NLP

Author : Arnab Dana

▼ Resource

▼ Content

▼ 0. Road Map

Roadmap for Natural Language Processing (NLP) Series

1. Programming Language

- **Step 1**: Master Python, the essential language for NLP tasks.

2. Text Pre-processing

- **Part 1**: **Basic Text Pre-processing**
  - **Topics**: Tokenization, Lemmatization, Stemming, Stop Words Removal.
  - **Goal**: Clean and prepare text data for analysis.
- **Part 2**: **Intermediate Text Pre-processing**
  - **Topics**: Bag of Words (BoW), TF-IDF, Unigrams, Bigrams.
  - **Goal**: Convert text into vectors using basic vectorization techniques.
- **Part 3**: **Advanced Text Pre-processing**
  - **Topics**: Word Embeddings (Word2Vec, Average Word2Vec).
  - **Goal**: Achieve better vector representation by using advanced techniques for converting text into meaningful vectors.

3. Deep Learning for NLP

- **Step 1**: **Introduction to Deep Learning Models**
  - **Topics**: Recurrent Neural Networks (RNN), Long Short-Term Memory (LSTM), Gated Recurrent Unit (GRU).
  - **Goal**: Understand how these models handle sequential data and dependencies in text.
- **Step 2**: **Advanced Models**
  - **Topics**: Transformers, BERT, GPT.
  - **Goal**: Dive into state-of-the-art architectures for cutting-edge NLP tasks.

4. Application of Libraries

- **Machine Learning**:
  - **NLTK**: Use for text pre-processing and basic NLP tasks.
  - **SpaCy**: Explore for efficient NLP pipelines and advanced features.
- **Deep Learning**:
  - **TensorFlow or PyTorch**: Choose and master one for building complex NLP models.

Summary

- **Initial Focus**: Start with Python and basic text preprocessing techniques.
- **Intermediate**: Progress to advanced text preprocessing, focusing on vectorization techniques like Word2Vec.
- **Advanced**: Learn deep learning models such as RNNs, LSTMs, GRUs, and advanced models like Transformers and BERT.
- **Practical Application**: Utilize NLTK and SpaCy for machine learning tasks and TensorFlow or PyTorch for deep learning.

This roadmap provides a clear and structured path to mastering NLP, from foundational skills to advanced techniques.

chatgpt

Roadmap for Learning Natural Language Processing (NLP)

1. Introduction to NLP

- **Understand the Basics of NLP**:
    - What is NLP?
    - History and evolution of NLP
    - Applications of NLP in various domains (healthcare, finance, customer service, etc.)
- **Key NLP Tasks**:
    - Tokenization
    - Stopwords removal
    - Lemmatization and stemming
    - Part-of-Speech tagging
    - Named Entity Recognition (NER)
    - Sentiment analysis
    - Language modeling

2. Pre-requisites

- **Programming Languages**:
    - **Python**: Focus on libraries like NLTK, spaCy, and TensorFlow/PyTorch.
    - **R** (optional): Useful for statistical NLP.
- **Mathematics & Statistics**:
    - **Linear Algebra**: Vectors, matrices, eigenvalues, eigenvectors.
    - **Probability & Statistics**: Probability distributions, Bayes' theorem.
    - **Calculus**: Differentiation and integration basics.
    - **Information Theory**: Entropy, mutual information.

3. Text Preprocessing

- **Tokenization**:
    - Word-level tokenization
    - Sentence-level tokenization
    - Byte-pair encoding
- **Text Normalization**:
    - Lowercasing
    - Removing punctuation
    - Removing stopwords
- **Stemming and Lemmatization**:
    - Porter Stemmer, Snowball Stemmer
    - WordNet Lemmatizer

4. Feature Engineering

- **Bag of Words (BoW)**:
    - Creating vocabulary
    - Document-term matrix
- **TF-IDF (Term Frequency-Inverse Document Frequency)**:
    - Calculating term frequency
    - Understanding inverse document frequency
- **Word Embeddings**:
    - Word2Vec
    - GloVe
    - FastText
- **Advanced Embeddings**:
    - BERT (Bidirectional Encoder Representations from Transformers)
    - GPT (Generative Pre-trained Transformer)
    - ELMo (Embeddings from Language Models)

5. Statistical NLP

- **N-grams**:
    - Unigrams, Bigrams, Trigrams
    - Smoothing techniques (Laplace, Good-Turing)
- **Language Models**:
    - Markov models
    - Hidden Markov Models (HMM)
    - N-gram language models

