Vectorization in Natural Language Processing

Natural Language Processing (NLP) is a pivotal field in artificial intelligence concerned with the interaction between computers and human language. At its core, NLP seeks to enable machines to read, understand, and derive meaning from textual data. However, machine learning algorithms are mathematical constructs that require numerical input. The process of converting unstructured text into a structured, numerical format that algorithms can process is known as **vectorization** or **embedding**. This document provides a detailed exploration of fundamental and advanced vectorization techniques, complete with mathematical foundations, examples, and their inherent limitations.

Part 1: Foundational Concepts (Prerequisites)

Before diving into the specific techniques of vectorization, it's essential to understand the fundamental concepts that form the bedrock of text processing and representation in NLP.

1.1. Why is Vectorization Necessary?

Machine learning and deep learning models are mathematical functions that operate on numbers. They cannot directly process raw text. Therefore, to perform any task like sentiment analysis, machine translation, or text classification, we must first find a way to convert words, sentences, and documents into numerical vectors. This conversion process is **vectorization**. An effective vectorization scheme should not only represent the text numerically but also, ideally, capture its underlying meaning and context.

1.2. Essential NLP Terminology

Corpus (plural: Corpora)

A corpus is a large, structured collection of text documents. It serves as the dataset for training and evaluating NLP models. A corpus could be a collection of books, a set of all Wikipedia articles, or a database of customer reviews.

Document

A single piece of text within a corpus. The definition of a "document" is context-dependent. It could be a full book, a news article, a single sentence, or even a tweet.

Tokenization

The process of breaking down a stream of text (a document) into its fundamental constituent parts, called tokens. This is a crucial first step in any NLP pipeline.

Tokens

The output of tokenization. Most commonly, tokens are words. For example, the sentence "The cat sat." can be tokenized into three tokens: "The", "cat", and "sat". Punctuation can also be treated as a token.

Vocabulary

The set of all unique tokens present in a corpus. The size of the vocabulary, denoted as $|V|$, is a critical parameter that determines the dimensionality of many vectorization techniques.

1.3. Measuring Similarity: Cosine Similarity

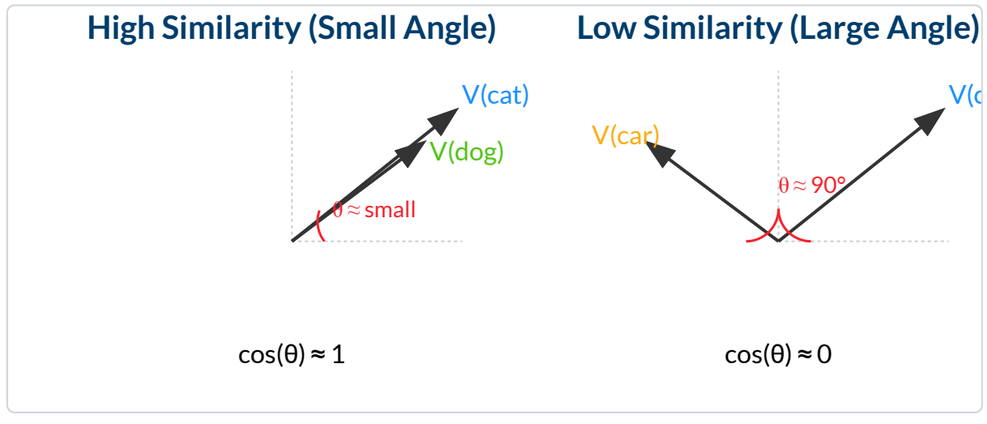
> **Concept:** Once words or documents are represented as vectors, we need a way to measure how similar they are. Cosine similarity is the most common metric for this. Instead of measuring the Euclidean distance between vectors, it measures the cosine of the angle between them. This captures the orientation of the vectors, not their magnitude.

