Syllabus NLP UNIT II

UNIT-II:

Word and document embeddings: Introducing embeddings, what are Word embeddings, why are they important, building blocks of language: Characters, words, and phrases, Tokenization, stemming, and lemmatization, Skip-gram and continuous bag of words (CBOW), Glove, fastText, Document-level embeddings, Visualizing embeddings.

A typical text processing workflow - Data Collection and labelling, collecting labelled data, Text normalization, tokenization, stop word removal, part-of-speech tagging, Vectorizing text, count-based vectorization, term frequency-Inverse document frequency, word vectors.

2.1 Embeddings:

Definition: Embeddings are real-valued vector representations of discrete symbols like words, characters, or sentences. These vectors capture the semantic meaning of items in a way that computers can process—because computers understand numbers, not words.

What Are Word Embeddings?

Word embeddings are one of the most important concepts in modern NLP. Word *embedding is a continuous vector representation of something that is usually discrete. Simply we can say it's a* continuous vector representation that capture the sematic and syntactic in the sentences.

A word embedding is a way to represent a word using a list of real numbers, also known as a vector. This is crucial because computers cannot understand words directly—they understand numbers. For instance, the words *"cat"*, *"dog"*, and *"pizza"* might be represented as follows:

- vec("cat") = [0.7, 0.5, 0.1]
- vec("dog") = [0.8, 0.3, 0.1]
- vec("pizza") = [0.1, 0.2, 0.8]

These embeddings can be visualized in a 3-dimensional space since each vector has three values. As illustrated in Figure 3.1, similar words like *"cat"* and *"dog"* are placed near each other, indicating they are semantically related (both are animals or pets). Meanwhile, a word like *"pizza"*, which belongs to a different category (food), appears farther away in that space.

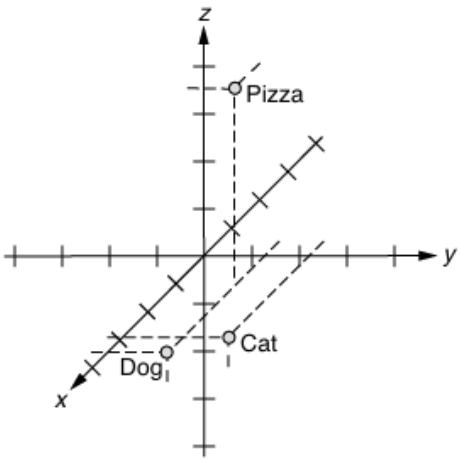


Figure 3.1 Word embeddings on a 3-D space

This idea isn't limited to words. You can embed anything symbolic—such as characters, phrases, sentences, or even categories. For example, in a movie recommendation system, you could embed genres like *"comedy"*, *"drama"*, or *"thriller"* using the same method. Each would be a vector learned based on patterns in user preferences.

Why Are Embeddings Important?

Embeddings are vital because neural networks can only work with numbers, not symbols. Natural language is made up of symbolic data—words, tags, grammar, etc.—that computers don't natively understand. To make neural networks usable for NLP, we convert these symbols into numbers, and embeddings make this possible.

For example, in a sentiment analysis task, the model needs to understand that the sentence *"The movie was amazing"* is positive. To do this, it must convert each word into a vector that captures its meaning. These word vectors are then fed into the model, as shown in Figure 3.2, to predict the sentiment.

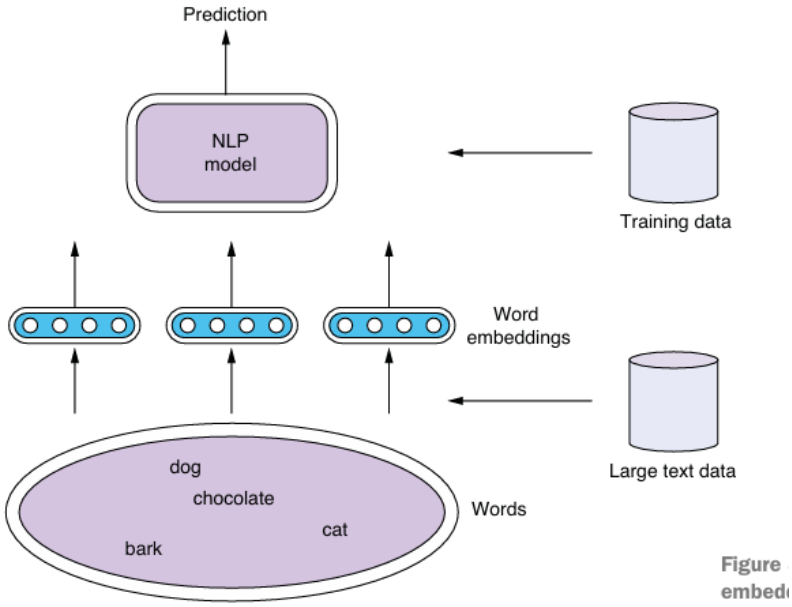


Figure 3.2 Using word embeddings with NLP models

Here's a simple example: If *"amazing"* and *"awesome"* are both associated with positive reviews, their embeddings will be close in vector space. If a new sentence contains *"awesome"*, the model generalizes based on its similarity to *"amazing"*—even if *"awesome"* didn't appear in the training data.

How Are Embeddings Used in NLP?

There are three main ways to use word embeddings in an NLP model:

➤ Scenario 1: Train Embeddings from Scratch

In this approach, embeddings are randomly initialized and trained alongside your NLP model using the available labeled dataset. For instance, when building a model to detect spam emails, you may start with random vectors for each word like *"free"*, *"offer"*, and *"win"*. During training, the model updates these vectors based on how strongly they indicate spam.

However, this is like teaching a baby to walk and dance at the same time—possible but not efficient. Without any prior knowledge about language, the model must learn both basic word meanings and the specific task.

➤ Scenario 2: Use Pretrained Embeddings and Fine-Tune

A better approach is to first train embeddings on a large corpus (like Wikipedia or Common Crawl) or use publicly available ones such as Word2Vec, GloVe, or FastText. Then, use these embeddings to initialize your NLP model and continue training them along with your model.

For example, if you're building a chatbot, initializing with pretrained embeddings helps the model already know that *"hi"*, *"hello"*, and *"hey"* are similar greetings. Training on your chatbot data then fine-tunes this knowledge to the context of your application.

This approach is like teaching a child to walk first, then dance—a more natural and efficient progression.

➤ Scenario 3: Use Pretrained Embeddings Without Fine-Tuning

This is similar to Scenario 2, except the pretrained embeddings are kept fixed during training. The model uses them as they are without updating. This is useful when:

- Your training data is small.
- You want to preserve the general-purpose semantic structure learned from a large corpus.

Think of it as using printed maps for directions instead of adjusting them on the go. It's safer but less flexible.

Transfer Learning and Embeddings

Using pretrained embeddings falls under a broader strategy called transfer learning—where a model trained on one task is adapted for another. In NLP, this is especially powerful because large unlabeled text datasets (like Wikipedia) are freely available, but labeled data (e.g., for emotion detection, question answering) is scarce and expensive to obtain.

Just as kids pick up walking from casual experience before enrolling in a dance class, your model benefits from learning general language structure before focusing on specific tasks.

Whether you fine-tune embeddings (Scenario 2) or keep them fixed (Scenario 3) depends on the problem. Fine-tuning might improve performance on complex tasks, just like ballet training can enhance walking posture. But in some cases, fixing embeddings provides more stability.

Where Do Embeddings Come From?

Embeddings are learned by training models on massive text corpora using unsupervised learning. This means no labeled data is needed—just plain text. Models like Word2Vec, GloVe, and FastText predict word contexts or co-occurrence statistics to learn these vector representations.

For example, if *"king"* appears frequently near *"royal"*, *"crown"*, and *"queen"*, then Word2Vec might learn:

- vec("king") - vec("man") + vec("woman") ≈ vec("queen")

This famous analogy demonstrates how embeddings capture semantic relationships through simple arithmetic.

2.2 Building Blocks of Language: Characters, Words, and Phrases

Before diving into word embedding models, it's essential to understand the basic building blocks of language: **characters**, **words**, and **phrases**. These elements form the foundation of any text and play a key role in designing Natural Language Processing (NLP) applications. **Figure 3.3** provides examples of each of these components.

