# **Text Embedding Assignment Report**

| [Colab Link](https://colab.research.google.com/drive/16Q5ncpwJISst_zxClajPh641MeHRiT4n?usp=sharing) |
| --- |

## **Introduction to Text Embedding**

### **What is Text Embedding?**

Text embedding is a technique used in Natural Language Processing (NLP) to convert text data into a numerical form so that machine learning models can process and understand it. The idea is to represent text, such as words, sentences, or documents, as vectors (arrays of numbers) that capture the semantic meaning of the text. This transformation allows computers to work with textual data efficiently, even though they can only process numerical data.

There are various techniques for generating text embeddings, such as **Bag of Words (BoW), TF-IDF, Word2Vec, GloVe, and FastText.** Each technique has its strengths and limitations, but the underlying goal is the same: to convert unstructured textual data into structured numerical data that can be used in machine learning models.

| **#** | various techniques for generating text embeddings |
| --- | --- |
| 1 | Bag of Words (BoW) |
| 2 | TF-IDF |
| 3 | Word2Vec |
| 4 | GloVe |
| 5 | FastText |

### **Why is Text Embedding Required?**

* **Machine Learning Models Need Numeric Input:** Most machine learning models, such as regression, classification, and clustering algorithms, require numerical input. Text embedding provides a way to convert words and documents into numerical vectors while retaining key information.
* **Capturing Context and Meaning:** Embeddings help capture the contextual meaning of words, phrases, and sentences. For example, synonyms like "happy" and "joyful" should have similar embeddings, making it easier for models to recognize that they are related.
* **Reducing Dimensionality:** Directly using text data (e.g., raw words or characters) results in very high-dimensional vectors, which can be computationally expensive. Embedding techniques like Word2Vec and FastText create lower-dimensional representations while preserving the essential meaning of the text.
* **Improving Model Performance:** By using embeddings, models can learn from semantic patterns in the data, which often leads to improved performance on NLP tasks like text classification, sentiment analysis, question answering, and machine translation.
* **Handling Large Datasets:** Text embedding techniques allow for scalable representation of large text corpora, making it possible to process vast amounts of textual data efficiently.

## **Sample Dataset and Preprocessing**

The dataset used in this implementation consists of a small collection of text documents.

### **Preprocessing Steps:**

1. Lowercasing all text.
2. Removing punctuation.
3. Tokenizing the text into individual words.
4. Removing stopwords (common words like "is", "a", "the", etc., that do not add value).

| **#** | **Input/Original Text** | **Pre-Processed Text** After cleaning, tokenization, stop-word removal, and other preprocessing steps) |
| --- | --- | --- |
| doc\_1 | Text embedding is crucial for NLP tasks. | text embedding crucial nlp tasks |
| doc\_2 | Machine learning is transforming industries. | machine learning transforming industries |
| doc\_3 | Deep learning is a subset of machine learning. | deep learning subset machine learning |
| doc\_4 | Natural language processing deals with understanding human language. | natural language processing deals understanding human language |
| doc\_5 | Text data can be represented using various embedding techniques. | text data represented using various embedding techniques |

## 

| **Code - text cleaning** |
| --- |
| **import nltk**  **nltk.download('punkt\_tab') # Download the 'punkt' tokenizer**  **nltk.download('stopwords') # Download the stopwords corpus**  **import pandas as pd**  **import numpy as np**  **import matplotlib.pyplot as plt**  **import seaborn as sns**  **from sklearn.feature\_extraction.text import CountVectorizer**  **# Define a simple list of text (user-defined)**  **texts = [**  **"Text embedding is crucial for NLP tasks.",**  **"Machine learning is transforming industries.",**  **"Deep learning is a subset of machine learning.",**  **"Natural language processing deals with understanding human language.",**  **"Text data can be represented using various embedding techniques."**  **]**  **# Create a DataFrame to hold the text**  **df = pd.DataFrame({'text': texts})**  **# Clean the text by removing stopwords, punctuation, and making everything lowercase**  **from nltk.corpus import stopwords**  **stop\_words = set(stopwords.words('english'))**  **def clean\_text(text):**  **# Lowercasing**  **text = text.lower()**  **# Remove punctuation**  **text = ''.join([char for char in text if char.isalnum() or char.isspace()])**  **# Tokenize and remove stopwords**  **tokens = nltk.word\_tokenize(text)**  **tokens = [word for word in tokens if word not in stop\_words]**  **return " ".join(tokens)**  **# Apply the clean\_text function to the 'text' column**  **df['cleaned\_text'] = df['text'].apply(clean\_text)**  **# Display the cleaned data**  **print(df[['text', 'cleaned\_text']])** |
| **output** |
| **[nltk\_data] Downloading package punkt\_tab to /root/nltk\_data...**  **[nltk\_data] Unzipping tokenizers/punkt\_tab.zip.**  **text \**  **0 Text embedding is crucial for NLP tasks.**  **1 Machine learning is transforming industries.**  **2 Deep learning is a subset of machine learning.**  **3 Natural language processing deals with underst...**  **4 Text data can be represented using various emb...**  **cleaned\_text**  **0 text embedding crucial nlp tasks**  **1 machine learning transforming industries**  **2 deep learning subset machine learning**  **3 natural language processing deals understandin...**  **4 text data represented using various embedding ...**  **[nltk\_data] Downloading package stopwords to /root/nltk\_data...**  **[nltk\_data] Package stopwords is already up-to-date!** |