In NLP, this is particularly useful because it allows us to determine if two words or documents share a similar context or meaning, regardless of their raw frequency counts. A smaller angle between two vectors implies higher cosine similarity.

- A cosine similarity of **1** means the vectors point in the exact same direction (the angle is 0°), indicating maximum similarity.
- A cosine similarity of **0** means the vectors are orthogonal (the angle is 90°), indicating no similarity or that they are unrelated.
- A cosine similarity of **-1** means the vectors point in opposite directions (the angle is 180°), indicating they are "opposites".

$$\text{Cosine Similarity}(\mathbf{A}, \mathbf{B}) = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \sqrt{\sum_{i=1}^{n} B_i^2}}$$

For example, in a well-trained embedding space, the vector for `cat` and `dog` would have a high cosine similarity, while the vectors for `cat` and `car` would have a very low one.



Part 2: Basic Word Vectorization Techniques

These initial approaches are primarily based on word frequencies and distributions across a corpus. While simpler than neural methods, they are highly interpretable and serve as powerful baselines for many NLP tasks like text classification and topic modeling.

2.1. One-Hot Encoding

> **Concept:** One-Hot Encoding (OHE) is the most straightforward method of vectorization. It represents each unique word in a corpus with a unique binary vector. This vector has a dimension equal to the size of the vocabulary. For any given word, the vector contains a single '1' at the index corresponding to that word, and '0's in all other positions.

Key Concepts

- **Vocabulary:** The set of all unique words present in the text corpus, denoted as V. The size of the vocabulary is $|V|$.
- **Vector Representation:** If a word w_i is the i-th word in the vocabulary, its one-hot vector representation v_{w_i} is a $|V|$-dimensional vector where the i-th element is 1 and all other elements are 0.
- **Orthogonality:** A key mathematical property is that the vectors for any two distinct words are orthogonal. The dot product of their vectors is always zero ($v_{w_i} \cdot v_{w_j} = 0$ for $i \neq j$). This mathematically implies a lack of any shared features or similarity.

Example:

Consider the sentence: `"The cat sat on the mat"` .

1. First, we build the vocabulary: $V = \{\text{The}, \text{cat}, \text{sat}, \text{on}, \text{mat}\}$. The size is $|V| = 5$.
2. Each word is assigned a unique index: The:0, cat:1, sat:2, on:3, mat:4.
3. The one-hot vectors are 5-dimensional:

```
The: [1, 0, 0, 0, 0]
cat: [0, 1, 0, 0, 0]
sat: [0, 0, 1, 0, 0]
on : [0, 0, 0, 1, 0]
mat: [0, 0, 0, 0, 1]
```

Drawbacks of One-Hot Encoding

- **High Dimensionality & Sparsity:** For a typical corpus, the vocabulary can contain tens of thousands of words. This leads to extremely high-dimensional and sparse (mostly zero) vectors, which are computationally and memory-inefficient.
- **No Semantic Meaning:** OHE fails to capture any semantic relationship between words. The vector for "cat" is equidistant from "dog" and "car", providing no useful information about their relatedness.
- **Out-of-Vocabulary (OOV) Problem:** If a new word appears that was not in the original vocabulary, it cannot be represented.

2.2. Count Vectorization (Bag-of-Words)

> **Concept:** The Count Vectorization model, also known as Bag-of-Words (BoW), moves from representing single words to representing entire documents. It describes a document by the frequency of each word from the vocabulary that appears within it. This model disregards grammar and word order, treating the text as an unordered "bag" of words.

Key Concepts & Mathematics

- **Document-Term Matrix:** This method is often visualized as a matrix M of size $D \times V$, where D is the number of documents and V is the vocabulary size. The entry M_{ij} contains the count of term j in document i.
- **Vector Representation:** Each row of the Document-Term Matrix is a vector representing the corresponding document.