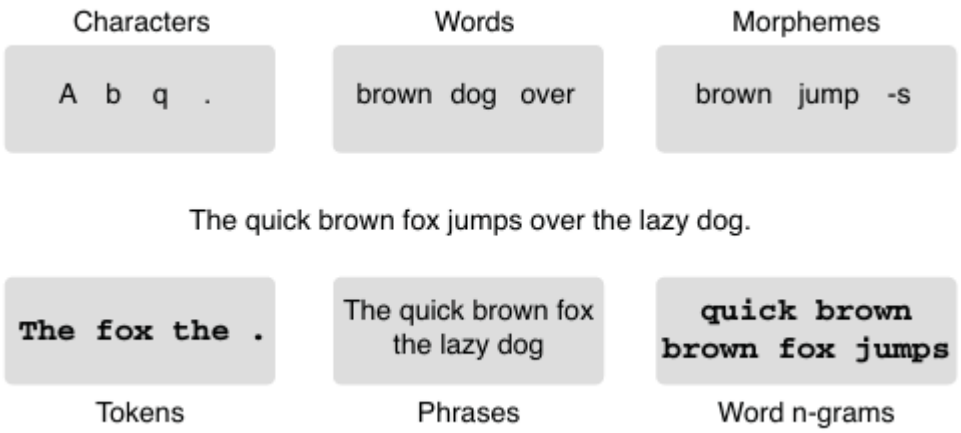


Figure 3.3 Examples of characters, words, and phrases

A **character** is defined as the smallest unit of a writing system. In linguistic terms, it is also called a **grapheme**. In written English, individual letters like “a,” “b,” or “z” are characters. Characters by themselves typically do not carry meaning and do not necessarily correspond to a fixed sound when spoken. However, in some languages, such as Chinese, many characters are meaningful and can represent words on their own. Technically, characters are represented by **Unicode codepoints**, which allow a standardized digital representation of text. For example, the letter “a” is represented as \u0061 in Unicode. In some cases, a single character may require more than one Unicode codepoint—such as characters with accent marks like “é.” Additionally, punctuation marks including period (“.”), comma (“,”), and question mark (“?”) are also considered characters.

A **word** is defined as a meaningful unit composed of one or more characters. In most NLP tasks, words are the most commonly used linguistic units. They are identified through a process called **tokenization**, which splits a sentence into individual words. Each word usually represents a distinct meaning or function in a sentence. Examples of words include “cat,” “running,” and “India.” Words are essential for tasks like sentiment analysis, text classification, and machine translation.

A **phrase** is a group of two or more words that together form a meaningful unit within a sentence. Phrases often carry more contextual or semantic meaning than individual words. For example, the phrase “black cat” refers to a specific concept that is not fully captured by “black” or “cat” alone. Similarly, “New York” and “artificial intelligence” are phrases that represent named entities or compound concepts. In NLP, recognizing and processing phrases accurately is important for applications like named entity recognition, question answering, and translation.

Understanding these definitions and distinctions—**character (grapheme)**, **word**, and **phrase**—is crucial for building effective NLP pipelines. They determine how text is broken down, analyzed, and interpreted by computational models.

N-grams

In Natural Language Processing (NLP), you'll often encounter the concept of **n-grams**, which are defined as **contiguous sequences of one or more linguistic units**, such as characters or words.

An **n-gram** can be composed of **words** (word n-grams) or **characters** (character n-grams), depending on the application.

For example, in **word n-grams**, the sentence "the quick brown fox jumps" contains the following:

- **Unigram (n = 1):** "the", "quick", "brown", "fox", "jumps"
- **Bigram (n = 2):** "the quick", "quick brown", "brown fox", "fox jumps"
- **Trigram (n = 3):** "the quick brown", "quick brown fox", "brown fox jumps"

In contrast, **character n-grams** are sequences of characters. Taking the word **"brown"** as an example:

- **Unigram:** "b", "r", "o", "w", "n"
- **Bigram:** "br", "ro", "ow", "wn"
- **Trigram:** "bro", "row", "own"

An n-gram with **n = 1** is called a **unigram**, **n = 2** is a **bigram**, and **n = 3** is a **trigram**. These patterns are useful because they help capture local context and sequential relationships between elements in text.

Word n-grams are frequently used in NLP as rough approximations of **phrases**. When all n-grams are generated from a sentence, many of them correspond to meaningful units—for example, "Los Angeles" or "take off". These are important in tasks like text classification, language modeling, and machine translation.

Similarly, **character n-grams** are especially helpful for capturing **subword information**, such as **morphemes**—the smallest units of meaning in a language (like "un-" or "-ing"). This makes them valuable in tasks like spelling correction, language modeling for morphologically rich languages, and handling out-of-vocabulary words.

Note: In fields like **information retrieval** and **search engines**, the term "n-gram" often refers specifically to **character n-grams** used for indexing and document matching. It's important to understand from the context whether word-level or character-level n-grams are being discussed when reading academic papers or technical documents.

2.3 Tokenization, Stemming, and Lemmatization

Before feeding text into an NLP model, it needs to be **preprocessed**. Three important steps in this process are **tokenization**, **stemming**, and **lemmatization**. These help convert raw text into analyzable linguistic units.

3.3.1 Tokenization

Definition:
Tokenization is the process of breaking a string of text into smaller pieces called **tokens**. These can be **words** or **sentences**.

- **Word tokenization** splits a sentence into words and punctuation marks.

- **Sentence tokenization** splits a paragraph or document into sentences.

If someone says “tokenization” without context, they usually mean **word tokenization**.

Example Text:

s = '''Good muffins cost $3.88\nin New York. Please buy me two of them.\n\nThanks.'''

Using NLTK:

```
from nltk.tokenize import word_tokenize, sent_tokenize

word_tokenize(s)

# ['Good', 'muffins', 'cost', '$', '3.88', 'in', 'New', 'York', '.', 'Please', 'buy', 'me', 'two', 'of', 'them', '.', 'Thanks', '.']

sent_tokenize(s)

# ['Good muffins cost $3.88\nin New York.', 'Please buy me two of them.', 'Thanks.']
```

Using spaCy:

```
import spacy

nlp = spacy.load('en_core_web_sm')

doc = nlp(s)

[token.text for token in doc]

# ['Good', 'muffins', 'cost', '$', '3.88', '\n', 'in', 'New', 'York', '.', ' ', 'Please', 'buy', 'me', 'two', 'of', 'them', '.', '\n\n', 'Thanks', '.']

[sent.string.strip() for sent in doc.sents]

# ['Good muffins cost $3.88\nin New York.', 'Please buy me two of them.', 'Thanks.']
```

Note: NLTK and spaCy differ slightly. For instance, spaCy retains newline characters (\n) as tokens.

Advanced Tokenization:
Byte-Pair Encoding (BPE) is a popular method in neural NLP. It tokenizes based on **character statistics** rather than spaces/punctuation. BPE helps handle unknown or rare words and is widely used in models like BERT and GPT.

Advantages:

- Fundamental step for all NLP models.
- Well-supported in major NLP libraries.
- Easily customizable.

Challenges:

- Language-dependent.
- Different tools give different outputs.
- Handling compound words or abbreviations can be tricky.

3.3.2 Stemming

Definition:
Stemming reduces words to their **base or root form**, called a **stem**, by chopping off prefixes/suffixes.

- Example:
  - "apples" → "apple"
  - "meets" → "meet"
  - "unbelievable" → "believe"

Using NLTK – Porter Stemmer:

```
from nltk.stem.porter import PorterStemmer

stemmer = PorterStemmer()

words = ['caresses', 'flies', 'dies', 'mules', 'denied',

    'died', 'agreed', 'owned', 'humbled', 'sized',

    'meetings', 'stating', 'siezing', 'itemization',

    'sensational', 'traditional', 'reference', 'colonizer',

    'plotted']

[stemmer.stem(word) for word in words]

# ['caress', 'fli', 'die', 'mule', 'deni', 'die', 'agre', 'own', 'humbl', 'size', 'meet', 'state', 'siez', 'item', 'sensat', 'tradit', 'refer', 'colon', 'plot']
```

Note: spaCy doesnot support the stemming process of the inference speed and incorrectness.

Advantages:
- Fast and rule-based.
- Useful in search engines to index root forms.
- Reduces dimensionality in text data.