6. Machine Learning for NLP

- **Supervised Learning**:
    - Classification (Naive Bayes, SVM, Logistic Regression)
    - Sequence labeling (CRFs, Bi-LSTM)
- **Unsupervised Learning**:
    - Clustering (K-means, DBSCAN)
    - Topic modeling (Latent Dirichlet Allocation)
- **Deep Learning**:
    - Recurrent Neural Networks (RNNs)
    - Long Short-Term Memory (LSTM)
    - GRU (Gated Recurrent Unit)
    - Transformers (Self-attention mechanism)

7. Advanced NLP Techniques

- **Transformers**:
    - Understanding the transformer architecture
    - Attention Mechanism
    - Multi-head attention
- **Transfer Learning in NLP**:

    - Fine-tuning pre-trained models (BERT, GPT, RoBERTa)
    - Zero-shot and few-shot learning
- **Sequence-to-Sequence Models**:
    - Encoder-Decoder architecture
    - Applications in machine translation, text summarization
- **Speech Recognition and Generation**:
    - ASR (Automatic Speech Recognition)
    - TTS (Text-to-Speech)
    - Transformer models in speech (Wav2Vec)

8. Applications of NLP

- **Text Classification**:
    - Spam detection
    - Sentiment analysis
- **Text Generation**:
    - Language translation
    - Chatbots and conversational agents
    - Summarization (extractive and abstractive)
- **Information Retrieval**:
    - Search engines
    - Question answering systems
- **Information Extraction**:
    - Named entity recognition (NER)
    - Relationship extraction

9. Tools and Libraries

- **NLP Libraries**:
    - **NLTK**: Classical NLP tasks
    - **spaCy**: Efficient and modern NLP library
    - **Stanford NLP**: Java-based NLP tools
- **Deep Learning Frameworks**:
    - **TensorFlow**: For building and training neural networks
    - **PyTorch**: More dynamic and flexible for research
    - **Hugging Face Transformers**: Pre-trained models and APIs for quick implementation
- **Other Tools**:
    - **Gensim**: Topic modeling and document similarity
    - **OpenAI GPT**: Access to generative pre-trained models

10. Research and Development

- **Current Trends**:
    - Ethical considerations in NLP (bias, fairness)
    - Low-resource languages
    - Multimodal NLP (text, image, and audio)

- **Staying Updated**:
  - Follow conferences like ACL, EMNLP, NAACL
  - Read papers from arXiv, Google Scholar
  - Participate in Kaggle competitions
- **Contribute to Open Source**:
  - Contributing to NLP libraries and frameworks
  - Writing blogs and tutorials on NLP topics

11. Real-World Projects

- **Build and Deploy**:
  - Sentiment analysis on real-time data
  - Chatbot using transformer models
  - Text summarization tool
  - Language translation application
- **Deploy Models**:
  - Deploying on cloud platforms (AWS, GCP, Azure)
  - Use of Docker for model deployment
  - Creating APIs using Flask/Django

12. Ethics and Bias in NLP

- **Understanding Bias**:
  - Sources of bias in language models
  - Mitigating bias in NLP systems
- **Ethical NLP**:
  - Data privacy concerns
  - Responsible AI in NLP applications

This roadmap provides a comprehensive path to mastering NLP, from the basics to advanced techniques and applications. As NLP is a rapidly evolving field, staying updated with the latest research and continuously experimenting with new tools and models is key to success.

▼ 1.

▼ 2. Basic Definition

Vector

In **Natural Language Processing (NLP)**, a **vector** is a numerical representation of text data that allows machine learning models to process and analyze language.

Since computers cannot directly understand words, NLP converts text into vectors (arrays of numbers) that capture semantic and syntactic information.

Types of Vectors in NLP:

1. **One-Hot Encoding**
   - Represents each word as a binary vector.
   - High-dimensional and sparse (mostly zeros).
2. **TF-IDF (Term Frequency-Inverse Document Frequency)**
   - Measures word importance in a document relative to a corpus.
   - Helps in filtering out common but less important words.
3. **Word Embeddings (Dense Vectors)**

- **Word2Vec**: Uses neural networks to capture word relationships (CBOW, Skip-gram).
- **GloVe**: Creates word vectors based on word co-occurrence statistics.
- **FastText**: Considers subword information to handle rare words better.