Example:

Consider the corpus:

- **Doc 1:** `"The cat sat on the mat."`
- **Doc 2:** `"The dog sat on the log."`

1. The vocabulary is: $V = \{\text{The}, \text{cat}, \text{sat}, \text{on}, \text{mat}, \text{dog}, \text{log}\}$, with $|V| = 7$.
2. The document vectors are constructed based on word counts:

```
Vocabulary: [The, cat, sat, on, mat, dog, log]

Doc 1:      [ 2,   1,   1,   1,   1,   0,   0]
Doc 2:      [ 2,   0,   1,   1,   0,   1,   1]
```

The vector for Doc 1 indicates that "The" appears twice, "cat" once, "sat" once, etc., while "dog" and "log" do not appear at all.

Drawbacks of Count Vectorization

- **Loses Word Order:** By treating text as a "bag," all information about syntax and word order is lost. "The cat sat on the dog" and "The dog sat on the cat" would have identical vector representations.
- **Bias Towards Frequent Words:** Common words like "the", "a", "is" (stop words) often dominate the counts, overshadowing more meaningful terms. This often requires a pre-processing step to remove them.
- **No Semantic Understanding:** Like OHE, this model does not understand that "cat" and "dog" are more similar than "cat" and "log".
- **Sparsity and High Dimensionality:** It still produces high-dimensional and sparse vectors, though for documents rather than individual words.

2.3. TF-IDF Vectorization

Concept: TF-IDF (Term Frequency-Inverse Document Frequency) is a refinement of Count Vectorization. It evaluates how important a word is to a document within a corpus. The core idea is to assign each word a weight that is high when it appears frequently in a document but rarely across the entire corpus, and low when it is common throughout the corpus.

Key Concepts & Mathematics

- **Term Frequency (TF):** Measures how often a term t appears in a document d. To prevent a bias towards longer documents, this is often normalized. A common normalization is dividing by the total number of terms in the document.

$$TF(t, d) = \frac{\text{count of } t \text{ in } d}{\text{total number of terms in } d}$$

- **Inverse Document Frequency (IDF):** Measures the "informativeness" of a term. It is a logarithmic scaling of the ratio of total documents to the number of documents containing the term.

$$IDF(t, D) = \log\left(\frac{N}{df_t}\right)$$

Where N is the total number of documents in the corpus D, and df_t is the document frequency of term t. To avoid division by zero for new words and to prevent the IDF of a word appearing in all documents from becoming zero, a smoothed version is often used:

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

- **TF-IDF Score:** The final score for a term in a document is the product of its TF and IDF scores.

$$w_{t,d} = TF(t,d) \times IDF(t,D)$$

Example:

Using the same corpus as before: $D = \{\text{Doc 1}, \text{Doc 2}\}$, so $N = 2$.

- **Doc 1:** `"The cat sat on the mat"` (6 terms)

Let's calculate the TF-IDF score for **"cat"** and **"the"** in **Doc 1** (using the standard, non-smoothed IDF for simplicity).

- For the term **"cat"**:
  - $TF(\text{"cat"}, \text{Doc 1}) = 1/6 \approx 0.167$
  - "cat" appears in 1 document ($df_{\text{cat}} = 1$).
  - $IDF(\text{"cat"}, D) = \log(2/1) \approx 0.693$
  - $w_{\text{cat, Doc1}} = 0.167 \times 0.693 \approx 0.116$
- For the term **"the"**:
  - $TF(\text{"the"}, \text{Doc 1}) = 2/6 \approx 0.333$
  - "the" appears in 2 documents ($df_{\text{the}} = 2$).
  - $IDF(\text{"the"}, D) = \log(2/2) = \log(1) = 0$
  - $w_{\text{the, Doc1}} = 0.333 \times 0 = 0$

The score for "the" is zero, effectively filtering it out as an unimportant common word, while "cat" receives a meaningful score.