Disadvantages:
- Often too **aggressive**.
- Doesn't consider **context** or **part-of-speech**.
- Can produce **non-real words** (e.g., "sensational" → "sensat", "flies" → "fli").

Example Issues:
- "colonizer", "colonize", and "colon" are treated as the same.
- "meetings" → "meet", although a better stem might be "meeting".

3.3.3 Lemmatization

Definition:
Lemmatization reduces a word to its **lemma**, the **dictionary form** or **base form** of the word.

- Unlike stemming, lemmatization considers **grammar** and **context**.

- Examples:
    - “meetings” → “meeting” (noun)
    - “met” → “meet” (verb)

Using NLTK – WordNet Lemmatizer:

```
from nltk.stem import WordNetLemmatizer

lemmatizer = WordNetLemmatizer()

[lemmatizer.lemmatize(word) for word in words]

# ['caress', 'fly', 'dy', 'mule', 'denied', 'died', 'agreed', 'owned', 'humbled', 'sized',

# 'meeting', 'stating', 'siezing', 'itemization', 'sensational', 'traditional', 'reference', 'colonizer', 'plotted']
```

NLTK's lemmatizer treats everything as a **noun by default**, which may give inaccurate results for verbs.

Using spaCy:

```
doc = nlp(' '.join(words))

[token.lemma_ for token in doc]

# ['caress', 'fly', 'die', 'mule', 'deny', 'die', 'agree', 'own', 'humble', 'sized',

# 'meeting', 'state', 'siezing', 'itemization', 'sensational', 'traditional', 'reference', 'colonizer', 'plot']
```

spaCy infers **parts of speech** from the context and gives more accurate lemmas.

Advantages:

- More accurate than stemming.
- Handles **irregular words** (e.g., “went” → “go”, “better” → “good”).
- Respects linguistic rules and grammar.

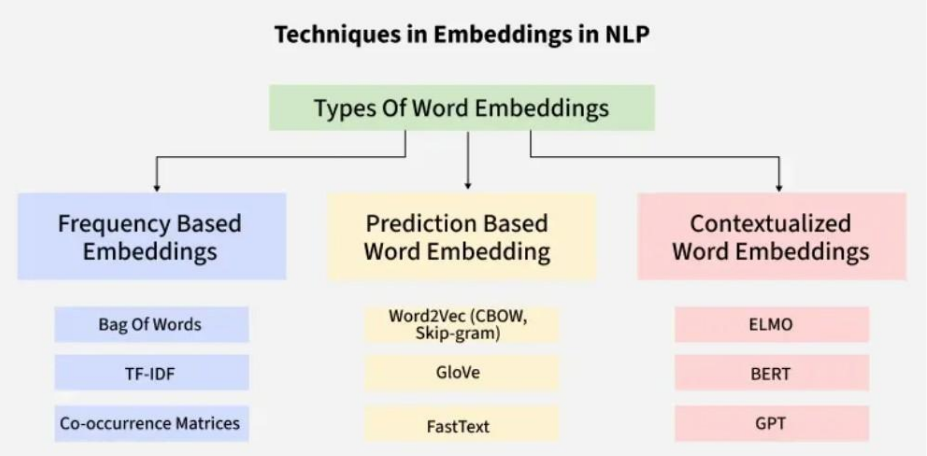
Disadvantages:

- **Slower** and more **resource-intensive**.
- Needs **POS tagging** or linguistic resources like WordNet.
- May still make errors depending on context and language model.

Summary Table

Task	Definition	Example	Pros	Cons
Tokenization	Split text into words/sentences	"New York" → ["New", "York"]	Fast, essential preprocessing step	Tool-dependent results
Stemming	Chop affixes to get root form	“running” → “run”	Fast, reduces vocabulary size	Can over-stem or distort words
Lemmatization	Get dictionary base form with grammar info	“met” → “meet”	Accurate, context-aware	Slower, requires more computation

2.4 Approaches for Generating the Word Embeddings for text

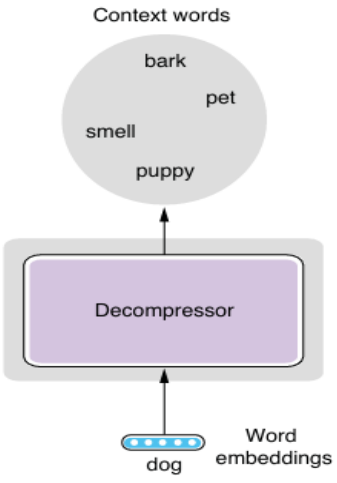


Prediction based Traditional - Word Embedding Model

a. Skip-gram Model

Word embeddings are a way to represent each word in a vocabulary using a list (or vector) of floating-point numbers. For example, the word “cat” might be represented as [0.7, 0.5, 0.1] and “dog” as [0.8, 0.3, 0.1]. Similar words end up with similar vectors. These numbers aren’t chosen manually; they’re learned from large text datasets using machine learning techniques. One popular model used to learn these vectors is the Skip-gram model, which is simple yet powerful for this task.

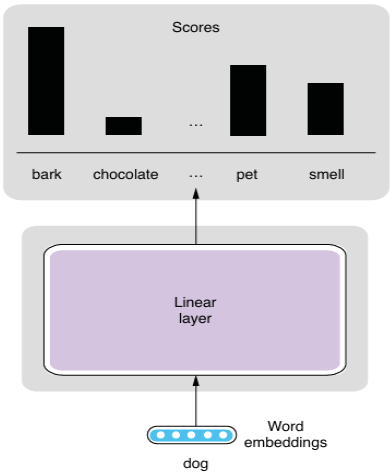
To understand how the Skip-gram model works, let’s think about how humans learn what words mean. For example, no one formally teaches you what a dog is — you learn it from experience, through seeing, hearing, and reading about dogs. Computers can’t physically experience the world like humans, but they can look at text. If the word “dog” often appears near words like “bark,” “puppy,” and “pet,” then the model can learn that “dog” is likely related to these words. Similarly, the word “cat” might appear near “meow,” “kitten,” and “tail.” Since both “dog” and “cat” share many similar context words, we can infer they are conceptually related. This idea is known as the **distributional hypothesis**—words that occur in similar contexts tend to have similar meanings.

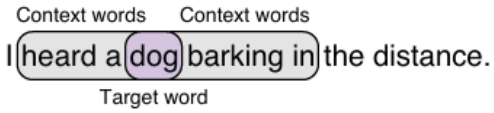


To represent the relationship between a word and its context, we can assign scores to each context word. For instance, “dog” might be associated with {"bark": 1.4, "smell": 0.6, "pet": 1.2, ...}. These scores can be converted into a long vector where each position represents a word in the vocabulary. This process is shown conceptually in **Figure 3.4**, where we treat a word embedding as a compressed form of these context scores, and the model learns to “decompress” them using a mathematical function.

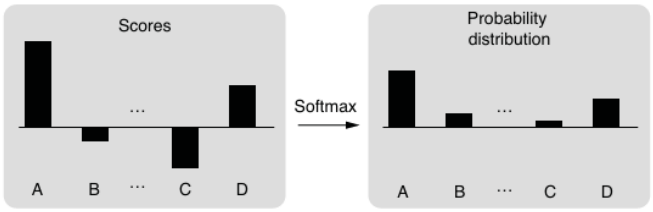
This decompression is done using something called a **linear layer**). A linear layer is a mathematical operation that transforms one vector into another using weights and biases. Think of it like a function output = weight × input + bias. If the input is a 3-dimensional vector (like the word embedding), and we want to output a large vector (one score per vocabularyword), the linear layer handles this transformation. It’s the simplest and most fundamental type of layer in a neural network. This setup is illustrated in **Figure** where the Skip-gram model consists of a word embedding feeding into a linear layer, producing scores for all possible context words.

The model is trained by creating a "fake" task: given a word like "dog" (the **target word**), the model tries to predict the nearby words (the **context words**) in a sentence. For example, in the sentence "I heard a dog barking in the distance," with a window size of 2, the context words for "dog" would be "heard," "a," "barking," and "in." This training setup is shown in **Figure 3.6**. Even though the model's goal is to predict context words, what we really want are the useful word embeddings it learns during this process. This technique is called **self-supervised learning**, because we don't need labeled data—the context labels are created automatically from the text itself.



