated text.

c) Sampling Methods

- Greedy search
- Top-k sampling
- Nucleus sampling (top-p)

8. Topic Modeling Algorithms

a) Latent Dirichlet Allocation (LDA)

- Probabilistic model for discovering topics.

b) Non-negative Matrix Factorization (NMF)

- Matrix-based topic extraction.

9. Similarity & Retrieval Algorithms

a) Cosine Similarity

- Measures similarity between two text vectors.

b) BM25

- Advanced algorithm for search engines.

c) Semantic Search (Embedding-based)

- Uses vector similarity with large language models.

10. Evaluation Algorithms

a) BLEU Score

- Evaluates machine translation.

b) ROUGE Score

- Evaluates text summarization.

c) Perplexity

- Evaluates language models.

Model Evaluation

Model Evaluation

- Model evaluation helps us understand **how well an NLP model performs** on real-world tasks. It measures the accuracy, quality, and reliability of the system.
- Model evaluation differs based on the **type of NLP task**:
  - Classification
  - Sequence labeling
  - Language modeling
  - Machine translation
  - Summarization
  - Information retrieval
  - Embedding-based similarity

Train–Validation–Test Split

- Before evaluation, data is divided into three parts:
- **a) Training Set**
  - Used to train the model.
- **b) Validation Set**
  - Used for hyperparameter tuning (e.g., learning rate, dropout).
- **c) Test Set**
  - Used for final evaluation.

- **Important:** The model should not “see” test data during training.

Evaluation Metrics for Classification Tasks

- Used for:
  - Sentiment analysis
  - Spam detection
  - Text categorization
- a) Accuracy:
  - But accuracy can be misleading if the dataset is imbalanced.

$$\text{Accuracy} = \frac{\text{Correct Predictions}}{\text{Total Predictions}}$$

- b) Precision:
  - "How many predicted positive are actually positive?"

$$\text{Precision} = \frac{TP}{TP + FP}$$

- **c) Recall**
  - "How many actual positives did we correctly detect?"

$$\text{Recall} = \frac{TP}{TP + FN}$$

- **d) F1-Score**
  - Harmonic mean of precision and recall.
  - Useful when there is class imbalance.

$$F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

- **e) Confusion Matrix**
  - Matrix showing:
    - True Positive (TP)
    - True Negative (TN)
    - False Positive (FP)
    - False Negative (FN)
  - Helps visualize performance.

Evaluation Metrics for Sequence Labeling

- Used for:
  - Named Entity Recognition (NER)
  - Part-of-Speech (POS) tagging
  - Chunking
- Key metric:
  - **Token-Level or Entity-Level F1 Score**
    - Token-level: evaluates each word.
    - Entity-level: evaluates full entities (more strict).

Evaluation Metrics for Language Models

- Used for:
  - Next-word prediction
  - Text generation
- **a) Perplexity**
  - Measures how well a model predicts the next word.
  - **Lower perplexity = better model**.

$$\text{Perplexity} = 2^{-\frac{1}{N}\sum \log_2 P(w_i)}$$

Evaluation Metrics for Machine Translation

- Used for:
  - English → Nepali translation
  - Nepali → Hindi, etc.
- **a) BLEU Score (Bilingual Evaluation Understudy)**
  - Measures similarity between:
  - model-generated translation
  - reference (human-written) translation
  - Ranges from **0 to 1** (or 0 to 100).
    Higher = better translation.
- **b) METEOR Score**
  - Considers:
    - synonyms
    - stemming
    - word order
  - Often more accurate for languages with rich morphology.
- **c) TER (Translation Error Rate)**
  - Measures number of edits needed to correct a translation.
  - Lower TER = better.

Evaluation Metrics for Summarization

- Used for:
  - Text summarization
  - News summarization
- **ROUGE Score**
  - Types:
    - **ROUGE-N** (unigram, bigram overlap)
    - **ROUGE-L** (longest common subsequence)
  - Higher ROUGE = better summary.

Evaluation Metrics for Information Retrieval

- Used for:
  - Search engines
  - Document retrieval
  - Question answering
- **a) Precision@k**
  - Precision of top-k search results.
- **b) Recall@k**
  - Recall at top-k results.
- **c) Mean Average Precision (mAP)**
  - Measures ranking quality.
- **d) BM25 Score**
  - Ranking algorithm used in search systems.

Evaluation for Embeddings & Similarity Tasks

- Used for:
  - Semantic similarity
  - Question-answer matching
  - Duplicate detection
- **a) Cosine Similarity**
  - Measures angle between two vectors.
  - Ranges from -1 to 1.
  - Higher = more similar.
- **b) Intrinsic Evaluation**
  - Examples:
    - Word analogy tasks (king - man + woman = queen)
    - Similarity benchmarks (SimLex, WordSim)

End of Chapter