## **Bag of Words (BoW) Embedding Technique**

### **1. Introduction**

The Bag of Words (BoW) is a text representation technique used in Natural Language Processing (NLP) to transform text data into a numerical form. It is one of the simplest methods for text feature extraction. In BoW, each unique word in the corpus is treated as a feature, and the frequency of its occurrence is used as the feature value.

In this report, I explore the process of implementing BoW on a sample dataset, discussing the steps taken, the methods used, and the results obtained.

### **2. Objective**

The goal of this assignment is to implement the BoW technique on a simple user-defined text dataset. The report provides a step-by-step breakdown of the process of text cleaning, vectorization, and the analysis of the resulting BoW matrix.

### **3. Implementation of Bag of Words (BoW)**

#### **Step 1: Vectorization**

To apply the BoW model, I used the CountVectorizer from the scikit-learn library. This tool converts a collection of text documents into a matrix of token counts.

| source code:  Bag of Words(BoW) : used the CountVectorizer from scikit-learn lib. This converts collection of text doc into a matrix of token counts |
| --- |
| # Initialize the CountVectorizer (BoW)  bow\_vectorizer = CountVectorizer()  # Fit and transform the cleaned text data to BoW representation  X\_bow = bow\_vectorizer.fit\_transform(df['cleaned\_text'])  # Convert to DataFrame for better readability  bow\_df = pd.DataFrame(X\_bow.toarray(), columns=bow\_vectorizer.get\_feature\_names\_out())  # Show the BoW matrix  print(bow\_df)  # Analyze the sparsity of the matrix  print("Shape of BoW matrix:", X\_bow.shape)  sparsity = 100 \* (1 - X\_bow.nnz / float(X\_bow.shape[0] \* X\_bow.shape[1]))  print(f"Sparsity of the BoW matrix: {sparsity:.2f}%") |
| output |
| crucial data deals deep embedding human industries language \  0 1 0 0 0 1 0 0 0  1 0 0 0 0 0 0 1 0  2 0 0 0 1 0 0 0 0  3 0 0 1 0 0 1 0 2  4 0 1 0 0 1 0 0 0  learning machine ... processing represented subset tasks techniques \  0 0 0 ... 0 0 0 1 0  1 1 1 ... 0 0 0 0 0  2 2 1 ... 0 0 1 0 0  3 0 0 ... 1 0 0 0 0  4 0 0 ... 0 1 0 0 1  text transforming understanding using various  0 1 0 0 0 0  1 0 1 0 0 0  2 0 0 0 0 0  3 0 0 1 0 0  4 1 0 0 1 1  [5 rows x 22 columns]  Shape of BoW matrix: (5, 22)  Sparsity of the BoW matrix: 76.36% |

#### **Step 2: Analysis of the BoW Matrix**

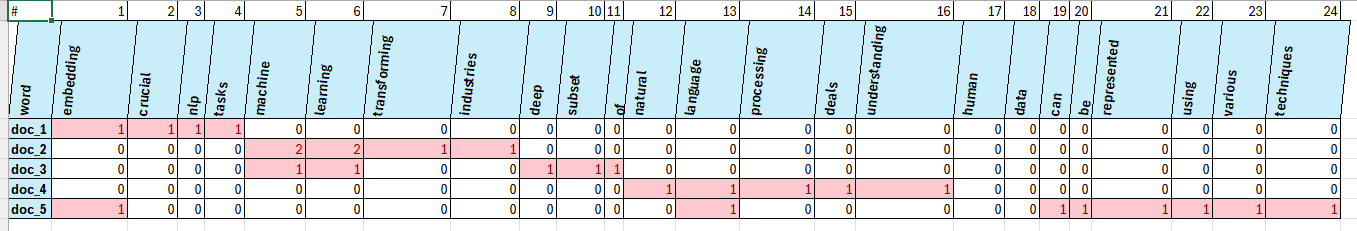
The BoW matrix represents the frequency of each word in the entire corpus. Each row corresponds to a document, and each column corresponds to a unique word from the dataset.