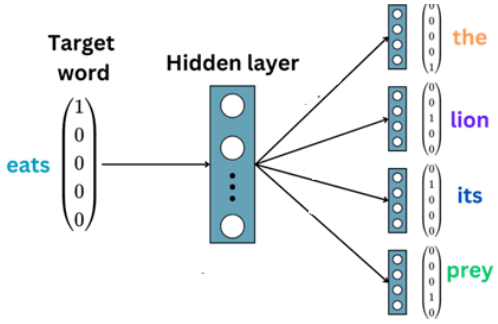


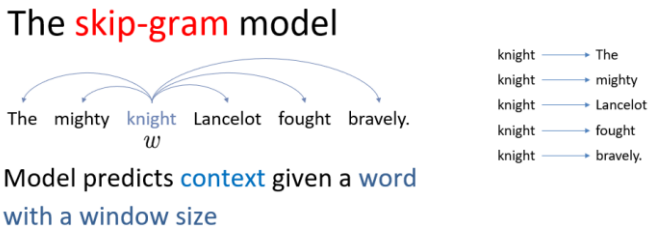
To make predictions, the model first outputs a vector of scores, then applies a function called **softmax**. The softmax function turns these scores into a probability distribution—it "squashes" the numbers into values between 0 and 1 and makes sure they all add up to 1. This allows us to interpret the output as a set of probabilities or confidence scores for each possible context word. **Figure 3.7** illustrates how softmax transforms raw scores into a meaningful distribution.



Finally, to train the model, we compare the predicted probability distribution to the actual correct context word using a mathematical tool called **cross entropy**. Cross entropy measures the difference between two probability distributions. If the prediction is perfect, the cross entropy is zero; if it's wrong, the value is higher. By minimizing cross entropy during training, the model improves its predictions and the word embeddings become more accurate.

In practice, word embeddings are usually around 100 dimensions or more, and the training process involves tweaking millions of numbers (called parameters, which include the weights and biases of the model). Deciding the exact values of these numbers—whether a "0.8" should be assigned to the first element of the "dog" vector or not—is not something we can do by hand. The model figures this out automatically through training.



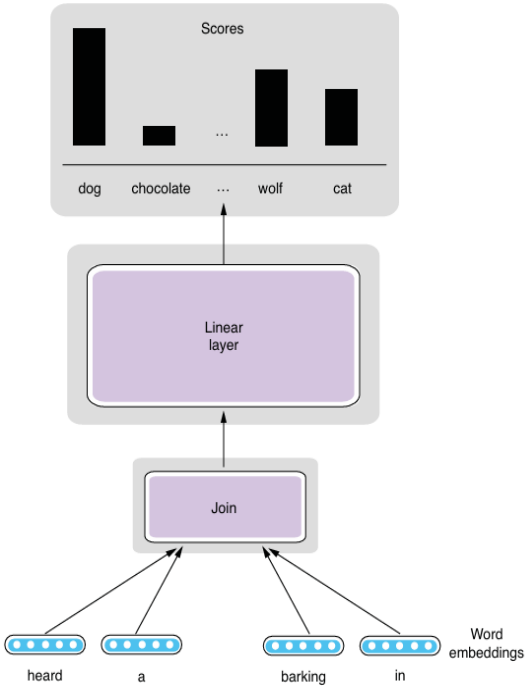


Comparison of CBOW vs Skip-gram

Aspect	**CBOW**	**Skip-gram**
Input	Context words	Target word
Output	Target word	Context words
Efficiency	More efficient (faster training)	Less efficient
Performance on Tasks	Slightly lower semantic accuracy	Better accuracy, especially on rare words
Preferred Use	Frequent words	Rare words

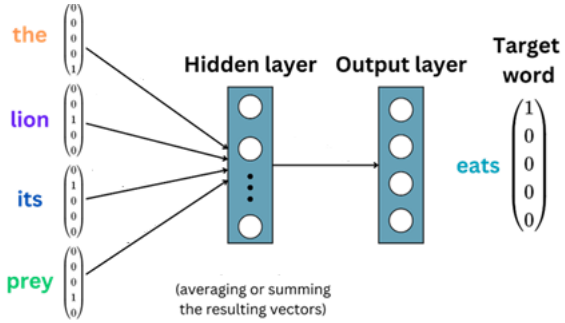
b) Continuous Bag of Words (CBOW)

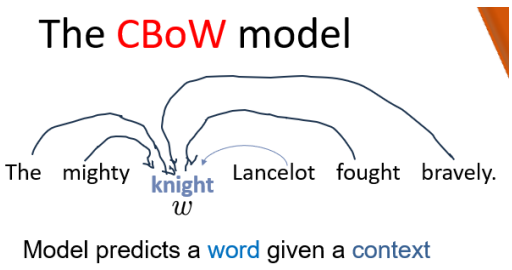
The Continuous Bag of Words (CBOW) model is a foundational word embedding technique introduced by Mikolov et al. in 2013, alongside the Skip-gram model.
While both models share architectural similarities, the CBOW model can be thought of as the inverse of the Skip-gram model.



In CBOW, the model is trained to predict a target (center) word from a set of surrounding context words. This setup is analogous to solving a fill-in-the-blank type of question. For example, given the sentence ***"I heard a ___ barking in the distance,"*** the model should be able to predict the missing word "dog" based on the surrounding context words like "I," "heard," "a," "barking," and so on.
The **architecture of CBOW**, illustrated in Figure 3.8, shows how multiple context word embeddings are averaged (or summed) to create a single representation, which is then used to predict the target word. Only one output word is generated per input context in each training instance. This architecture makes CBOW relatively faster and more efficient to train, especially on large corpora with frequent words.
When compared with the Skip-gram model, CBOW tends to perform slightly worse on word semantic similarity tasks. However, it is generally more efficient and performs well with frequent words. In contrast, the Skip-gram model is better suited for learning representations of rare words, albeit with higher computational cost. Both models were included in the original Word2Vec toolkit (https://code.google.com/archive/p/word2vec/), and were widely used in early NLP applications.

Despite its historical significance, CBOW is less used in modern NLP pipelines due to the availability of more advanced word embedding models like GloVe (Global Vectors for Word Representation) and fastText, which capture more nuanced semantic and subword-level information. Nonetheless, understanding CBOW provides an essential foundation for grasping how neural word embeddings work and how distributional semantics can be learned using shallow neural networks.