Drawbacks of TF-IDF

- **Still Ignores Word Order:** Like Count Vectorization, TF-IDF is a bag-of-words model and does not capture syntactic relationships.
- **Fails to Capture Semantics:** It cannot recognize synonyms or related concepts. "Car" and "automobile" are treated as completely different terms. It also struggles with polysemy (words with multiple meanings).
- **Sparsity:** The resulting document vectors are still high-dimensional and sparse, although the values are now weighted real numbers instead of just integer counts.

Part 3: Neural Network Based Vectorization (Embedding)

To overcome the limitations of count-based methods, particularly their inability to capture semantic meaning, neural network-based techniques were developed. The world of word embeddings is rich and constantly evolving, with other notable models like **GloVe** (which uses global matrix factorization) and **FastText** (which learns embeddings for sub-word units), as well as powerful contextual models like **BERT** and **GPT**. However, to understand the core principles of predictive embeddings, we will focus on the foundational and highly influential **Word2Vec** model.

3.1. Word2Vec

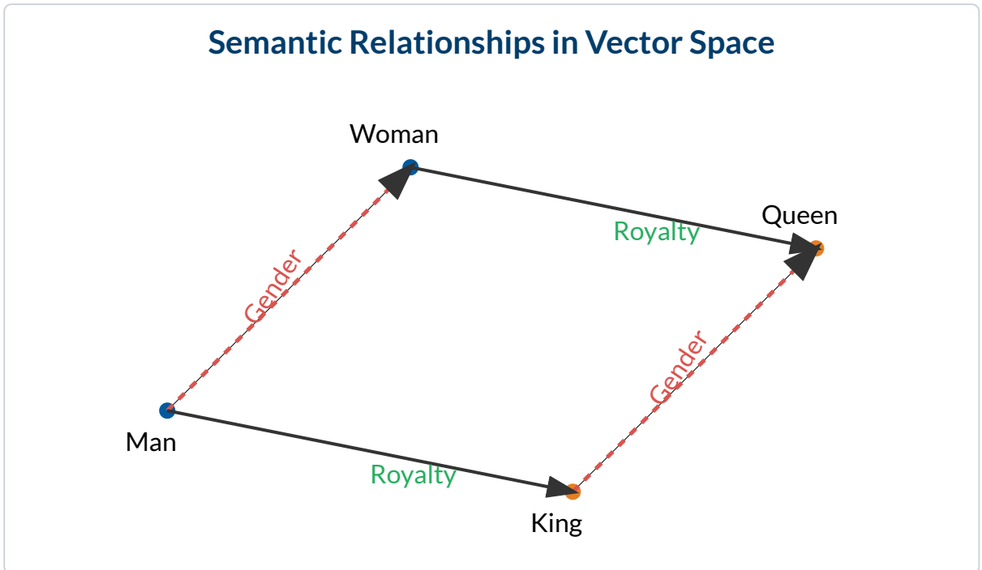
> **Concept:** Word2Vec, developed by a team at Google led by Tomas Mikolov, is a predictive model that learns word embeddings. It is not a single algorithm but a family of models. The core idea is based on the **distributional hypothesis**: "a word is characterized by the company it keeps." By training a simple neural network to predict a word from its neighbors (or vice versa), the model learns dense vector representations in the hidden layer as a byproduct.

3.1.1. Capturing Semantic Similarity: The Power of Vector Arithmetic

The most remarkable property of Word2Vec embeddings is their ability to capture complex semantic relationships. These relationships are represented as linear translations in the vector space. The classic example is the analogy "king is to queen as man is to woman." In the vector space, this translates to:

$$V_{\text{king}} - V_{\text{man}} + V_{\text{woman}} \approx V_{\text{queen}}$$

This means the vector capturing the "male-to-female" relationship ($V_{\text{woman}} - V_{\text{man}}$) is approximately the same as the vector capturing the "king-to-queen" relationship ($V_{\text{queen}} - V_{\text{king}}$). This allows for reasoning through vector arithmetic, a feat impossible with count-based methods. Other relationships, such as country-capital (e.g., $V_{\text{France}} - V_{\text{Paris}} \approx V_{\text{Germany}} - V_{\text{Berlin}}$), are also captured.



