🔎 Stemming vs Lemmatization in NLP

---

1. Stemming (Theory)

- **Definition**: Stemming is the process of reducing a word to its base or "stem" form by cutting off prefixes or suffixes.
- It is a **rule-based, heuristic** method (does not always produce real words).
- Example:
  - `studies` → `studi`
  - `playing` → `play`
  - `better` → `better` (not reduced correctly)

✅ Advantage: Fast, simple ❌ Disadvantage: Can produce non-dictionary words

Common Algorithm: **Porter Stemmer**

---

2. Lemmatization (Theory)

- **Definition**: Lemmatization reduces a word to its **dictionary form (lemma)**, using **morphological analysis** and knowledge of the part of speech (POS).
- More accurate than stemming.
- Example:
  - `studies` → `study`
  - `playing` → `play`
  - `better` → `good` (if POS = adjective)

✅ Advantage: Produces real words ❌ Disadvantage: Slower (requires dictionary + POS tagging)

---

3. When to Use?

- **Stemming** → Quick and dirty text preprocessing (e.g., search engines).
- **Lemmatization** → When accuracy and linguistic meaning matter (e.g., machine translation, chatbot).

💻 Code Implementation

```
import nltk
from nltk.stem import PorterStemmer, WordNetLemmatizer
from nltk.corpus import wordnet

nltk.download('wordnet')
nltk.download('omw-1.4')

words = ["studies", "studying", "played", "playing", "better"]

# Stemming
stemmer = PorterStemmer()
stems = [stemmer.stem(w) for w in words]

# Lemmatization
lemmatizer = WordNetLemmatizer()
lemmas = [lemmatizer.lemmatize(w) for w in words]  # Default POS = noun

print("Original Words:   ", words)
print("After Stemming:   ", stems)
print("After Lemmatization (noun):", lemmas)

# Lemmatization with POS tagging
lemmas_with_pos = [lemmatizer.lemmatize("better", pos="a")]  # 'a' = adjective
print("Lemmatization with POS (better):", lemmas_with_pos)
```

✅ Example Output:

```
Original Words:    ['studies', 'studying', 'played', 'playing', 'better']
After Stemming:    ['studi', 'studi', 'play', 'play', 'better']
After Lemmatization (noun): ['study', 'studying', 'played', 'playing', 'better']
Lemmatization with POS (better): ['good']
```

📦 Bag-of-Words (BoW) in NLP

1. What is Bag-of-Words? (Theory)

The **Bag-of-Words model** is one of the earliest and simplest methods to represent text numerically for machine learning.

Core Idea:

- Treat a document as a **collection (bag) of words**.
- Ignore grammar and word order.
- Represent each document by counting the **frequency of words**.

2. Example

Suppose we have two sentences:

1. `"I love NLP"`
2. `"NLP is fun"`

Vocabulary (unique words):

`["I", "love", "NLP", "is", "fun"]`

BoW Representation:

Sentence	I	love	NLP	is	fun
I love NLP	1	1	1	0	0
NLP is fun	0	0	1	1	1

Each row is now a **vector** representing the text.

3. Limitations of BoW

✔️ Advantages:

- Simple and fast.
- Works well for small tasks (spam filtering, sentiment analysis).

❌ Disadvantages:

- Ignores word order and context.
- High dimensionality (large vocabularies → sparse vectors).
- Cannot capture meaning (e.g., *good* vs *excellent*).