Generating word embedding using Word2Vec

```
import matplotlib.pyplot as plt
from nltk.tokenize import word_tokenize, sent_tokenize
from nltk.corpus import stopwords
from gensim.models import Word2Vec
from sklearn.decomposition import PCA
```

```
# === Step 1: Load text from file ===
file_path = r"C:\Users\kamli\1NLP25-26\rawtext.txt"  # Change to your file path
with open(file_path, "r", encoding="utf-8") as f:
    raw_text = f.read().strip()
print(raw_text[0:2000]) # read only 2000 tokens
```

```
Cloud computing is "a paradigm for enabling network access to a scalable and elastic pool of shareable physical or virtual resources with self-service provisioning and administration on-demand," according to ISO.


== Essential characteristics ==
In 2011, the National Institute of Standards and Technology (NIST) identified five "essential characteristics" for cloud systems. Below are the exact definitions according to NIST:

On-demand self-service: "A consumer can unilaterally provision computing capabilities, such as server time and network storage, as needed automatically without requiring human interaction with each service provid
Broad network access: "Capabilities are available over the network and accessed through standard mechanisms that promote use by heterogeneous thin or thick client platforms (e.g., mobile phones, tablets, laptops,
Resource pooling: " The provider's computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned accordin
Rapid elasticity: "Capabilities can be elastically provisioned and released, in some cases automatically, to scale rapidly outward and inward commensurate with demand. To the consumer, the capabilities available
Measured service: "Cloud systems automatically control and optimize resource use by leveraging a metering capability at some level of abstraction appropriate to the type of service (e.g., storage, processing, ban
By 2023, the International Organization for Standardization (ISO) had expanded and refined the list.
```

```
# === Step 2: Preprocess the text ===
def preprocess_text(sentence):
    words = word_tokenize(sentence.lower())
    cleaned = []
    for word in words:
        if word.isalpha():  # keep only alphabetic words
            if word not in stopwords.words('english'):
                if len(word) > 2:
                    cleaned.append(word)
    return cleaned

Sentences = sent_tokenize(raw_text)
tokenized_sentences = [preprocess_text(sent) for sent in sentences if preprocess_text(sent)]
```

```
# === Step 3: Train Word2Vec ===
model = Word2Vec(
    sentences=Sentences,
    vector_size=100,
    window=5,
    #min_count=1,
   #workers=4,
    #sg=0
)
```

```
#=== Step 4: printing the value of the embedding ===
X = model.wv[model.wv.index_to_key]
#print (len(tokenized_sentences))
print(X[0])
print(X[0].shape)
```

```
Embedding  with word2vec:
[ 9.59144381e-04  4.23873402e-03  4.73224046e-03  8.03092215e-03
-8.55783839e-03 -1.27106290e-02  1.15346750e-02  1.66230146e-02
-9.50790104e-03 -7.77277537e-03  7.08151422e-03 -9.03903600e-03
-2.22231005e-03  1.11327814e-02 -2.77458108e-03 -4.10200842e-03
 3.36871063e-03 -1.52737787e-03 -1.11589599e-02 -1.86882336e-02
 1.01818070e-02  7.30626751e-03  8.27749539e-03 -3.83280381e-03
 6.75958069e-03 -3.09467176e-03 -1.12909870e-03  2.53198668e-03
-1.05363382e-02 -5.26565127e-03 -4.24474431e-03 -9.79950186e-04
 1.21153444e-02 -1.23002445e-02 -5.37115848e-03  4.07625083e-03
 1.02058081e-02 -7.93583039e-03  7.10119202e-04 -1.14804916e-02
-7.38994265e-03  3.32175405e-03 -9.89814568e-03 -4.13692091e-03
 3.53680248e-03  1.59029779e-03 -1.24262562e-02  7.85195641e-03
 6.70068990e-03  1.07863108e-02 -6.95022754e-03  1.41905108e-03
-4.22168290e-03 -1.39693540e-04  8.91489908e-03 -1.11795415e-03
 5.91144292e-03 -6.30621612e-03 -8.01755022e-03  1.04511501e-02
 4.52237291e-04 -1.20726581e-04 -3.24769621e-03 -6.26658183e-03
-4.85169701e-03  6.99033123e-03 -2.45312462e-04  9.26076807e-03
-8.44109897e-03  9.40710213e-03  3.46148852e-03  1.29596572e-02
 1.53373554e-03 -8.56889971e-03  9.77304112e-03  6.73078524e-04
 8.53386801e-03 -6.84402825e-04 -5.34934411e-03 -7.37331202e-03
-1.43028668e-03  1.78791047e-03 -1.00840395e-03  1.11431405e-02
-9.10572149e-03  3.52186144e-05  1.09291468e-02 -1.93483813e-03
 2.20412156e-03  8.90736841e-03  7.37498561e-03  3.85267427e-03
 4.94501134e-03  1.82644173e-03  1.43893352e-02  8.47854465e-03
-7.19913514e-03 -8.40401929e-03  1.86589663e-03  5.46512567e-03]
one X sentence shape of X[0] is : (100,)
```

```
pca = PCA(n_components=2)
result = pca.fit_transform(X)

plt.figure(figsize=(10, 8))
plt.scatter(result[:, 0], result[:, 1])

for i, word in enumerate(model.wv.index_to_key):
    plt.annotate(word, xy=(result[i, 0], result[i, 1]))

plt.title("Word2Vec Word Embeddings (PCA 2D)")
plt.show()
```

0.5s

```
# === Step 5: Word similarity check ===
target_word = "specific"  # Change to any word you know is in your text
if target_word in model.wv:
    print(f"\nTop similar words to '{target_word}':")
    for similar_word, similarity in model.wv.most_similar(target_word, topn=5):
        print(f"{similar_word}: {similarity:.4f}")
else:
    print(f"\nThe word '{target_word}' is not in the vocabulary.")
```

0.0s

```
Top similar words to 'specific':
amazon: 0.2880
applications: 0.2669
one: 0.2660
concerns: 0.2612
including: 0.2589
```

Prediction based - Pretrained word Embedding Model:

C) GloVe – Global Vectors

For modern NLP converting textual data into machine-readable numerical form is essential. One of the most effective and widely used approaches to achieve this is through word embeddings.

What is GloVe?

GloVe (Global Vectors) used for Word Representation is an unsupervised learning algorithm designed to generate dense vector representations also known as embeddings. Its primary objective is to capture semantic relationships between words by analysing their co-occurrence patterns in a large text corpus.

GloVe approach is unique as it effectively combines the strengths of two major approaches:

- Latent Semantic Analysis (LSA) which uses global statistical information.
- Context-based models like Word2Vec which focuses on local word context.

At the core of GloVe lies the idea of mapping each word into a continuous vector space where both the magnitude and direction of the vectors reflect meaningful semantic relationships.

For instance, the relationship captured in a vector equation can be like:

king - man + woman = queen.

To achieve this, GloVe constructs a word co-occurrence matrix where each element reflects how often a pair of words appears together within a given context window. It then optimizes the word vectors such that the dot product between any two-word vectors approximates the pointwise mutual information (PMI) of the corresponding word pair. This optimization allows GloVe to produce embeddings **that effectively encode both syntactic and semantic relationships across the vocabulary.**

How GloVe works?

The creation of a word co-occurrence matrix is the fundamental component of GloVe. This matrix provides a quantitative measure of the semantic affinity between words by capturing the frequency with which they appear together in a given context. Further, by minimising the difference between the dot product of vectors and the pointwise mutual information of corresponding words, GloVe optimises word vectors. It is able to produce dense vector representations that capture syntactic and semantic relationships.

Glove Data

Glove has pre-defined dense vectors for around every 6 billion words of English literature along with many other general-use characters like commas, braces and semicolons. The algorithm's developers frequently make the pre-trained GloVe embeddings available. It is not necessary to train the model from scratch when using these pre-trained embeddings which can be downloaded and used immediately in a variety of natural language processing (NLP) applications. Users can select a pre-trained GloVe embedding in a dimension like 50-d, 100-d, 200-d or 300-d vectors that best fits their needs in terms of computational resources and task specificity.

Here d stands for dimension. 100d means in this file each word has an equivalent vector of size 100. Glove files are simple text files in the form of a dictionary. Words are key and dense vectors are values of key.

d) FastText Architecture

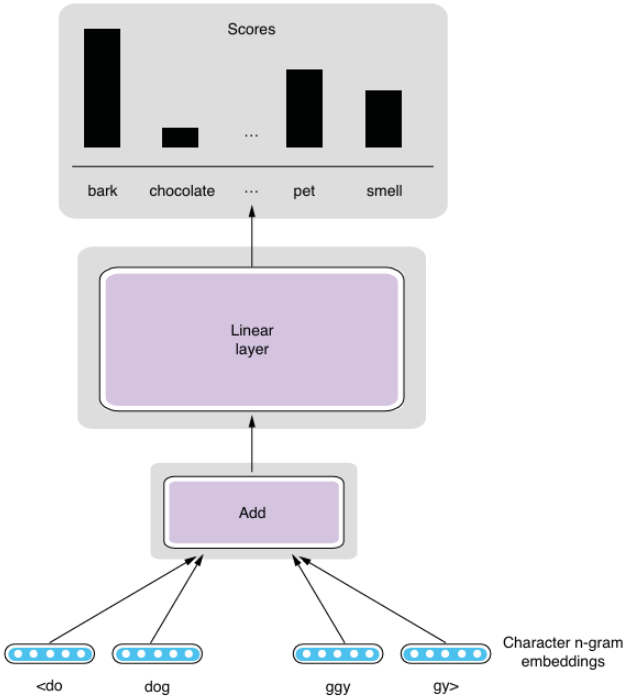
FastText, developed by Facebook, is an extension of the Skip-gram and CBOW word embedding models that incorporates subword information to generate richer and more robust word vectors. Unlike traditional models that treat each word as a single atomic token, FastText represents words as bags of character n-grams. This allows it to capture morphological patterns and generate embeddings for words not seen during training (Out-of-Vocabulary or OOV words) as shown in figure below

Subword-Based Representation

- Words are split into character n-grams, with special boundary symbols to indicate word limits.
- Example (n = 3) for "doggy": <do, dog, ogg, ggy, gy>
- The word vector is obtained by summing the vectors of all its n-grams.