3.1.2. Mathematical Foundation

The goal of Word2Vec is to learn a set of embedding vectors $\{v_w\}_{w \in V}$ for all words in the vocabulary. The models are trained by optimizing an objective function over a large text corpus. For the **Skip-gram** model, the objective is to maximize the probability of observing the actual context words given a center word. For a center word w_t and its context words $C(t)$, the objective function is:

$$J(\theta) = \frac{1}{T} \sum_{t=1}^{T} \sum_{j \in C(t)} \log P(w_j | w_t; \theta)$$

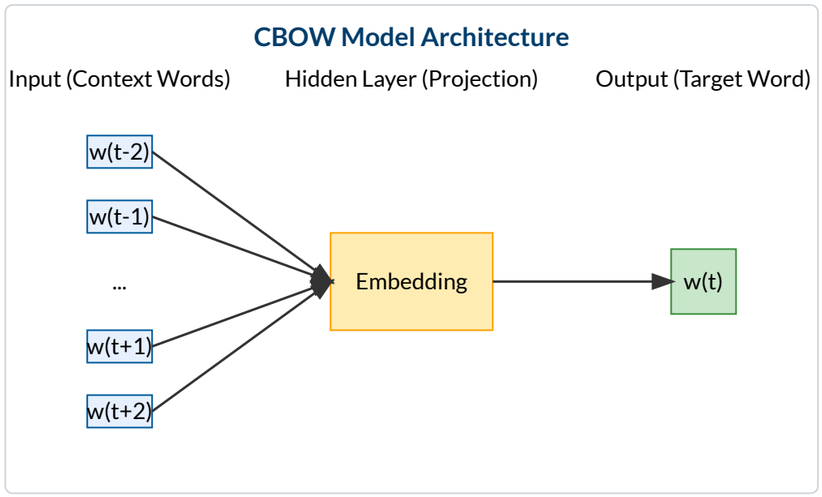
The probability $P(w_j|w_t)$ is typically defined using the **softmax function**:

$$P(w_j|w_t) = \frac{\exp(v'_{w_j} \cdot v_{w_t})}{\sum_{i=1}^{|V|} \exp(v'_{w_i} \cdot v_{w_t})}$$

Here, v_w is the "center" vector for word w, and v'_w is its "context" vector. The summation in the denominator is over the entire vocabulary, which is computationally very expensive. To make training feasible, optimization techniques like **Negative Sampling** or **Hierarchical Softmax** are used instead of the full softmax.

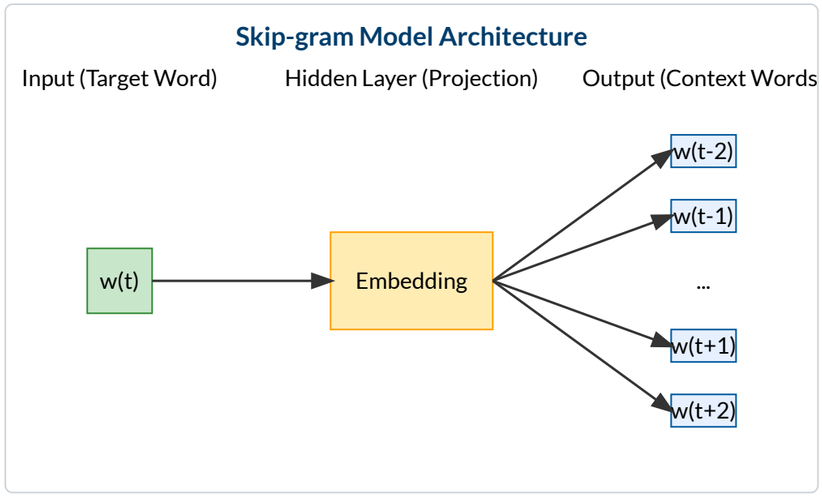
3.1.3. Continuous Bag-of-Words (CBOW)

Concept: The CBOW architecture predicts a target word from its surrounding context words. For example, given the context {"The", "brown", "fox"}, it tries to predict the word "quick". It is faster to train and performs well with frequent words.