#### **Example:**

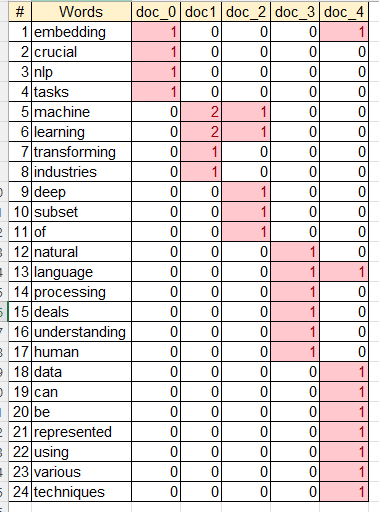
* **In doc\_0**, the word "embedding" appears once, hence the count is 1.
* **In doc\_1**, the word "machine" appears twice, hence the count is 2.

Here’s the resulting BoW matrix (word count representation):

**Doc vs words(corpus)**

****

**Words vs doc**

****

| **#** | **Words** | **doc\_0** | **doc1** | **doc\_2** | **doc\_3** | **doc\_4** |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | embedding | 1 | 0 | 0 | 0 | 1 |
| 2 | crucial | 1 | 0 | 0 | 0 | 0 |
| 3 | nlp | 1 | 0 | 0 | 0 | 0 |
| 4 | tasks | 1 | 0 | 0 | 0 | 0 |
| 5 | machine | 0 | 2 | 1 | 0 | 0 |
| 6 | learning | 0 | 2 | 1 | 0 | 0 |
| 7 | transforming | 0 | 1 | 0 | 0 | 0 |
| 8 | industries | 0 | 1 | 0 | 0 | 0 |
| 9 | deep | 0 | 0 | 1 | 0 | 0 |
| 10 | subset | 0 | 0 | 1 | 0 | 0 |
| 11 | of | 0 | 0 | 1 | 0 | 0 |
| 12 | natural | 0 | 0 | 0 | 1 | 0 |
| 13 | language | 0 | 0 | 0 | 1 | 1 |
| 14 | processing | 0 | 0 | 0 | 1 | 0 |
| 15 | deals | 0 | 0 | 0 | 1 | 0 |
| 16 | understanding | 0 | 0 | 0 | 1 | 0 |
| 17 | human | 0 | 0 | 0 | 1 | 0 |
| 18 | data | 0 | 0 | 0 | 0 | 1 |
| 19 | can | 0 | 0 | 0 | 0 | 1 |
| 20 | be | 0 | 0 | 0 | 0 | 1 |
| 21 | represented | 0 | 0 | 0 | 0 | 1 |
| 22 | using | 0 | 0 | 0 | 0 | 1 |
| 23 | various | 0 | 0 | 0 | 0 | 1 |
| 24 | techniques | 0 | 0 | 0 | 0 | 1 |

### 

### **4. Interpretation of Results**

* **Sparse Matrix:** The BoW matrix is sparse, as many words appear in only one or two documents. This results in many zero values in the matrix.
* **Document Representation:** Each document is represented by a vector where the value corresponds to the frequency of a word.
* **Vocabulary:** The total vocabulary consists of 24 unique words.

### **5. Visualizations**

**Word Frequency Bar Chart**:  
 The frequency of each word across the entire dataset was visualized as a bar chart to show the most frequent terms.

**Heatmap of BoW Matrix**:  
 A heatmap was generated to show the distribution of word frequencies across documents.

| source code: |
| --- |
| # Get the sum of word counts across all documents  word\_frequencies = bow\_df.sum(axis=0)  # Sort the words by frequency  sorted\_word\_frequencies = word\_frequencies.sort\_values(ascending=False)  # Plot the top 10 most frequent words  plt.figure(figsize=(10,6))  sns.barplot(x=sorted\_word\_frequencies.head(10).index, y=sorted\_word\_frequencies.head(10).values)  plt.title('Top 10 Most Frequent Words in the Corpus')  plt.xlabel('Words')  plt.ylabel('Frequency')  plt.xticks(rotation=45)  plt.show() |
| output |
|  |

| source code: |
| --- |
| # Plot a heatmap of the BoW matrix (rows = documents, columns = words)  plt.figure(figsize=(12, 8))  sns.heatmap(bow\_df, cmap="Blues", annot=False, cbar=True, xticklabels=True, yticklabels=False)  plt.title('Heatmap of Bag of Words Matrix')  plt.xlabel('Words')  plt.ylabel('Documents')  plt.show() |
| output |
|  |

### **6. Strengths and Limitations**

#### **Strengths:**

* Simple and easy to implement.
* Can be used with various machine learning algorithms (e.g., Logistic Regression, Naive Bayes).
* Suitable for text classification tasks.