This subword approach learning in FastText

- Capture semantic relationships between morphologically related words (e.g., “run” and “running”).
- Assign embeddings to unseen words by using n-grams that were present in the training data.



Hierarchical Softmax for Efficiency

- FastText uses hierarchical softmax instead of standard softmax to improve computational efficiency.
- Constructs a binary tree where leaves represent words, and internal nodes store probability distributions.
- Uses Huffman coding to prioritize frequent words, reducing time complexity from O(V) to O(log V), where V is the vocabulary size.

Key Advantages

- Handles OOV words effectively.
- Learns morphological and semantic relationships at the character level.
- Maintains prediction accuracy while significantly speeding up training.
- Suitable for morphologically rich languages and domain-specific vocabularies.

2.5 Document-Level Embeddings

Document-Level Embeddings are designed to represent entire sentences, paragraphs, or documents as single vectors. While word embeddings like Word2Vec are effective for word-level tasks, they are limited when applied to larger structures because they tend to ignore word order. For instance, the sentences *“Mary loves John”* and *“John loves Mary”* may result in the same vector representation if you simply average their word embeddings, even though their meanings differ.

To address this issue, **Doc2Vec**, proposed by Le and Mikolov in 2014, learns vector representations for documents of any length — whether a sentence, paragraph, or entire article. This method is sometimes referred to as *Sentence2Vec* or *Paragraph2Vec*.

One of the main approaches within Doc2Vec is **PV-DM (Distributed Memory Model of Paragraph Vectors)**, which works similarly to the Continuous Bag of Words (CBOW) model but with an additional

paragraph vector as input. In this setup, the model takes context words along with the paragraph vector to predict the target word. The paragraph vector acts as a memory that stores extra contextual information not captured by individual words. As shown in *Figure 3.11*, each paragraph has a unique vector (paragraph embedding) that is updated during training to capture the overall meaning of the paragraph.

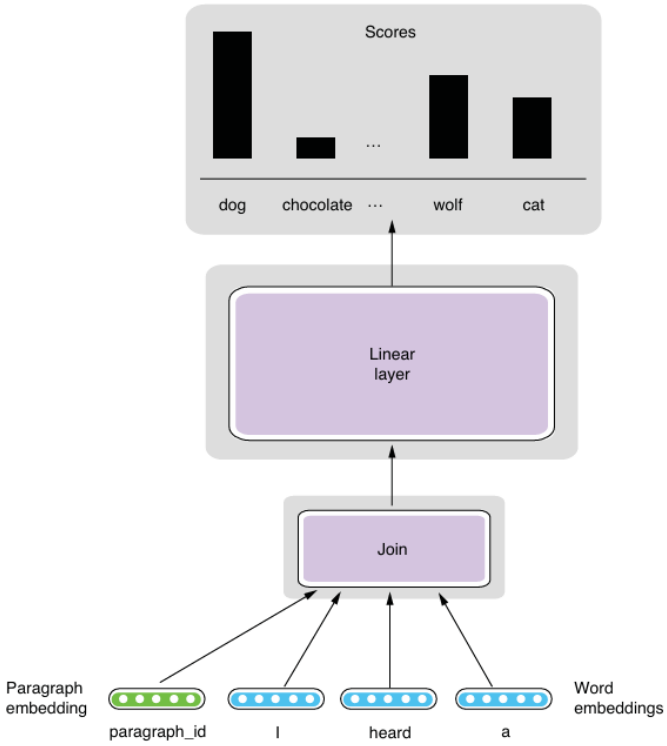


Fig: Distributed memory model of paragraph vectors

Doc2Vec offers several advantages. It captures the meaning of entire documents, can be combined with traditional machine learning algorithms such as Logistic Regression and Support Vector Machines (SVM), and preserves contextual information far better than simple word averaging.

When implementing Doc2Vec in **Gensim**, the typical steps involve converting text into TaggedDocument objects, building a vocabulary, training the model, and then inferring vectors for new documents. Once trained, the model can also retrieve similar documents using the most_similar() function. For example, if the query is *"I heard a dog barking in the distance"*, the retrieved sentences will likely also describe experiences of hearing sounds.

2.6 Visualizing the Word Embeddings (t-SNE)

Word embeddings are N-dimensional vectors, often with 100, 200, or even 300 dimensions. Due to this high dimensionality, it is difficult to directly observe or interpret relationships between words. Visualization transforms these complex numerical representations into a visible form, making it easier to check whether embeddings capture meaningful semantic relationships—for example, ensuring that *king* and *queen* appear close together (Figure 3.1). It also allows us to quickly identify clusters of related words without manually checking each pair.

We cannot directly plot data in more than three dimensions, yet embeddings often have hundreds of dimensions. When N is small (e.g., 3), we can directly plot them in 3D (Figure 3.1). However, for large N, we must use dimensionality reduction—a process that projects high-dimensional data into 2D or 3D while attempting to preserve the important relative distances and patterns between words.

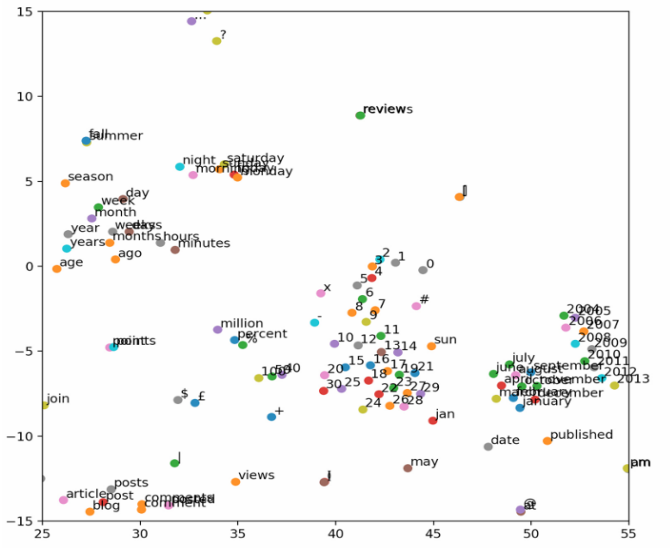
Dimensionality Reduction Techniques

- PCA (Principal Component Analysis) – Finds main axes of variation and is fast but may lose fine-grained neighborhood details.
- ICA (Independent Component Analysis) – Separates independent features, useful in specific contexts.
- t-SNE (t-distributed Stochastic Neighbor Embedding) – The most popular method for visualizing embeddings. It focuses on preserving local neighborhood relationships, grouping similar words into tight clusters (Figure 3.3). This makes it especially effective for revealing semantic patterns in embeddings.

Example Observations from GloVe t-SNE Plot in Figure 3.12 reveals clear semantic groupings:

- Bottom-left: Web-related words (*posts*, *article*, *blog*, *comments*, …).
- Upper-left: Time-related words (*day*, *week*, *month*, *year*, …).
- Middle: Numbers (*0*, *1*, *2*, …) arranged in increasing order toward the bottom, showing that GloVe learned numerical relationships.
- Bottom-right: Months (*January*, *February*, …) and years (*2004*, *2005*, …) arranged chronologically, almost parallel to the number sequence.

These clusters emerge without explicit programming—purely from how the embeddings were trained.



The visualization allows NLP models to start with a built-in understanding of language before being fine-tuned for specific tasks.

Key Points to Remember

Purpose: t-SNE is a tool for exploration and understanding, not for downstream ML tasks.

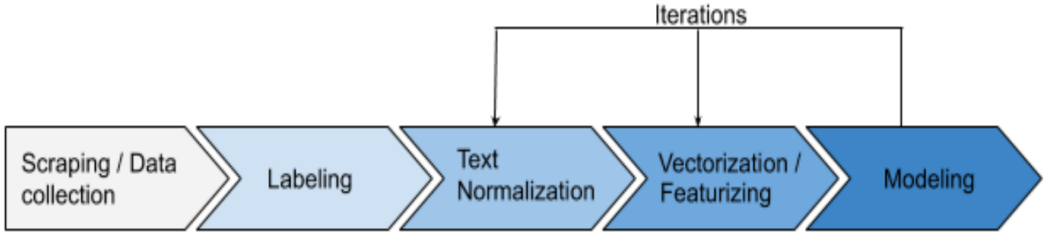
Subset selection: Best used on small subsets (<10,000 words) to avoid clutter and performance issues.