4. **Sentence and Document Embeddings**
   - **Doc2Vec**: Extends Word2Vec for documents.
   - **Transformers (BERT, GPT, etc.)**: Uses contextual embeddings to capture deep semantics.

Why Are Vectors Important in NLP?

- Enable mathematical operations on words.
- Improve performance in tasks like sentiment analysis, machine translation, and text generation.
- Capture semantic meanings and relationships between words.

Matrix of Size : V*D

- **V** = vocabulary size (# of total words)
- **D** = vector dimensionality

Word Embedding

Word embedding is a technique in **Natural Language Processing (NLP)** where words or phrases are represented as dense, continuous-valued numerical vectors. These vectors capture semantic relationships between words based on their context in a large corpus.

Key Features of Word Embeddings:

- Words with similar meanings have closer vector representations.
- Reduces the **dimensionality** of text data while preserving meaning.
- Encodes both **semantic** and **syntactic** relationships between words.

Popular Word Embedding Models:

1. **Word2Vec** (CBOW & Skip-gram) – Uses neural networks to generate word vectors.
2. **GloVe** – Based on word co-occurrence statistics.
3. **FastText** – Captures subword information, useful for rare words.
4. **BERT & Transformer-based Models** – Contextual word embeddings for deep understanding.

Word Analogy

Word analogy is the ability of word embeddings to capture relationships between words based on their meanings. It follows the pattern:

"King - Man + Woman = Queen"

This is possible because word embeddings represent words in a continuous vector space where relationships like gender, tense, and pluralization are preserved through vector arithmetic.

Example using Word2Vec:

- **Paris - France + Italy = Rome**
- **Walking - Walk + Running = Run**

Dimensionality

In the context of word embeddings, **dimensionality (D)** refers to the number of numerical values (features) used to represent a word in vector space.

Key points:

- Higher dimensions capture more meaning but increase computation.
- Typical sizes: **50, 100, 300** (Word2Vec, GloVe), **768, 1024** (BERT).
- Reducing dimensions (PCA, t-SNE) helps visualize embeddings.

Similarity / Distance measure techniques

Word similarity measures how closely related two words are based on their vector representations.

🔷 **Common similarity methods:**

- **Cosine Similarity**: Measures the cosine of the angle between two word vectors.
- **Euclidean Distance**: Measures the straight-line distance between vectors.
- **Jaccard Similarity**: Used for text sets (not vectors).

Example:

```
Similarity( "dog", "cat" ) > Similarity( "dog", "car" )
```

Because **dog** and **cat** are semantically closer than **dog** and **car**.

Bag of Words(BoW)

The **Bag of Words (BoW)** model is a simple way to represent text data numerically by treating a document as a collection of words, **ignoring grammar and word order**, but keeping word frequency.

How BoW Works?

1. **Create a vocabulary** – List all unique words in the dataset.
2. **Count occurrences** – Represent each document as a vector of word frequencies.

Example

Consider these two sentences:

1. "I love NLP and Machine Learning."
2. "Machine Learning and AI are amazing!"

Vocabulary (unique words)

`['I', 'love', 'NLP', 'and', 'Machine', 'Learning', 'AI', 'are', 'amazing']`

BoW Representation (word count in each sentence)

Words	I	love	NLP	and	Machine	Learning	AI
Sentence 1	1	1	1	1	1	1	0
Sentence 2	0	0	0	1	1	1	1

Each sentence is represented as a numerical vector.

Limitations of BoW

- **Ignores word order** – "I love NLP" and "NLP love I" have the same vector.
- **Does not capture meaning** – Synonyms and context are not considered.
- **Sparse representation** – Large vocabulary leads to high-dimensional vectors.

Alternatives to BoW

- **TF-IDF (Term Frequency-Inverse Document Frequency)** – Weighs important words more.
- **Word Embeddings (Word2Vec, GloVe, BERT, etc.)** – Capture context and meaning.

▼ 3. Coding

1. TF-IDE and t-SNE
2. GloVe
3. word2vec
4.

▼ 4. Tokenization in NLP

https://www.datacamp.com/blog/what-is-tokenization

Tokenization in Natural Language Processing (NLP) is the process of breaking down a text into smaller units, called tokens.

These tokens can be words, phrases, or even characters, depending on the level of tokenization.

Tokenization is a fundamental step in NLP tasks, as it prepares the text for further analysis, such as parsing, sentiment analysis, or machine learning.