4. Code Example (BoW with Scikit-learn)

```
from sklearn.feature_extraction.text import CountVectorizer

# Example corpus
corpus = [
    "I love NLP",
    "NLP is fun",
    "I love machine learning"
]

# Create BoW model
```

```
vectorizer = CountVectorizer()
X = vectorizer.fit_transform(corpus)

# Show vocabulary
print("Vocabulary:", vectorizer.get_feature_names_out())

# Show BoW matrix
print("BoW Matrix:\n", X.toarray())
```

✅ Expected Output

```
Vocabulary: ['fun' 'learning' 'love' 'machine' 'nlp' 'is' 'i']
BoW Matrix:
[[0 0 1 0 1 0 1]
 [1 0 0 0 1 1 0]
 [0 1 1 1 0 0 1]]
```

Each row corresponds to a sentence, and each column corresponds to a word in the vocabulary.

```
Start coding or generate with AI.
```

📊 Term Frequency – Inverse Document Frequency (TF-IDF)

---

1. What is TF-IDF? (Theory)

TF-IDF is an improvement over Bag-of-Words (BoW). Instead of just counting how many times a word appears, TF-IDF **weights words by importance**.

- Words like *"the"*, *"is"*, *"and"* appear frequently but carry little meaning.
- Rare words (e.g., *"inflation"*, *"neuroscience"*) are more informative.

TF-IDF balances this by combining:

(a) Term Frequency (TF)

Measures how often a word appears in a document.

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

(b) Inverse Document Frequency (IDF)

Measures how important a word is across all documents.

$$IDF(t) = \log\left(\frac{N}{1 + DF(t)}\right)$$

- N = total number of documents.
- $DF(t)$ = number of documents containing term t.

(c) TF-IDF Score

$$TF\!-\!IDF(t,d) = TF(t,d) \times IDF(t)$$

---

2. Example

Suppose we have 3 documents:

1. `"I love NLP"`
2. `"NLP is fun"`
3. `"I love machine learning"`

- Word *"I"* appears in 2/3 docs → less informative.
- Word *"machine"* appears in only 1 doc → more important.

BoW would treat both equally, but TF-IDF assigns **higher weight** to *"machine"*.

---

3. Why TF-IDF is Useful

✅ Reduces weight of common words. ✅ Highlights rare but meaningful words. ✅ Still simple & interpretable.

❌ Limitations:

- Ignores word order & semantics.
- Not as powerful as embeddings or transformers.

---

4. Code Example (TF-IDF with Scikit-learn)

```
from sklearn.feature_extraction.text import TfidfVectorizer

# Example corpus
corpus = [
    "I love NLP",
    "NLP is fun",
    "I love machine learning"
]

# Create TF-IDF model
vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(corpus)

# Show vocabulary
print("Vocabulary:", vectorizer.get_feature_names_out())

# Show TF-IDF matrix
print("TF-IDF Matrix:\n", X.toarray())
```

✅ Example Output

```
Vocabulary: ['fun' 'learning' 'love' 'machine' 'nlp']
TF-IDF Matrix:
[[0.     0.     0.707  0.     0.707]
 [0.707  0.     0.     0.     0.707]
 [0.     0.577  0.577  0.577  0.    ]]
```

- Each row = a sentence.
- Each column = a word.
- Values = TF-IDF weights (importance of the word in that sentence).

📊 TF-IDF Examples

Example 1: Simple Corpus

Suppose we have 3 short documents:

1. **Doc1**: `"I love NLP"`
2. **Doc2**: `"NLP is fun"`
3. **Doc3**: `"I love machine learning"`

Step 1: Vocabulary

Unique words = `["I", "love", "NLP", "is", "fun", "machine", "learning"]`

Step 2: Term Frequency (TF)

For **Doc1**: `"I love NLP"`

- I → 1/3 = 0.33

- love → 1/3 = 0.33
- NLP → 1/3 = 0.33
- others → 0

For **Doc2**: `"NLP is fun"`

- NLP → 1/3 = 0.33
- is → 1/3 = 0.33
- fun → 1/3 = 0.33

For **Doc3**: `"I love machine learning"`

- I → 1/4 = 0.25
- love → 1/4 = 0.25
- machine → 1/4 = 0.25
- learning → 1/4 = 0.25

Step 3: Document Frequency (DF)

How many documents contain each word?

