In [1]:
```
import nltk
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer

# Download required NLTK data (only the first time)
nltk.download('punkt')
nltk.download('stopwords')
nltk.download('punkt_tab') # Download the punkt_tab data package

def nlp_preprocessing_pipeline(sentence):
    # Step 1: Tokenize
    tokens = word_tokenize(sentence)
    print("Original Tokens:", tokens)

    # Step 2: Remove Stopwords
    stop_words = set(stopwords.words('english'))
    filtered_tokens = [word for word in tokens if word.lower() not in stop_
    print("Tokens Without Stopwords:", filtered_tokens)

    # Step 3: Apply Stemming
    stemmer = PorterStemmer()
    stemmed_tokens = [stemmer.stem(word) for word in filtered_tokens]
    print("Stemmed Words:", stemmed_tokens)

# Example usage
sentence = "NLP techniques are used in virtual assistants like Alexa and Si
nlp_preprocessing_pipeline(sentence)
```

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt.zip.
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
[nltk_data] Downloading package punkt_tab to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt_tab.zip.

Original Tokens: ['NLP', 'techniques', 'are', 'used', 'in', 'virtual', 'as
sistants', 'like', 'Alexa', 'and', 'Siri', '.']
Tokens Without Stopwords: ['NLP', 'techniques', 'used', 'virtual', 'assist
ants', 'like', 'Alexa', 'Siri', '.']
Stemmed Words: ['nlp', 'techniqu', 'use', 'virtual', 'assist', 'like', 'al
exa', 'siri', '.']
```

Short Answer Questions

1 . Stemming vs. Lemmatization Stemming chops off word endings to reach a root form (not always a valid word).

e.g., running → run (PorterStemmer might output run or runn)

Lemmatization uses vocabulary and grammar to return a base or dictionary form (lemma).

e.g., running → run (linguistically correct base form)

Example with "running":

Stemming: running → run or runn

Lemmatization: running → run

*2. *Stopword Removal: Helpful vs. Harmful Helpful when: Focusing on core semantics (e.g., search engines, topic modeling).

e.g., In "The dog barked at the stranger", removing "the" helps isolate key content.

Harmful when: Stopwords carry important meaning (e.g., sentiment analysis, translation, question answering).

e.g., In "I am not happy", removing "not" changes sentiment entirely.

Type *Markdown* and LaTeX: α^2

In [2]:
```
import spacy

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

# Input sentence
sentence = "Barack Obama served as the 44th President of the United States

# Process sentence
doc = nlp(sentence)

# Print Named Entities
for ent in doc.ents:
    print(f"Entity Text: {ent.text}, Label: {ent.label_}, Start: {ent.start
```

```
Entity Text: Barack Obama, Label: PERSON, Start: 0, End: 12
Entity Text: 44th, Label: ORDINAL, Start: 27, End: 31
Entity Text: the United States, Label: GPE, Start: 45, End: 62
Entity Text: the Nobel Peace Prize, Label: WORK_OF_ART, Start: 71, End: 92
Entity Text: 2009, Label: DATE, Start: 96, End: 100
```

Short Answer Questions.

1. How does NER differ from POS tagging in NLP? Named Entity Recognition (NER) identifies real-world entities like names of people, places, organizations, dates, etc.

e.g., "Barack Obama" → PERSON, "2009" → DATE

Part-of-Speech (POS) Tagging assigns grammatical roles to words like noun, verb, adjective, etc.

e.g., "Obama" → NNP (proper noun), "served" → VBD (past tense verb)

-> NER focuses on semantic meaning, whereas POS tagging focuses on syntactic structure.

2. Two Real-World Applications of NER a. Financial News & Market Analysis

NER extracts company names, stock tickers, acquisition dates, CEO names, etc., to support algorithmic trading or news summarization.

Example: “Apple acquired Beats in 2014” → Apple (ORG), Beats (ORG), 2014 (DATE)

b. Search Engines and Question Answering

Helps improve relevance by identifying entities in queries and documents.

Example: "Who won the Nobel Prize in 2009?" → Recognizes Nobel Prize (WORK_OF_ART), 2009 (DATE)

Type *Markdown* and LaTeX: α^2

In [3]:
```python
import numpy as np

def scaled_dot_product_attention(Q, K, V):
    """
    Scaled Dot-Product Attention

    Args:
    Q (numpy.ndarray): Query matrix of shape (batch_size, seq_len, d)
    K (numpy.ndarray): Key matrix of shape (batch_size, seq_len, d)
    V (numpy.ndarray): Value matrix of shape (batch_size, seq_len, d)

    Returns:
    tuple: attention_weights (numpy.ndarray), output (numpy.ndarray)
    """
    # Step 1: Compute dot product of Q and Kᵀ
    attn_scores = np.dot(Q, K.T)