- **Corpus** = Collection of texts (paragraphs or entire datasets)
- **Document** = A unit of text (like a sentence or paragraph)
- **Vocabulary** = Unique words in the corpus
- **Words** = Tokens or individual elements of the text

▼ 5. Stemming

Stemming in Natural Language Processing (NLP) is a technique used to reduce words to their root or base form. The goal is to remove prefixes and suffixes from words, so they can be analyzed as the same word, regardless of tense, number, or form. For example, the words "running," "runner," and "ran" can all be reduced to the root word "run."

Stemming helps improve text analysis by treating variations of a word as one, making processes like information retrieval, text classification, and sentiment analysis more efficient. Popular stemming algorithms include the **Porter Stemmer** and the **Lancaster Stemmer**.

▼ 6. Regex

Here is a table of common **regular expression (regex) symbols** with explanations and examples of how they work:

Symbol	Meaning	Example	Matches
`.`	Matches any single character except a newline.	`a.c`	Matches "abc", "axc", "a1c", etc.
`*`	Matches 0 or more occurrences of the preceding character or group.	`a.*`	Matches "a", "ab", "abc", "abcd", etc.
`+`	Matches 1 or more occurrences of the preceding character or group.	`a+`	Matches "a", "aa", "aaa", etc.
`?`	Matches 0 or 1 occurrence of the preceding character or group.	`a?b`	Matches "b" or "ab"
`[]`	Matches any one character from the enclosed characters.	`[aeiou]`	Matches "a", "e", "i", "o", or "u".
`^`	Anchors the match to the beginning of the string.	`^a`	Matches "a" if it's at the beginning of a string.
`$`	Anchors the match to the end of the string.	`ing$`	Matches words ending in "ing" (e.g., "running", "singing")
`()`	Groups expressions together for applying operators or capturing matches.	`(abc	def)`
`	`	Acts as a logical OR to match one of several expressions.	`a
`{n,m}`	Matches between **n** and **m** occurrences of the preceding character.	`a{2,4}`	Matches "aa", "aaa", or "aaaa".
`\d`	Matches any digit (equivalent to `[0-9]`).	`\d+`	Matches "1", "23", "456", etc.
`\D`	Matches any character that is **not** a digit.	`\D+`	Matches "a", "abc", "!" (not digits).
`\w`	Matches any word character (alphanumeric + underscore).	`\w+`	Matches "apple", "word1", "_underscore".

`\W`	Matches any non-word character (opposite of `\w`).	`\W+`	Matches "!", "#", "@", etc. (not letters or numbers).
`\s`	Matches any whitespace character (space, tab, newline).	`\s+`	Matches spaces, tabs, newlines.
`\S`	Matches any non-whitespace character.	`\S+`	Matches "abc", "123", "!" (not spaces).
`\\`	Escapes special characters to match them literally.	`\\.`	Matches the literal dot `.`.
`\b`	Matches a word boundary (e.g., space or punctuation).	`\bcat\b`	Matches "cat" as a whole word (not in "catalog").
`\B`	Matches a **non-word boundary**.	`\Bcat\B`	Matches "cat" only when it's **not** at the word boundary.
`()`	Groups expressions together, used for capturing matches or applying quantifiers.	`(abc	def)`

Examples:

- `.` **(dot)**:
  - Pattern: `a.c`
  - Matches: "abc", "axc", "a1c", etc.
- **(asterisk)**:
  - Pattern: `a.*`
  - Matches: "a", "ab", "abc", "abcd", etc.
- `[]` **(square brackets)**:
  - Pattern: `[aeiou]`
  - Matches: "a", "e", "i", "o", "u" (vowels).
- `\d` **(digit)**:
  - Pattern: `\d+`
  - Matches: "1", "23", "456", etc. (digits).

This table covers a variety of the most common regex symbols and their usage in regex patterns. It should help in understanding how these symbols function for pattern matching and text processing.

▼ 7. Parts of speech (POS)

In NLP (Natural Language Processing), parts of speech (POS) are categorized into different specific tags. These tags help to identify the role of each word in a sentence. The categories can vary depending on the tag set being used, but here is a more detailed breakdown of the commonly used POS categories based on the **Penn Treebank Tag Set**, which is widely used in NLP:

1. **Nouns**

- **NN**: Noun, singular or mass (e.g., `dog`, `car`, `water`)
- **NNS**: Noun, plural (e.g., `dogs`, `cars`)
- **NNP**: Proper noun, singular (e.g., `John`, `New York`)
- **NNPS**: Proper noun, plural (e.g., `Americas`, `Smiths`)

