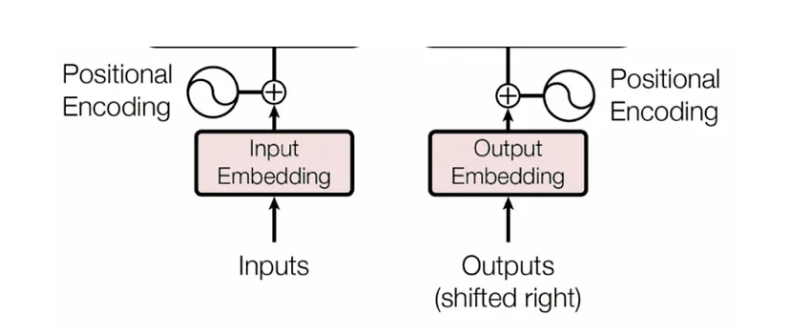
EMBEDDINGS

The Transformer is an architecture that significantly improves the performance of deep learning Natural Language Processing models. Transformers can be **very efficiently parallelized** and this ability allows it to train really big models. And what sets it apart from RNN and CNN is a new concept introduced in Transformers called **the attention mechanism**

The important features of the transformer are: **tokenisation**, the **embedding layer,** the **attention mechanism,** the **encoder** and the **decoder.** In this we are focusing on the **embedding layer**

In transformer models, the very first step is Embeddings in converting tokenized text into numerical vectors that the model can understand and process.



**Input embeddings** are like a translator for the model. They convert words or tokens from the input sequence into **dense vectors of numbers** which the model can then use to perform tasks such as classification, translation, or generation. These vectors represent not just the word itself but also **capture its meaning and relationship to other words in the context of the sequence**.

Each word in the sequence is mapped to a vector of a fixed size. These vectors are learned in a way that similar words (like “dog” and “cat”) will have embeddings that are closer in the vector space.

# Word Embeddings

**Word Embeddings** are **dense vector representations of words** in a continuous vector space. Words with similar meanings have similar embeddings (i.e., are close together in this space).

They capture **semantic relationships** like:

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

**Key Properties:**

* Fixed size vector (e.g., 100 or 300 dimensions)
* Context **independent** (e.g., Word2Vec, GloVe)
* Later models like BERT give **contextualized** word embeddings

# Sentence Embeddings

**Sentence Embeddings** represent an **entire sentence or paragraph** as a single dense vector. This allows comparing entire sentences for similarity, clustering, classification, etc.

Unlike word embeddings, these capture:

* Word meaning
* Word **order**
* Sentence **structure and context**

# Word vs Sentence Embeddings

|  |  |  |
| --- | --- | --- |
| **Feature** | **Word Embeddings** | **Sentence Embeddings** |
| Unit of input | Individual word | Full sentence or paragraph |
| Context awareness | Often limited | High (contextualized models) |
| Vector size | Fixed (e.g., 300) | Fixed (e.g., 384, 768, etc.) |
| Common models | Word2Vec, GloVe, FastText | BERT, RoBERTa, SBERT |
| Use cases | Word similarity, analogy | Sentence similarity, clustering |

**When to Use What?**

* Use **word embeddings** when analysing individual terms, keywords, or building custom models.
* Use **sentence embeddings** when comparing or classifying entire sentences, FAQ matching, chatbot responses, etc.

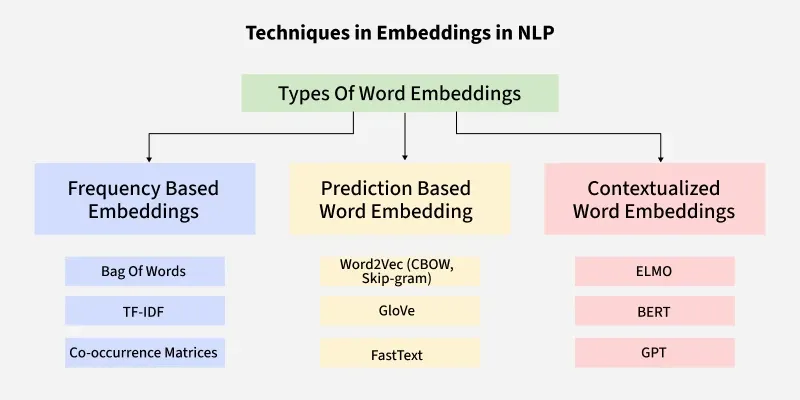
**What is the need for word embedding in NLP?**

Word embeddings are fundamental in NLP for several reasons:

* [**Dimensionality Reduction**](https://www.geeksforgeeks.org/machine-learning/dimensionality-reduction/)**:** They represent words in a lower-dimensional continuous vector space, making it computationally efficient to handle extensive vocabularies.
* [**Semantic Similarity**](https://www.geeksforgeeks.org/nlp/different-techniques-for-sentence-semantic-similarity-in-nlp/)**:**Word embeddings encode semantic relationships, allowing algorithms to understand synonyms, antonyms, and related meanings.
* **Contextual Information:** By capturing context from surrounding words, embeddings help models understand a word's meaning in context, crucial for tasks like sentiment analysis and named entity recognition.
* **Generalization:**They generalize well to unseen words, learning from the distributional properties of words in training data.
* **Feature Representation:** Word embeddings serve as feature representations for machine learning models, enabling the application of various techniques to NLP tasks.
* **Efficient Training:** Models trained with word embeddings converge faster and often perform better than those using sparse representations.
* [**Transfer Learning:**](https://www.geeksforgeeks.org/machine-learning/ml-introduction-to-transfer-learning/) Pre-trained embeddings, like [Word2Vec](https://www.geeksforgeeks.org/python/python-word-embedding-using-word2vec/)or GloVe, allow models to leverage knowledge from large corpora, even with limited task-specific data.

# Word Embedding Techniques

The journey of numerical representation in NLP has evolved significantly over time. Below are the Popular and simple method of feature extraction with text data which are currently used:



1. Bag-of-Words

The **bag-of-words model** is a simplifying representation used in NLP. In this model, a text is represented as the bag of its words, disregarding grammar and even word order but keeping [multiplicity](https://en.wikipedia.org/wiki/Multiplicity_(mathematics)). It is simple to understand and implement and has seen great success in problems such as language modelling and document classification.

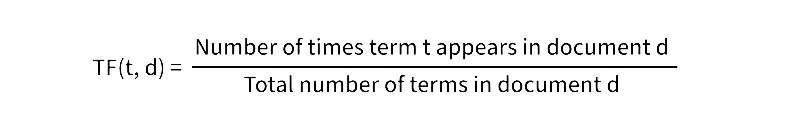
Although simple, this has limitations as it does not capture the context of words within sentences.

### TF-IDF

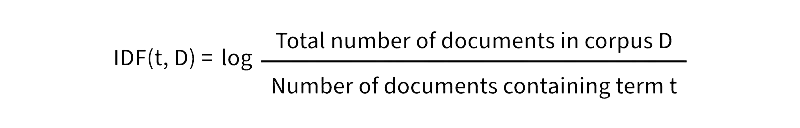
TF-IDF is a statistical measure used to determine the mathematical significance of words in documents. In other words, it’s a method used in information retrieval to evaluate how important a word is to a document in relation toa larger collection of documents.

TF-IDF combines two components:

1. **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.



1. **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.



The TF-IDF score for a term t in a document d is then given by multiplying the TF and IDF values:

**TF−IDF (t, d, D) =TF (t, d) ×IDF (t, D) TF−IDF (t, d, D) =TF (t, d) ×IDF (t, D)**

1. **Converting Text into vectors with TF-IDF**

Let's take an example where we have a corpus (a collection of documents) with three documents and our goal is to calculate the TF-IDF score for specific terms in these documents.

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."

Our goal is to calculate the TF-IDF score for specific terms in these documents. Let’s focus on the word **"cat"** and see how TF-IDF evaluates its importance.

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

In Document 1 and Document 3 the word **"**cat**"** has the same TF score. This means it appears with the same relative frequency in both documents. In Document 2 the TF score is 0 because the word **"**cat**"** does not appear.

**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) =log32≈0.176IDF (cat, D) =log23​≈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) 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)-0x 0.176-0
* For Document 3: TF-IDF (cat, Document 3, D)-0.167 x 0.176 ~ 0.029

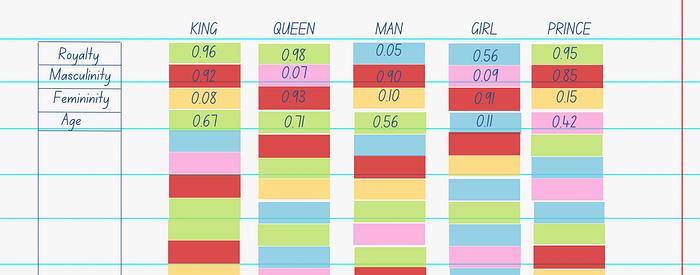
1. Applications
2. Document Similarity and Clustering**:**By converting documents into numerical vectors TF-IDF enables comparison and grouping of related texts. This is valuable for clustering news articles, research papers or customer support tickets into meaningful categories.
3. Text Classification**:** It helps in identify patterns in text for spam filtering, sentiment analysis and topic classification.
4. Keyword Extraction**:** It ranks words by importance making it possible to automatically highlight key terms, generate document tags or create concise summaries.
5. Recommendation Systems**:**Through comparison of textual descriptions TF-IDF supports suggesting related articles, videos or products enhancing user engagement.