Parameter tuning: Adjusting *perplexity* and *learning rate* can improve clarity.

Interpretation: Always analyze clusters carefully—they reveal how the model organizes language.

2.7 Text Processing workflow:

Text processing is an essential step in Natural Language Processing (NLP) that involves transforming raw text into a clean and structured format suitable for analysis or machine learning models as shown in figure below



Below are the main steps and their sub-steps with explanations and examples

1. Data Collection and Labelling

This is the first step where we gather text data from different sources and (if needed) label it for supervised learning.

Data Sources: Web scraping, APIs (e.g., Twitter API), datasets (Kaggle, UCI ML repository), news articles, emails, etc.

Labelling: Assigning a category or value to each text (e.g., sentiment = positive/negative, topic = sports/politics).

Example:

Collecting tweets about a product and labelling them as positive, negative, or neutral for sentiment analysis.

2. Text Normalization is to make text uniform so that variations in writing don't cause inconsistencies in analysis.

Lowercasing: Convert all text to lowercase.

Example: "Hello World" → "hello world"

Removing punctuation and special characters.

Example: "Hello!!! Are you #excited?" → "hello are you excited"

Expanding contractions.

Example: "don't" → "do not", "I'm" → "I am"

3. Tokenization Breaks the text into smaller units called tokens (words, sentences, or subwords).

The Types are

Word Tokenization: Example: "NLP is fun" → ["NLP", "is", "fun"]

Sentence Tokenization: Example: "I love NLP. It's amazing!" → ["I love NLP.", "It's amazing!"]

Subword Tokenization (used in BERT/GPT):Example: "playing" → ["play", "##ing"]

4. Stop Word Removal

Stop words are common words (like is, the, and, in) that don't add significant meaning and can be removed to reduce noise.

Example:

Input: "The cat is sleeping in the garden"

After stop word removal: "cat sleeping garden"

5. Part-of-Speech (POS) Tagging

POS Tagging is assigning grammatical categories to words (noun, verb, adjective, etc.).

Example:"The dog barked loudly" → [('The', DET), ('dog', NOUN), ('barked', VERB), ('loudly', ADV)]

6. Vectorizing Text : Transforming text into numerical format for machine learning.

Count-Based Vectorization: Counts the frequency of words.

Example: "cat sat" and "cat slept" → {"cat": 2, "sat": 1, "slept": 1}

TF-IDF (Term Frequency – Inverse Document Frequency): Gives higher weight to rare but important words.

Example: In a set of documents, the word "cat" gets a high score if it's frequent in one document but rare in others.

Word Embeddings (Word2Vec, GloVe, FastText): Represents words as dense vectors capturing meaning.

Example: "king" - "man" + "woman" ≈ "queen"

7. Lemmatization and Stemming : Reducing words to their base/root form.

Stemming: Cuts words to their root by rule-based chopping.

Example: "playing" → "play", "studies" → "studi"

Lemmatization: Uses dictionary-based mapping for the correct root word.
Example: "better" → "good", "running" → "run"

8. Handling Special Cases Dealing with emojis, hashtags, numbers, URLs, and domain-specific text.
Examples:
- Emojis: 😊 → "happy face"
- URLs: "https://example.com" → <URL>
- Hashtags: "#NLP" → "NLP"

9. Final Cleaned Data Ready for Modeling
Once all steps are complete, we have structured, clean, and meaningful text data that can be fed into classification, sentiment analysis, translation, or any other NLP model.

Raw text:
"I can't believe how amazing this phone is!!! #BestPurchase 😊 "
Processed:
- Lowercasing → "i can't believe how amazing this phone is!!! #bestpurchase 😊 "
- Remove punctuation → "i cant believe how amazing this phone is bestpurchase 😊 "
- Tokenization → ["i", "cant", "believe", "how", "amazing", "this", "phone", "is", "bestpurchase", " 😊 "]
- Stop word removal → ["cant", "believe", "amazing", "phone", "bestpurchase", " 😊 "]
- Lemmatization → ["cannot", "believe", "amazing", "phone", "bestpurchase", " 😊 "]
- Vectorization → Numerical vector form

2.8 Stopword Removal:

Stop word removal is a text preprocessing technique used in Natural Language Processing (NLP) to eliminate commonly occurring words (like "the," "a," "is") that don't contribute much to the meaning of a text. Removing these words can improve the efficiency and accuracy of various NLP tasks by focusing on more informative terms.

Why remove stop words?

- **Reduce noise:** Stop words are very common and can obscure the actual meaning of the text.
- **Improve efficiency:** Removing them reduces the amount of data to be processed, leading to faster processing times and potentially better model performance.
- **Focus on key information:** By removing less informative words, the focus shifts to the more important content-bearing terms.

Several Python libraries can be used for stop word removal, including:

- NLTK: A popular library with pre-defined stop word lists for various languages.
- spaCy: Another powerful library for NLP tasks, including stop word removal.
- Gensim: A library for topic modeling and document analysis, also offering stop word removal.

NLTK Code

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

nltk.download('stopwords')
nltk.download('punkt')

# Sample text
text = "This is a sample sentence showing stopword removal."

# Get English stopwords and tokenize
stop_words = set(stopwords.words('english'))
tokens = word_tokenize(text.lower())

# Remove stopwords
filtered_tokens = [word for word in tokens if word not in stop_words]

print("Original:", tokens)
print("Filtered:", filtered_tokens)
```

Spacy:

```
import spacy

# Load spaCy model
nlp = spacy.load("en_core_web_sm")

# Sample sentence
text = "This is a sample sentence showing stopword removal."

# Process with spaCy
doc = nlp(text.lower())

# Remove stopwords
filtered_tokens = [token.text for token in doc if not token.is_stop]

print("Original:", [token.text for token in doc])
print("\nFiltered:", filtered_tokens)
```

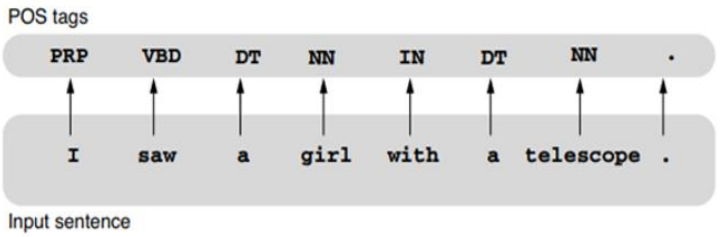
```
Original: ['this', 'is', 'a', 'sample', 'sentence', 'showing', 'stopword', 'removal', '.']