2. **Verbs**

- **VB**: Verb, base form (e.g., `run`, `eat`)
- **VBD**: Verb, past tense (e.g., `ran`, `ate`)
- **VBG**: Verb, gerund/present participle (e.g., `running`, `eating`)
- **VBN**: Verb, past participle (e.g., `run`, `eaten`)
- **VBP**: Verb, non-3rd person singular present (e.g., `run`, `eat`)

- **VBZ**: Verb, 3rd person singular present (e.g., `runs`, `eats`)

3. Adjectives

- **JJ**: Adjective, general (e.g., `big`, `blue`)
- **JJR**: Adjective, comparative (e.g., `bigger`, `more interesting`)
- **JJS**: Adjective, superlative (e.g., `biggest`, `most interesting`)

4. Adverbs

- **RB**: Adverb (e.g., `quickly`, `here`)
- **RBR**: Adverb, comparative (e.g., `more quickly`, `better`)
- **RBS**: Adverb, superlative (e.g., `most quickly`, `best`)

5. Pronouns

- **PRP**: Personal pronoun (e.g., `I`, `he`, `she`)
- **PRP$**: Possessive pronoun (e.g., `my`, `his`, `her`)
- **WP**: Wh-pronoun (e.g., `who`, `what`)
- **WP$**: Possessive wh-pronoun (e.g., `whose`)

6. Prepositions

- **IN**: Preposition or subordinating conjunction (e.g., `in`, `on`, `at`, `because`)

7. Conjunctions

- **CC**: Coordinating conjunction (e.g., `and`, `but`, `or`)
- **IN**: Subordinating conjunction (also used as a preposition, e.g., `if`, `because`, `although`)

8. Interjections

- **UH**: Interjection (e.g., `wow`, `oh`, `ouch`)

9. Determiners and Articles

- **DT**: Determiner (e.g., `the`, `a`, `some`)
- **PDT**: Predeterminer (e.g., `all`, `both`, `half`)
- **WDT**: Wh-determiner (e.g., `which`, `what`)

10. Particles

- **RP**: Particle (e.g., `up`, `off`, `out`)

11. Numerals

- **CD**: Cardinal number (e.g., `one`, `two`, `three`)
- **OD**: Ordinal number (e.g., `first`, `second`)

12. Possessive Endings

- **POS**: Possessive ending (e.g., `'s` as in `John's`)

13. Other

- **LS**: List item marker (e.g., `1.`, `a.`)
- **SYM**: Symbol (e.g., `+`, `$`, `&`)
- **TO**: Infinitive marker (e.g., `to` in "to run", "to eat")
- **FW**: Foreign word (e.g., `bonjour`, `hasta`)

14. Special Cases

- **X**: Other (e.g., words that don't fit into a standard POS category)

These tags are used to identify the syntactic role of words in a sentence and are helpful for various NLP tasks such as syntactic parsing, information extraction, and named entity recognition (NER). The specific tags can vary slightly depending on the tag set used (e.g., Universal Dependencies, Universal POS tags, etc.), but most of these categories are common across different NLP systems.

▼ 8. Text Into Vector(vectorizer)

1. One hot encoding (Bashed on count and frequency)
2. BOW (Bashed on count and frequency)
3. TF-IDF (Bashed on count and frequency)
4. Word2Vec (Bashed on Neural Network)
5. AvgWord2Vec

One hot encoding

A technique to represent categorical data as binary vectors. Each category is mapped to a unique vector with one "1" and the rest "0".

Example:

Categories: `["Red", "Blue", "Green"]`

Color	One-Hot Encoding
Red	[1, 0, 0]
Blue	[0, 1, 0]
Green	[0, 0, 1]

BOW

A text representation method that converts text into a numerical format by counting word occurrences, ignoring grammar and order.

Example:

Text: *"Machine learning is amazing. Learning is fun."*

BoW Representation:

Word	Count
machine	1
learning	2
is	2
amazing	1
fun	1

TF-IDF

A numerical statistic that reflects the importance of a word in a document relative to a collection of documents (corpus). It helps reduce the impact of common words.

```
TF (Term Frequency) = (Number of times a word appears in a document) / (Total words in the document)

IDF (Inverse Document Frequency) = log(Total documents / Documents containing the word)

TF-IDF Score = TF * IDF
```

Example

Corpus:

1. "Machine learning is amazing."
2. "Deep learning is powerful."