<https://www.geeksforgeeks.org/machine-learning/understanding-tf-idf-term-frequency-inverse-document-frequency/>

### Word2Vec

Word2Vec creates a representation of each word present in our vocabulary into a vector. Words used in similar contexts or having semantic relationships are captured effectively through their closeness in the vector space- effectively speaking similar words will have similar word vectors! History. Word2vec was created, patented, and published in 2013 by a team of researchers led by Tomas Mikolov at Google.

Let us consider a classic example: “king”, “queen”, “man”, “girl”, “prince”



In a hypothetical world, vectors could then define the weight of each criterion (for example royalty, masculinity, femininity, age etc.) for each of the given words in our vocabulary.

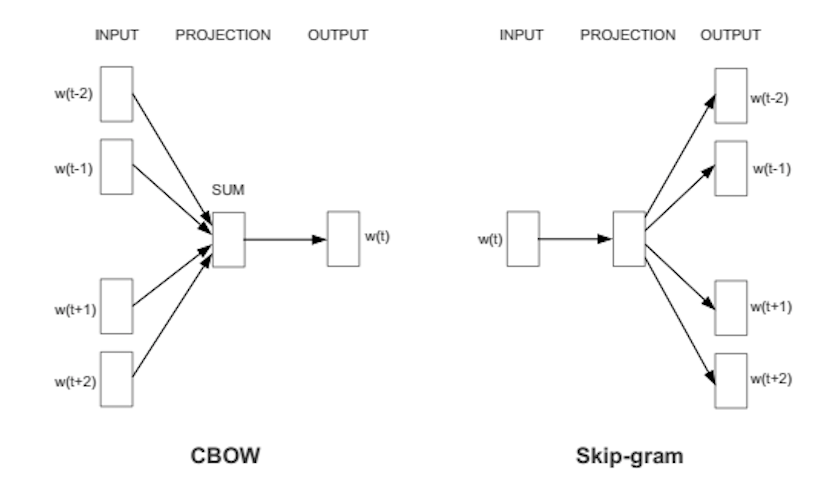
What we then observe is:

* As expected, “king”, “queen”, “prince” has similar scores for “royalty” and “girl”, “queen” has similar scores for “femininity”.
* An operation that removes “man” from “king”, would yield in a vector very close to “queen” (“king”- “man” = “queen”)
* Vectors “king” and “prince” have the same characteristics, except for age, telling us how they might possibly be semantically related to each other.

Word2Vec forms word embeddings that work in a similar fashion except for the fact that the criterion we have used for each of the words are not clearly determinable. What matters to us is the semantic and syntactic relations between words which can still be determined by our model without explicitly having defining features for units of the vector.

Word2Vec has also shown to identify relations like country-capital over larger datasets showing us how powerful word embeddings can be.

Algorithmically, these models are similar.



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

CBOW predicts target words (e.g. ‘mat’) from the surrounding context words (‘the cat sits on the’).  
Statistically, it has the effect that CBOW smoothed over a lot of the distributional information (by treating an entire context as one observation). For the most part, this turns out to be a useful thing for smaller datasets.

**Skip-Gram**

Skip-gram predicts surrounding context words from the target words (inverse of CBOW).  
Statistically, skip-gram treats each context-target pair as a new observation, and this tends to do better when we have larger datasets.

**How does Word2Vec produce word embeddings?**

Word2Vec is a simple neural network with a single hidden layer, and like all neural networks, it has weights, and during training, its goal is to adjust those weights to reduce a loss function. However, Word2Vec is not going to be used for the task it was trained on, instead, we will just take its hidden weights, use them as our word embeddings, and toss the rest of the model

Refer to <https://israelg99.github.io/2017-03-23-Word2Vec-Explained/> for more detailed word2vec architecture and <https://medium.com/@manansuri/a-dummys-guide-to-word2vec-456444f3c673>

Word2Vec struggle with out-of-vocabulary words and morphologically rich languages.

### GloVe

GloVe (Global Vectors 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](https://www.geeksforgeeks.org/nlp/pre-trained-word-embedding-using-glove-in-nlp-models/) is trained on global word co-occurrence statistics. It leverages the global context to create word embeddings that reflect the overall meaning of words based on their co-occurrence probabilities.