    # Step 2: Scale the result by √d (where d is the key dimension)
    d = K.shape[-1]  # The dimension of the key
    scaled_scores = attn_scores / np.sqrt(d)

    # Step 3: Apply softmax to get attention weights
    attn_weights = np.exp(scaled_scores) / np.sum(np.exp(scaled_scores), ax

    # Step 4: Multiply the attention weights by V to get the output
    output = np.dot(attn_weights, V)

    return attn_weights, output

# Test inputs
Q = np.array([[1, 0, 1, 0], [0, 1, 0, 1]])
K = np.array([[1, 0, 1, 0], [0, 1, 0, 1]])
V = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])

# Get attention weights and output
attn_weights, output = scaled_dot_product_attention(Q, K, V)

# Display results
print("Attention Weights:")
print(attn_weights)
print("\nOutput:")
print(output)
```

```
Attention Weights:
[[0.73105858 0.26894142]
 [0.26894142 0.73105858]]

Output:
[[2.07576569 3.07576569 4.07576569 5.07576569]
 [3.92423431 4.92423431 5.92423431 6.92423431]]
```

Short Answer Questions

1. Why do we divide the attention score by √d in the scaled dot-product attention formula?
   Dividing by √d (where d is the dimension of the key vector) ensures that the attention scores are not excessively large. As the dimension of the key vector increases, the dot

product tends to grow, which could lead to large gradients during backpropagation. Scaling by √d helps in stabilizing the learning process and prevents numerical instability.

2. How does self-attention help the model understand relationships between words in a sentence? Self-attention allows each word in a sentence to attend to every other word, thus capturing contextual relationships. For example, in the sentence "The cat sat on the mat," the word "cat" can attend to "sat" and "mat", helping the model understand

Type *Markdown* and LaTeX: α^2

```
In [4]: from transformers import pipeline

        # Load pre-trained sentiment analysis pipeline
        sentiment_analyzer = pipeline("sentiment-analysis")

        # Input sentence
        sentence = "Despite the high price, the performance of the new MacBook is o

        # Perform sentiment analysis
        result = sentiment_analyzer(sentence)

        # Display the result
        label = result[0]['label']
        confidence_score = result[0]['score']

        print(f"Sentiment: {label}")
        print(f"Confidence Score: {confidence_score:.4f}")
```

```
No model was supplied, defaulted to distilbert/distilbert-base-uncased-fin
etuned-sst-2-english and revision 714eb0f (https://huggingface.co/distilbe
rt/distilbert-base-uncased-finetuned-sst-2-english).
Using a pipeline without specifying a model name and revision in productio
n is not recommended.
/usr/local/lib/python3.11/dist-packages/huggingface_hub/utils/_auth.py:94:
UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings
tab (https://huggingface.co/settings/tokens), set it as secret in your Goo
gle Colab and restart your session.
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to acces
s public models or datasets.
  warnings.warn(

config.json:   0%|          | 0.00/629 [00:00<?, ?B/s]

Xet Storage is enabled for this repo, but the 'hf_xet' package is not inst
alled. Falling back to regular HTTP download. For better performance, inst
all the package with: `pip install huggingface_hub[hf_xet]` or `pip instal
l hf_xet`
WARNING:huggingface_hub.file_download:Xet Storage is enabled for this rep
o, but the 'hf_xet' package is not installed. Falling back to regular HTTP
download. For better performance, install the package with: `pip install h
uggingface_hub[hf_xet]` or `pip install hf_xet`

model.safetensors:   0%|          | 0.00/268M [00:00<?, ?B/s]

tokenizer_config.json:   0%|          | 0.00/48.0 [00:00<?, ?B/s]

vocab.txt:   0%|          | 0.00/232k [00:00<?, ?B/s]

Device set to use cuda:0

Sentiment: POSITIVE
Confidence Score: 0.9998
```

Short Answer Questions

1. What is the main architectural difference between BERT and GPT? Which uses an encoder and which uses a decoder? BERT (Bidirectional Encoder Representations from Transformers) is an encoder-only model. It is designed to capture context from both the left and right of a word (bidirectional context). BERT is primarily used for tasks like classification, question answering, and named entity recognition (NER).

   GPT (Generative Pre-trained Transformer) is a decoder-only model. It generates text in a left-to-right manner (unidirectional). GPT is primarily used for language generation tasks such as text completion and conversation generation.
2. Explain why using pre-trained models (like BERT or GPT) is beneficial for NLP applications instead of training from scratch. Using pre-trained models offers several benefits:

   Time and Resources: Training an NLP model from scratch requires enormous amounts of data and computational resources. Pre-trained models have already learned from massive datasets, which significantly reduces the time and resources needed for training.

   Generalization: Pre-trained models like BERT and GPT have been trained on diverse text data, allowing them to generalize better on various NLP tasks without needing task-specific training data.

   Transfer Learning: Pre-trained models provide a solid foundation that can be fine-tuned