Step 1: Calculate TF (Term Frequency)

For the word **"learning"** in Document 1 and 2:

```
For the word "learning" in Document 1:
TF = (Occurrences of "learning") / (Total words in Document 1)
   = 1 / 4
   = 0.25

For the word "learning" in Document 2:
TF = 1 / 4 = 0.25
```

Step 2: Calculate IDF (Inverse Document Frequency)

```
IDF = log(Total documents / Documents containing the word)
    = log(2 / 2)
    = log(1)
    = 0
```

Step 3: Calculate TF-IDF Score

```
TF-IDF for "learning" = TF * IDF
                      = 0.25 * 0
                      = 0
```

Since "learning" appears in both documents, its IDF is 0, meaning it is not a distinguishing word. Rare words will have a higher TF-IDF score.

Let's calculate the **TF-IDF score** for the word **"amazing"** in **Document 1**.

Step 1: Calculate TF (Term Frequency)

For **"amazing"** in **Document 1**:

```
For "amazing" in Document 1:
TF = (Occurrences of "amazing") / (Total words in Document 1)
   = 1 / 4
   = 0.25
For "amazing" in Document 2:
TF = 0 / 4
   = 0
```

Step 2: Calculate IDF (Inverse Document Frequency)

```
IDF = log(Total documents / Documents containing "amazing")
    = log(2 / 1)
    = log(2)
    ≈ 0.693
```

Step 3: Calculate TF-IDF Score

For **"amazing"** in **Document 1**:

```
TF-IDF = TF * IDF
       = 0.25 * 0.693
       ≈ 0.173
```

For **"amazing"** in **Document 2**:

```
TF-IDF = 0 * 0.693
       = 0
```

Final Result:

Word	Document 1 (TF-IDF)	Document 2 (TF-IDF)
amazing	**0.173**	**0**

Since **"amazing"** appears in only one document, it has a **higher TF-IDF score**, indicating it is more important in **Document 1**.

Word2Vec

A neural network-based technique that represents words as dense vectors in a continuous vector space, capturing semantic relationships.

Example:

Words like **"king"**, **"queen"**, and **"royalty"** will have similar vector representations.

It allows operations like:

📌 **king - man + woman ≈ queen**

Two main models:

- **CBOW (Continuous Bag of Words):** Predicts a word from its surrounding words.
  - Use for small dataset
- **Skip-gram:** Predicts surrounding words from a given word.
  - Use for large dataset

How Word2Vec Converts Words into Vectors (Step-by-Step)

Word2Vec is a neural network-based model that learns to represent words as continuous vectors. Below is a step-by-step explanation of how it converts words into numerical vectors using the **Skip-gram model** (one of the two approaches, the other being CBOW).

Step 1: Prepare the Training Data

Let's take a simple example sentence:

📌 *"I love machine learning."*

Create a Vocabulary

The unique words in the sentence are:

🔷 I, love, machine, learning

Let's assign indices:

- "I" → 0
- "love" → 1
- "machine" → 2
- "learning" → 3

Step 2: Create Training Pairs (Skip-gram Model)

In the Skip-gram model, we predict context words given a target word.

For example, if we use a **window size of 1**, the training pairs are:

Target Word	Context Word
"I"	"love"
"love"	"I"
"love"	"machine"

"machine"	"love"
"machine"	"learning"
"learning"	"machine"

Step 3: One-Hot Encoding of Words

Each word is converted into a one-hot encoded vector (size = vocabulary size).

Word	One-Hot Vector
"I"	[1, 0, 0, 0]
"love"	[0, 1, 0, 0]
"machine"	[0, 0, 1, 0]
"learning"	[0, 0, 0, 1]

Step 4: Initialize Random Weights for Word Vectors

We create a weight matrix of size **V × N**, where:

- **V** = vocabulary size (4 in this case)
- **N** = embedding size (let's assume 3)

Example random weight matrix:

$$W = \begin{bmatrix} 0.2 & 0.4 & 0.6 \\ 0.1 & 0.8 & 0.3 \\ 0.5 & 0.9 & 0.2 \\ 0.3 & 0.7 & 0.5 \end{bmatrix}$$

Each row corresponds to a word vector in the embedding space.

Step 5: Forward Propagation

When we input a one-hot encoded vector, it is multiplied with the weight matrix to get the word embedding.

For example, for the word **"love"** [0, 1, 0, 0]:

Embedding=[0,1,0,0]×W=[0.1,0.8,0.3,]

So, **"love"** is represented as a dense vector: **[0.1, 0.8, 0.3]**.