In this method, we take the corpus and iterate through it and get the co-occurrence of each word with other words in the corpus. We get a co-occurrence matrix through this. The words which occur next to each other get a value of 1, if they are one word apart then 1/2, if two words apart then 1/3 and so on.

Let's see how the matrix is created. Corpus:

*It is a nice evening.  
Good Evening!  
Is it a nice evening?*

|  | **it** | **is** | **a** | **nice** | **evening** | **good** |
| --- | --- | --- | --- | --- | --- | --- |
| **it** | 0 |  |  |  |  |  |
| **is** | 1+1 | 0 |  |  |  |  |
| **a** | 1/2+1 | 1+1/2 | 0 |  |  |  |
| **nice** | 1/3+1/2 | 1/2+1/3 | 1+1 | 0 |  |  |
| **evening** | 1/4+1/3 | 1/3+1/4 | 1/2+1/2 | 1+1 | 0 |  |
| **good** | 0 | 0 | 0 | 0 | 1 | 0 |

The upper half of the matrix will be a reflection of the lower half. We can consider a window frame as well to calculate the co-occurrences by shifting the frame till the end of the corpus. This helps gather information about the context in which the word is used.

* Vectors for each word is assigned randomly.
* Take two pairs of vectors and see closeness in space.
* If they occur together more often or have a higher value in the co-occurrence matrix and are far apart in space then they are brought close to each other.
* If they are close to each other but are rarely or not frequently used together then they are moved further apart in space.
* Output: Vector space representation that approximates the information from the co-occurrence matrix.

<https://www.geeksforgeeks.org/nlp/Glove-Word-Embedding-in-NLP/>

### FastText

FastText extends the Skip-gram and CBOW models by representing words as bags of character n-grams rather than atomic units. This fundamental shift allows the model to generate embeddings for previously unseen words and capture morphological relationships between related terms.

1. **The Subword Approach**

Traditional word embedding models treat each word as an indivisible token. FastText breaks words into character n-grams, enabling it to understand word structure and meaning at a granular level.

Consider the word "running":

* **3-grams:** <ru, run, unn, nni, nin, ing, ng>
* **4-grams:**<run, runn, unni, nnin, ning, ing>
* **5-grams:** <runn, runni, unnin, nning, ning>

The angle brackets indicate word boundaries, helping the model distinguish between sub-words that appear at different positions.

1. **Hierarchical Softmax Optimization**

FastText employs hierarchical softmax instead of standard [softmax](https://www.geeksforgeeks.org/deep-learning/the-role-of-softmax-in-neural-networks-detailed-explanation-and-applications/) for computational efficiency. Rather than computing probabilities across all vocabulary words, it constructs a binary tree where each leaf represents a word and internal nodes represent probability distributions.

1. **Key advantages of hierarchical softmax:**

* Reduces time complexity from O(V) to O (log V) where V is vocabulary size
* Uses Huffman coding to optimize frequent word access
* Maintains prediction accuracy while significantly improving training speed

1. **Edge Cases**

* **Character encoding issues**: FastText requires consistent UTF-8 encoding across training and inference data. Mixed encodings can lead to inconsistent subword generation.
* **Optimal n-gram range**: The choice of minimum and maximum n-gram lengths depends on the target language. For English, 3-6 character n-grams typically work well, while morphologically rich languages may benefit from longer ranges.
* **Training data quality**: FastText is sensitive to preprocessing decisions. Inconsistent tokenization or normalization can degrade model quality, particularly for subword-based features.

1. **Practical Applications**

FastText excels in scenarios requiring robust of morphological variations and out-of-vocabulary words. It's particularly effective for:

* **Multilingual applications** where training data may be limited for some languages
* **Domain-specific text** with specialized vocabulary not found in general corpora
* **Real-time systems** requiring fast inference and low memory overhead
* **Text classification tasks** where subword information provides discriminative features

The library's combination of efficiency and linguistic sophistication makes it a valuable tool for production NLP systems, especially when dealing with diverse or evolving vocabularies where traditional word-level approaches fall short.

1. **Advantages and Limitations**

**Key Advantages**

* **OOV handling**: Generates embeddings for unseen words through subword information
* **Morphological awareness**: Captures relationships between word variants (run, running, runner)
* **Computational efficiency**: Fast training and inference through hierarchical softmax
* **Language flexibility**: Works well with morphologically rich languages

**Limitations**

* **Memory overhead**: Requires more storage than traditional embeddings due to subword information
* **Hyperparameter sensitivity**: N-gram range (minn, maxn) significantly affects performance
* **Limited semantic depth**: May not capture complex semantic relationships as well as transformer-based models

<https://www.geeksforgeeks.org/machine-learning/fasttext-working-and-implementation/>