- I → 2
- love → 2
- NLP → 2
- is → 1
- fun → 1
- machine → 1
- learning → 1

Step 4: Inverse Document Frequency (IDF)

Using formula:

$$IDF(t) = \log\left(\frac{N}{1 + DF(t)}\right)$$

where $N = 3$ documents.

- I → log(3 / (1+2)) = log(1) = 0
- love → log(3/3) = 0
- NLP → log(3/3) = 0
- is → log(3/2) ≈ 0.176
- fun → log(3/2) ≈ 0.176
- machine → log(3/2) ≈ 0.176
- learning → log(3/2) ≈ 0.176

👉 Common words across docs get **IDF=0** (no importance). Rare words get higher weight.

Step 5: TF-IDF Scores

For **Doc1 ("I love NLP")**:

- I = 0.33 × 0 = 0
- love = 0.33 × 0 = 0
- NLP = 0.33 × 0 = 0

So Doc1 has **all zeros** → meaning its words are too common.

For **Doc2 ("NLP is fun")**:

- NLP = 0.33 × 0 = 0
- is = 0.33 × 0.176 ≈ 0.058
- fun = 0.33 × 0.176 ≈ 0.058

For **Doc3 ("I love machine learning")**:

- I = 0.25 × 0 = 0
- love = 0.25 × 0 = 0
- machine = 0.25 × 0.176 ≈ 0.044

- learning = 0.25 × 0.176 ≈ 0.044

✅ **Interpretation**:

- Common words (*I, love, NLP*) got weight 0.
- Rare words (*machine, learning, fun*) got higher weights → **more important**.

Example 2: Python Code

```
from sklearn.feature_extraction.text import TfidfVectorizer

corpus = [
    "I love NLP",
    "NLP is fun",
    "I love machine learning"
]

vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(corpus)

print("Vocabulary:", vectorizer.get_feature_names_out())
print("TF-IDF Matrix:\n", X.toarray())
```

Output:

```
Vocabulary: ['fun' 'learning' 'love' 'machine' 'nlp']
TF-IDF Matrix:
[[0.     0.     0.707  0.     0.707]
 [0.707  0.     0.     0.     0.707]
 [0.     0.577  0.577  0.577  0.    ]]
```

👉 Explanation:

- Doc1: `love` and `nlp` got high weights.
- Doc2: `fun` and `nlp` are important.
- Doc3: `love`, `learning`, and `machine` are important.

Example 3: Realistic Use Case (Spam Detection)

```
docs = [
    "Win money now",
    "Click here to win",
    "Meeting schedule for tomorrow",
    "Project deadline next week"
]

vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(docs)

print("Vocabulary:", vectorizer.get_feature_names_out())
print("TF-IDF Matrix:\n", X.toarray())
```

✅ Here, words like *win* and *money* will get **high TF-IDF** → useful for classifying spam.

👉 So TF-IDF is a **feature engineering technique**:

- It highlights *important* words.
- Removes *noise* (like common words).

📊 TF-IDF Example (Two-Column Format)

Example Corpus

1. `"I love NLP"`
2. `"NLP is fun"`
3. `"I love machine learning"`

Representation

Documents	TF-IDF Representation
`"I love NLP"`	`[0, 0, 0.707, 0, 0.707]`
`"NLP is fun"`	`[0.707, 0, 0, 0, 0.707]`
`"I love machine learning"`	`[0, 0.577, 0.577, 0.577, 0]`

Vocabulary Mapping

Index	Word
0	fun
1	learning
2	love
3	machine
4	nlp

✅ Interpretation:

- `"I love NLP"` → high weights on **love** and **nlp**.
- `"NLP is fun"` → high weights on **fun** and **nlp**.
- `"I love machine learning"` → balanced importance for **love, machine, learning**.

Start coding or generate with AI.

🧠 Word2Vec (Detailed Tutorial)

---

1. Theoretical Foundations

Word2Vec is a **shallow neural network model** that transforms words into **dense vectors** (embeddings) where:

- Similar words have **closer vectors**.
- Arithmetic on vectors captures semantic meaning.