Step 6: Train the Model Using Backpropagation

- The model uses a **softmax layer** to predict the probability of context words.
- It adjusts the weights using **gradient descent** to minimize prediction error.

Step 7: Extract Word Vectors

After training, words with similar meanings will have similar vector representations.

For example, final learned vectors might look like:

Word	Learned Vector
"I"	[0.21, 0.43, 0.65]
"love"	[0.15, 0.87, 0.29]
"machine"	[0.52, 0.92, 0.18]
"learning"	[0.31, 0.74, 0.55]

AvgWord2Vec

Advantages and Disadvantages

Technique	Advantages	Disadvantages
One-Hot Encoding	- Simple and easy to implement- Works well for small vocabularies	- High dimensionality for large vocabularies (curse of dimensionality)- No semantic relationships between words
Bag of Words (BOW)	- Simple and effective for text classification- Works well for traditional NLP tasks	- Ignores word order and meaning- Sparse representation and high dimensionality
TF-IDF	- Reduces the impact of common words (stopwords)- Enhances the importance of rare but relevant words	- Still lacks semantic understanding- High dimensionality remains a challenge
Word2Vec	- Captures semantic relationships between words- Dense and lower-dimensional representation	- Requires large data and training- Doesn't handle out-of-vocabulary (OOV) words well
AvgWord2Vec	- Reduces dimensionality by averaging word vectors- Preserves some semantic meaning in sentences	- Loses word order information- May not capture complex sentence structures effectively

There's no single "best" vectorizer in NLP — the choice depends on your **task**, **data size**, and whether you prioritize **performance** or **interpretability**. Here's a comparison of the most commonly used vectorizers:

Vectorizer	Type	Strengths	When to Use
CountVectorizer	Sparse, bag-of-words	Simple, fast, interpretable	Small datasets, classical ML models (e.g., Naive Bayes, SVM)
TF-IDF Vectorizer	Sparse, weighted BOW	Reduces impact of common words, better than plain counts	Document classification, information retrieval
Word2Vec (Gensim)	Dense, embedding	Captures semantic relationships, efficient	Text similarity, sentiment analysis, clustering
GloVe	Dense, pre-trained	Global context, pretrained on large corpora	When you want word semantics without training from scratch
FastText	Dense, subword aware	Handles out-of-vocabulary (OOV) words well	Morphologically rich languages, noisy text
BERT (Transformers)	Dense, contextual	State-of-the-art contextual embeddings, understands word meaning deeply	Advanced NLP tasks like Q&A, summarization, NER

Best for Each Category:

- **Classical ML Tasks:** `TF-IDF Vectorizer`
- **Lightweight word embeddings:** `Word2Vec` or `GloVe`
- **Contextual understanding (deep NLP):** `BERT` or other Transformer-based models (e.g., RoBERTa, DistilBERT)

Want a recommendation tailored to your specific use case?

▼ 9. Gram, Cosine Similarity

Gram

Unigram: A single word (e.g., "Hello")

Bigram: A sequence of two words (e.g., "Machine Learning")

Trigram: A sequence of three words (e.g., "Artificial Intelligence Model")

Cosine Similarity

Cosine Similarity is a measure of similarity between two vectors based on the **cosine of the angle** between them.

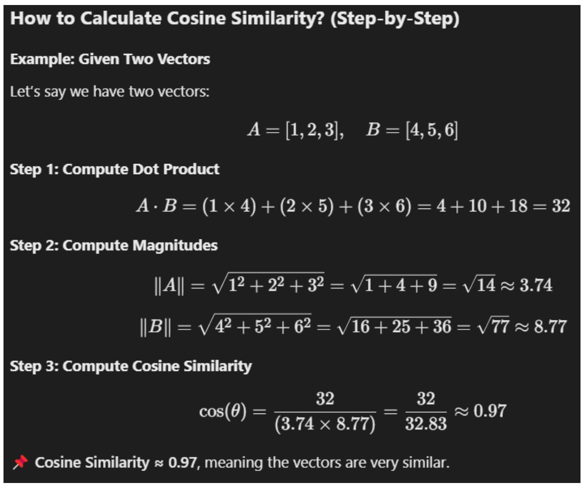
$$\text{Cosine Similarity} = \cos(\theta) = \frac{A \cdot B}{\|A\| \times \|B\|}$$

Where:

- A·B = Dot product of vectors **A** and **B**
- ⫽ A ⫽ = Magnitude (norm) of vector **A**
- ⫽ B ⫽ = Magnitude (norm) of vector **B**
- θ = Angle between the two vectors

🔷 **Value Range:**

- **1** → Identical vectors (same direction)
- **0** → Completely different vectors (orthogonal)
- **1** → Opposite direction (only in signed spaces)



How to Calculate Cosine Similarity? (Step-by-Step)

Example: Given Two Vectors

Let's say we have two vectors:

$$A = [1, 2, 3], \quad B = [4, 5, 6]$$

Step 1: Compute Dot Product

$$A \cdot B = (1 \times 4) + (2 \times 5) + (3 \times 6) = 4 + 10 + 18 = 32$$

Step 2: Compute Magnitudes

$$\|A\| = \sqrt{1^2 + 2^2 + 3^2} = \sqrt{1 + 4 + 9} = \sqrt{14} \approx 3.74$$

$$\|B\| = \sqrt{4^2 + 5^2 + 6^2} = \sqrt{16 + 25 + 36} = \sqrt{77} \approx 8.77$$

Step 3: Compute Cosine Similarity

$$\cos(\theta) = \frac{32}{(3.74 \times 8.77)} = \frac{32}{32.83} \approx 0.97$$

📌 **Cosine Similarity ≈ 0.97**, meaning the vectors are very similar.

Cosine Distance (Dissimilarity Measure)

To compute **Cosine Distance**, subtract Cosine Similarity from 1:

Cosine Distance=1−Cosine Similarity

For our example:

Cosine Distance=1−0.97=0.03

▼ 10. Vectorizer > **Word2Vec> Continuous bag-of-words (CBOW)**

https://blog.marketmuse.com/glossary/continuous-bag-of-words-definition/#:~:text=Continuous Bag of Words (CBOW,at%20Google%20in%202013.

Continuous Bag of Words (CBOW) is a neural network-based model used for learning word embeddings, which are dense vector representations of words that capture their semantic and syntactic properties.

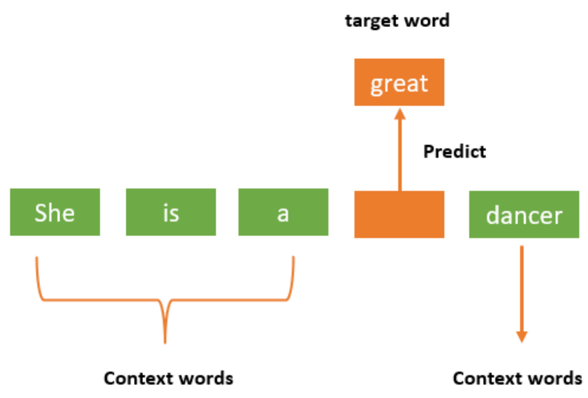
CBOW is a part of the Word2Vec family of models, developed by Mikolov et al. at Google in 2013.

The main idea behind CBOW is that the model learns to predict the target word based on the average of the context words' embeddings.

The CBOW model predicts the target word (center word) given a fixed-size window of context words surrounding it.

The word pairings would appear like this **([she, a], is), ([is, great], a) ([a, dancer], great)** having window size=2.

It does this by using a neural network with a single hidden layer to learn the weights that map the context words to the target word.



Here's a high-level overview of the CBOW model:

- **Input:** The input to the model is a fixed-size window of context words, typically represented as one-hot encoded vectors.
- **Embedding layer:** This layer maps the one-hot encoded input vectors to their corresponding embeddings, which are dense, low-dimensional vectors.
- **Average:** The model computes the average of the context word embeddings to create a single vector representing the combined context.
- **Hidden layer**: The averaged embedding vector is passed through a single hidden layer, which performs a non-linear transformation using an activation function, such as tanh or ReLU.
- **Output layer:** The output layer is a linear layer that maps the hidden layer's output to the target word embedding. The model uses softmax activation to output the probability distribution over the entire vocabulary.
- **Training:** The model is trained using stochastic gradient descent (SGD) to minimize the difference between the predicted and the actual target word embeddings.

Once the CBOW model is trained, it can generate embeddings for individual words that capture their semantic and syntactic relationships.

▼ 10. Vectorizer > **Word2Vec> Skip-gram Model** **

https://www.scaler.com/topics/nlp/skip-gram-model/https://www.geeksforgeeks.org/implement-your-own-word2vecskip-gram-model-in-python/

▼ 10.

▼ 10.