3.1.4. Skip-gram

Concept: The Skip-gram architecture does the inverse: it uses a single input word to predict its surrounding context words. Given "quick", it tries to predict {"The", "brown", "fox"}. Skip-gram is slower but is considered to perform better for infrequent words and complex semantic relationships.


Skip-gram Model Architecture
Input (Target Word)
Hidden Layer (Projection)
Output (Context Words
w(t)
Embedding
w(t-2)
w(t-1)
...
w(t+1)
w(t+2)

3.1.5. Making Training Feasible: Optimization Techniques

As mentioned, the softmax calculation is a major bottleneck. The following two techniques are the standard solutions to this problem.

Hierarchical Softmax

Concept: Instead of predicting one word out of $|V|$ options, Hierarchical Softmax reframes the problem. It represents the vocabulary as a binary tree (specifically, a Huffman tree), where each leaf is a word. The task now is to predict the path from the root of the tree to the target word. At each node in the tree, the model learns a simple binary classifier (e.g., go left or go right). The probability of reaching a specific word is the product of the probabilities of taking the correct turns along the path.

Example: Imagine a vocabulary {cat, dog, fox, mat}. The model doesn't calculate probabilities for all 4 words. Instead, it might learn a path. To predict "fox", it might need to go 'left' at the root, then 'right' at the next node. The probability would be $P(\text{fox}|\text{context}) = P(\text{left}|\text{root}) \times P(\text{right}|\text{node 1})$. This reduces the complexity from $O(|V|)$ to $O(\log_2 |V|)$, which is a massive speed-up.

Negative Sampling

Concept: Negative Sampling offers a simpler, yet highly effective, alternative. For each training instance (a center word and a true context word), it creates a few "negative" samples by picking random words from the vocabulary. The model's objective is then changed from a multi-class prediction to a set of binary classifications. It learns to distinguish the true "positive" pair from the randomly selected "negative" pairs.

Example: Given the sentence "... quick brown fox ...", the Skip-gram model would create a positive training sample: (input: `quick`, target: `brown`). For negative sampling, we might pick $k = 2$ random words from the vocabulary, say `apple` and `sky`. This gives us two negative samples: (`quick`, `apple`) and (`quick`, `sky`). The model is then trained with these three samples:

- (`quick`, `brown`) -> Target: 1 (This is a true context pair)
- (`quick`, `apple`) -> Target: 0 (This is a fake context pair)
- (`quick`, `sky`) -> Target: 0 (This is a fake context pair)

This approach is highly efficient because the model only updates a handful of weights (for the positive word and the few negative words) in each step, instead of all $|V|$ weights.

3.1.6. Drawbacks of Word2Vec

- **Out-of-Vocabulary (OOV) Words:** Word2Vec learns embeddings only for words present in its training vocabulary. It cannot produce a vector for a new, unseen word.
- **Struggles with Polysemy:** It assigns a single vector to each word, regardless of its context. A word like "bank" (a financial institution vs. a river edge) will have a single embedding that averages out its different meanings.
- **Local Context Window:** The model learns from a small, local window of surrounding words. This can sometimes prevent it from capturing broader, more global contextual information from a document.

Supplementary / Suggested Reads

- **Original Paper:**
  - **Efficient Estimation of Word Representations in Vector Space (Word2Vec)** - by Tomas Mikolov, et al.
- **Guides and Tutorials:**
  - **The Illustrated Word2vec** - by Jay Alammar. An excellent visual guide.
  - **The amazing power of word vectors** - by Adrian Colyer. A great summary of the concepts.