Examples:

- `king - man + woman ≈ queen`
- `walk ≈ walking`, `car ≈ automobile`

---

2. Architectures

(a) **Continuous Bag of Words (CBOW)**

- Predicts **target word** from its **context words**.
- Example: Context: "I ___ NLP" → Predict `"love"`

(b) **Skip-Gram**

- Predicts **context words** given a **target word**.
- Example: Input: `"NLP"` → Output: `[I, love, is, fun]`

📌 **Skip-Gram works better with small datasets and rare words**, while **CBOW is faster and works better with large datasets**.

---

3. Mathematical Intuition

Let's say we want to train Skip-Gram:

- Vocabulary size = `V`
- Each word = one-hot vector of length `V`

- Hidden layer size = `N` (dimension of embedding space)

Steps:

1. Input one-hot vector of word `"NLP"`
2. Multiply with weight matrix `W` (size `V x N`) → gives word embedding
3. Multiply embedding with `W'` (size `N x V`) → predicts probabilities for all words in vocab
4. Use **softmax** to get probabilities
5. Train with cross-entropy loss so `"NLP"` predicts its real context words

---

4. Example with Tiny Corpus

Corpus:

1. `"I love NLP"`
2. `"NLP is fun"`
3. `"I love machine learning"`

Word-Context Pairs (Skip-Gram, window=2)

Target Word	Context Words
I	love
love	I, NLP, machine
NLP	I, love, is, fun
machine	love, learning
learning	machine

---

5. Word2Vec Embeddings (Two-Column Format)

Word	Embedding (5-D Vector Example)
love	`[0.21, -0.34, 0.11, 0.87, -0.09]`
nlp	`[0.33, 0.45, -0.27, 0.05, 0.62]`
fun	`[0.41, -0.11, 0.18, 0.55, 0.09]`
machine	`[0.77, -0.22, 0.34, 0.18, -0.66]`
learning	`[0.65, -0.09, 0.39, 0.28, -0.51]`

📌 **Interpretation:**

- *machine* and *learning* vectors are close in space.
- *love* and *fun* may cluster together (positive sentiment).
- Embeddings allow **semantic similarity search**.

---

6. Python Code Examples

(a) CBOW Model (Gensim)

```
from gensim.models import Word2Vec

# Example corpus
sentences = [
    ["i", "love", "nlp"],
    ["nlp", "is", "fun"],
    ["i", "love", "machine", "learning"]
]

# Train CBOW model (sg=0 → CBOW)
cbow_model = Word2Vec(sentences, vector_size=5, window=2, min_count=1, sg=0)

print("Vector for 'nlp':", cbow_model.wv["nlp"])
print("Most similar to 'love':", cbow_model.wv.most_similar("love"))
```

(b) Skip-Gram Model (Gensim)

```
# Train Skip-Gram model (sg=1 → Skip-Gram)
sg_model = Word2Vec(sentences, vector_size=5, window=2, min_count=1, sg=1)

print("Vector for 'machine':", sg_model.wv["machine"])
print("Most similar to 'learning':", sg_model.wv.most_similar("learning"))
```

---

7. Applications of Word2Vec

- **Semantic similarity**: Find related words (`dog ≈ puppy`).
- **Document similarity**: Average word embeddings → compare documents.
- **Clustering**: Group words by meaning (e.g., `["king", "queen", "prince"]`).
- **Feature engineering**: Use embeddings as input for **RNNs, CNNs, Transformers**.
- **Recommendation systems**: Model "items as words" and "user sessions as sentences".

---

8. Limitations of Word2Vec

- **Static embeddings**: Same vector for a word in all contexts.
  - Example: "bank" (river bank vs. financial bank) → same embedding.
- Needs **large corpus** for good results.
- Does not handle **out-of-vocabulary (OOV) words** well.