Filtered: ['sample', 'sentence', 'showing', 'stopword', 'removal', '.']
```

2.9 Parts of Speech Tagging

- POS tagging is the process of assigning label to each word in a sentence.
- It is the assignment of corresponding part of speech—such as noun, verb, adjective, or preposition.
- A **part of speech** represents grammatical categories, and understanding them is essential for syntactic and semantic analysis.
- For example, in the sentence

"I saw a girl with a telescope", each word is tagged as shown in **Figure 3**



POS tag	Description
DT	Determiner
IN	Preposition
NN	Noun (singular or mass)
PRP	Pronoun
VBD	Verb (past tense)

Fig 3: Part-of-Speech (POS) Tagging

The tags used—like **PRP** (pronoun), **VBD** (past tense verb), and **NN** (noun)—are part of the **Penn Treebank POS tagset**, which is widely used for training and evaluating POS taggers.

The Stanford NLP Penn Tree Bank POS Tag Set is as shown in figure below:

The Penn Treebank POS tagset.

1. CC	Coordinating conjunction	25. TO	*to*
2. CD	Cardinal number	26. UH	Interjection
3. DT	Determiner	27. VB	Verb, base form
4. EX	Existential *there*	28. VBD	Verb, past tense
5. FW	Foreign word	29. VBG	Verb, gerund/present participle
6. IN	Preposition/subordinating conjunction	30. VBN	Verb, past participle
7. JJ	Adjective	31. VBP	Verb, non-3rd ps. sing. present
8. JJR	Adjective, comparative	32. VBZ	Verb, 3rd ps. sing. present
9. JJS	Adjective, superlative	33. WDT	*wh*-determiner
10. LS	List item marker	34. WP	*wh*-pronoun
11. MD	Modal	35. WP$	Possessive *wh*-pronoun
12. NN	Noun, singular or mass	36. WRB	*wh*-adverb
13. NNS	Noun, plural	37. #	Pound sign
14. NNP	Proper noun, singular	38. $	Dollar sign
15. NNPS	Proper noun, plural	39. .	Sentence-final punctuation
16. PDT	Predeterminer	40. ,	Comma
17. POS	Possessive ending	41. :	Colon, semi-colon
18. PRP	Personal pronoun	42. (	Left bracket character
19. PP$	Possessive pronoun	43.)	Right bracket character
20. RB	Adverb	44. "	Straight double quote
21. RBR	Adverb, comparative	45. '	Left open single quote
22. RBS	Adverb, superlative	46. "	Left open double quote
23. RP	Particle	47. '	Right close single quote
24. SYM	Symbol (mathematical or scientific)	48. "	Right close double quote

Traditionally, algorithms like Hidden Markov Models (HMMs) and Conditional Random Fields (CRFs) were used for POS tagging.

Recently, **recurrent neural networks (RNNs)** have become popular due to their ability to learn from context and deliver higher accuracy. POS tagging results are often used as input for downstream tasks such as parsing and machine translation.

2.10 Vectorization

- Vectorization converts text data into numerical vectors that can be processed by machine learning algorithms.
- *Vectorization is the process of converting text data into numerical vectors.*

- In the context of NLP, vectorization transforms words, phrases, or entire documents into a format that can be understood and processed by machine learning models.
- These numerical representations capture the semantic meaning and contextual relationships of the text, allowing algorithms to perform tasks such as classification, clustering, and prediction.

Why is Vectorization Important in NLP?

Vectorization is crucial in NLP for several reasons:

1. Vectorization converts text into a format that these models can process, enabling the application of statistical and machine learning techniques to textual data.
2. Effective vectorization methods, like word embeddings, capture the semantic relationships between words. This allows models to understand context and perform better on tasks like sentiment analysis, translation, and summarization.
3. Techniques like TF-IDF and word embeddings reduce the dimensionality of the data compared to one-hot encoding. This not only makes computation more efficient but also helps in capturing the most relevant features of the text.
4. Vectorization helps manage large vocabularies by creating fixed-size vectors for words or documents. This is essential for handling the vast amount of text data available in applications like search engines and social media analysis.
5. Advanced vectorization techniques, such as contextualized embeddings, significantly enhance model performance by providing rich, context-aware representations of words. This leads to better generalization and accuracy in NLP tasks.
6. Pre-trained models like BERT and GPT use vectorization to create embeddings that can be fine-tuned for various NLP tasks. This transfer learning approach saves time and resources by leveraging existing knowledge.

Traditional Vectorization Techniques in NLP (Frequency Based Technique)

The three traditional vectorization techniques:

a) Bag of Words (BoW),

b) Term Frequency-Inverse Document Frequency (TF-IDF),

c) Count Vectorizer

1. Bag of Words (BoW) :

The Bag of Words model represents text by converting it into a collection of words (or tokens) and their frequencies, disregarding grammar, word order, and context.

Each document is represented as a vector of word counts, with each element in the vector corresponding to the frequency of a specific word in the document.

*i)The **Bag of Words** model is a way to represent text as numbers.*

- *It ignores grammar and word order.*
- *It focuses only on the **presence** or **frequency** of words in a document.*
- *The "bag" is the **vocabulary** (all unique words from the dataset).*

ii) documents = [
"The cat sat on the mat.",
"The dog sat on the log.",
"Cats and dogs are pets."]

iii) Break each sentence into words:

- Doc1 → the, cat, sat, on, the, mat
- Doc2 → the, dog, sat, on, the, log
- Doc3 → cats, and, dogs, are, pets

iv) Build the vocabulary

List all **unique words** across all documents:

Vocabulary:

[the, cat, sat, on, mat, dog, log, cats, and, dogs, are, pets]

v) Frequency representation

Count how many times each word appears in each document.

	the	cat	sat	on	mat	dog	log	cats	and	dogs	are	pets
Doc1	2	1	1	1	1	0	0	0	0	0	0	0
Doc2	2	0	1	1	0	1	1	0	0	0	0	0
Doc3	0	0	0	0	0	0	0	1	1	1	1	1

```
[[0 0 1 0 0 0 0 1 1 0 1 2]
 [0 0 0 0 1 0 1 0 1 0 1 2]
 [1 1 0 1 0 1 0 0 0 1 0 0]]
['and' 'are' 'cat' 'cats' 'dog' 'dogs' 'log' 'mat' 'on' 'pets' 'sat' 'the']
```

Advantages of Bag of Words (BoW)

- Simple and easy to implement.
- Provides a clear and interpretable representation of text.

Disadvantages of Bag of Words (BoW)

- Ignores the order and context of words.
- Results in high-dimensional and sparse matrices.
- Fails to capture semantic meaning and relationships between words.

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

Term Frequency (TF):

- Measures how often a word appears in a document.
- A higher frequency suggests greater importance.
- If a term appears frequently in a document, it is likely relevant to the document's content.

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

Limitations of TF Alone:

- TF does not account for the global importance of a term across the entire corpus.
- Common words like "the" or "and" may have high TF scores but are not meaningful in distinguishing documents.

Inverse Document Frequency (IDF):

Reduces the weight of common words across multiple documents while increasing the weight of rare words. If a term appears in fewer documents, it is more likely to be meaningful and specific.

$$IDF(t, D) = \log \frac{\text{Total number of documents in corpus D}}{\text{Number of documents containing term t}}$$

- The logarithm is used to dampen the effect of very large or very small values, ensuring the IDF score scales appropriately.
- It also helps balance the impact of terms that appear in extremely few or extremely many documents.

Limitations of IDF Alone:

- IDF does not consider how often a term appears within a specific document.
- A term might be rare across the corpus (high IDF) but irrelevant in a specific document (low TF).

Problem: Converting Text into vectors with TF-IDF

Imagine we have a **corpus** (a collection of documents) with three documents. calculate the TF-IDF score for specific term word **"cat"** in these documents. Assume the Data is only Stemmed ,No lower case ,No stop word Removal.

1. **Document 1:** "The cat sat on the mat."
2. **Document 2:** "The dog played in the park."
3. **Document 3:** "Cats and dogs are great pets."

Step 1: Calculate Term Frequency (TF)

For Document 1:

- The word **"cat"** appears **1 time**.
- The total number of terms in Document 1 is **6** ("the", "cat", "sat", "on", "the", "mat").
- So, TF(cat,Document 1) = 1/6

For Document 2:

- The word **"cat"** does **not appear**.
- So, TF(cat,Document 2)=0.

For Document 3:

- The word **"cat"** appears **1 time** (as "cats").
- The total number of terms in Document 3 is **6** ("cats", "and", "dogs", "are", "great", "pets").
- So, TF(cat,Document 3)=1/6

Step 2: Calculate Inverse Document Frequency (IDF)

- **Total number of documents in the corpus (D):** 3
- **Number of documents containing the term "cat":** 2 (Document 1 and Document 3).

So, $IDF(cat, D) = log\frac{3}{2} \approx 0.176$

Step 3: Calculate TF-IDF

The TF-IDF score for **"cat"** is 0.029 in Document 1 and Document 3, and 0 in Document 2 that **reflects both the frequency of the term in the document (TF) and its rarity across the corpus (IDF).**

The TF-IDF score is the product of TF and IDF:

TF-IDF(t, d, D) = TF(t, d) × IDF (t, D)

For Document 1:

TF-IDF (cat, Document 1, D) = 0.167 × 0.176 ≈ 0.029

For Document 2:

TF-IDF(cat, Document 2, D) = 0 × 0.176 = 0

For Document 3:

TF-IDF (cat, Document 3, D) = 0.167 x 0.176 ≈ 0.029

Summary

- Word embeddings are numeric representations of words, and they help convert discrete units (words and sentences) to continuous mathematical objects (vectors).
- The Skip-gram model uses a neural network with a linear layer and SoftMax to learn word embeddings as a by-product of the "fake" word-association task.
- GloVe makes use of global statistics of word co-occurrence to train word embeddings efficiently.
- Doc2Vec and fastText learn document-level embeddings and word embeddings with subword information, respectively.
- t-SNE is used for visualize word embeddings in lower dimensional space.