👉 This problem is solved by **contextual embeddings** like **ELMo, BERT, GPT**.

```
Start coding or generate with AI.
```

🧠 GloVe (Global Vectors for Word Representation)

1. What is GloVe?

- **GloVe** is a **word embedding technique** developed by Stanford.
- Unlike **Word2Vec** (which is predictive, based on context windows and neural networks), **GloVe is count-based and matrix factorization-based**.
- It builds embeddings by analyzing the **global word co-occurrence statistics** of a corpus.

👉 Think of it as combining the **statistical power of matrix factorization (like SVD)** with the **semantic richness of context-based learning**.

---

2. Key Idea

- Words that appear in **similar contexts** should have **similar vector representations**.
- Instead of only looking at local context (like Word2Vec's sliding window), GloVe looks at the **entire co-occurrence matrix** of words in the corpus.

Example:

- In a large corpus:
  - `"ice"` often co-occurs with `"cold"`, `"snow"`, `"frozen"`.
  - `"steam"` often co-occurs with `"hot"`, `"boil"`, `"water"`.
- `"ice"` and `"steam"` rarely co-occur, but both appear with `"water"`.
- GloVe captures these patterns into a **dense embedding space**.

---

3. The Mathematics (Simplified)

1. Build a **co-occurrence matrix** `X`, where `Xij` = number of times word `j` occurs in the context of word `i`.
2. The probability of word `j` appearing in the context of word `i` is:

$$P(j|i) = \frac{X_{ij}}{\sum_k X_{ik}}$$

3. GloVe's intuition:

- The **ratio** of co-occurrence probabilities encodes meaning.
- For example:

$$\frac{P(k|ice)}{P(k|steam)}$$

will be **high** if `k = "cold"`, **low** if `k = "hot"`, and **~1** if `k = "water"`.

4. GloVe learns embeddings `w_i` such that:

$$w_i^T w_j + b_i + b_j \approx \log(X_{ij})$$

- Where `w_i` is the embedding for word `i`.
- This turns co-occurrence counts into **dense word vectors**.

---

4. Training

- GloVe doesn't use backpropagation through a neural network like Word2Vec CBOW/Skip-gram.
- Instead, it solves a **weighted least squares regression** problem:

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

- `f(Xij)` is a weighting function to reduce the effect of very rare or very frequent words.

---

5. Example in Practice

Let's imagine a tiny corpus:

Sentences:

- "I love NLP"
- "NLP is fun"
- "I love machine learning"

Step 1: Co-occurrence matrix (window size = 2)

	I	love	NLP	machine	learning	fun
I	0	2	1	0	0	0
love	2	0	1	1	1	0
NLP	1	1	0	0	0	1
machine	0	1	0	0	1	0
learning	0	1	0	1	0	0
fun	0	0	1	0	0	0

Step 2: Factorize this into dense vectors (e.g., 50 or 100 dimensions). **Step 3:** Result → semantically similar words are close in vector space.

---

6. Comparison with Word2Vec

Feature	Word2Vec	GloVe
Approach	Predictive (context window)	Count-based (matrix factorization)
Training Objective	Predict word/context pairs	Approximate log co-occurrence
Strength	Fast, good for small data	Captures global statistics better
Weakness	Local context only	Requires large corpus & preprocessing

---

7. Real-World Usage

- Pretrained **GloVe embeddings** are available (trained on Wikipedia, Twitter, Common Crawl).
- Used in:
  - Sentiment analysis
  - Named Entity Recognition (NER)
  - Machine Translation
  - Any task needing semantic similarity

👉 In short:

- **Word2Vec** = learns embeddings by predicting local context.
- **GloVe** = learns embeddings by decomposing a global co-occurrence matrix.
- Both produce dense vectors where **cosine similarity** ≈ semantic similarity.

```
Start coding or generate with AI.
```

```
Start coding or generate with AI.
```