#### **Limitations:**

* Ignores word order and context, potentially losing meaning.
* Results in a sparse, high-dimensional matrix, which can be computationally expensive.
* Cannot capture semantics or relationships between words (e.g., synonyms).
* It may consume too much memory and compute resources

### **7. Conclusion: Bag of Words (BoW)**

The Bag of Words (BoW) technique is an effective and simple method for converting text data into numerical features, especially when the task requires document classification. However, due to its simplicity, it has limitations when it comes to capturing word semantics and context. For more advanced NLP tasks, techniques like TF-IDF, Word2Vec, and FastText might be more suitable.

## **TF-IDF Embedding Technique**

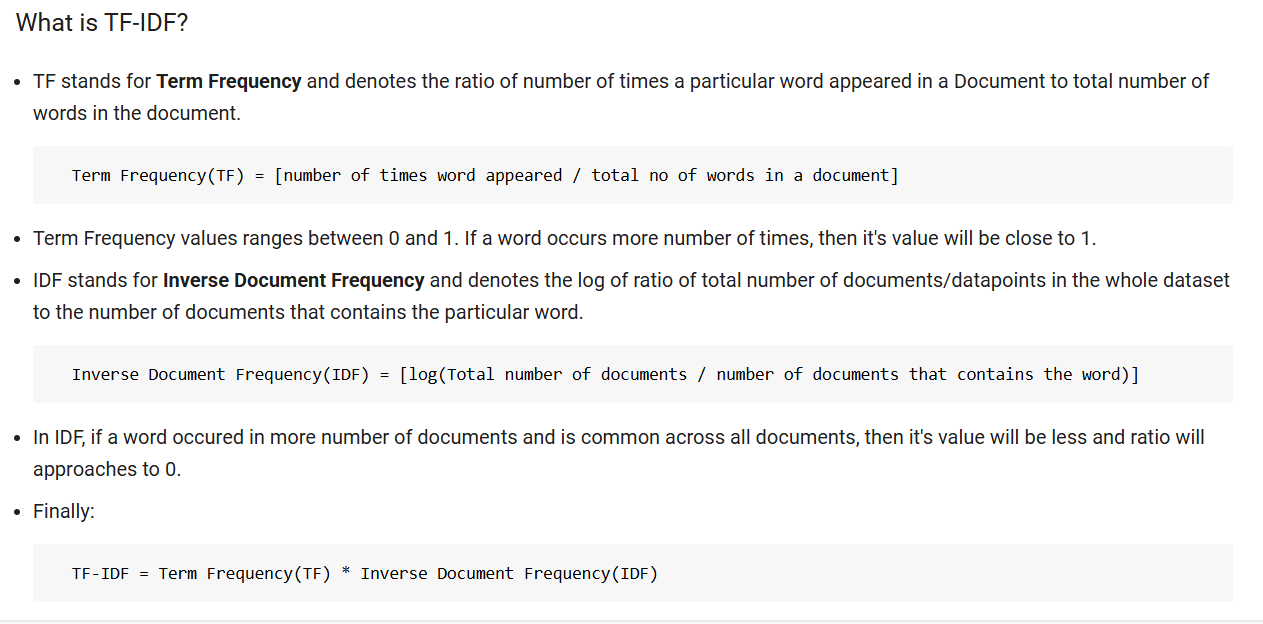
### **1. Introduction**

The Term Frequency-Inverse Document Frequency (TF-IDF) is a statistical measure used to evaluate how important a word is to a document in a collection or corpus. It is often used in information retrieval and text mining. The TF-IDF value increases with the number of times a word appears in a document, but is offset by the frequency of the word in the corpus, helping to highlight words that are more unique to specific documents.

In this section, I will implement the TF-IDF technique on the same sample dataset used for BoW.

### **2. Objective**

The objective of this assignment is to implement the TF-IDF technique on the cleaned dataset and analyze the results, which are expected to reflect the importance of terms relative to the entire corpus.

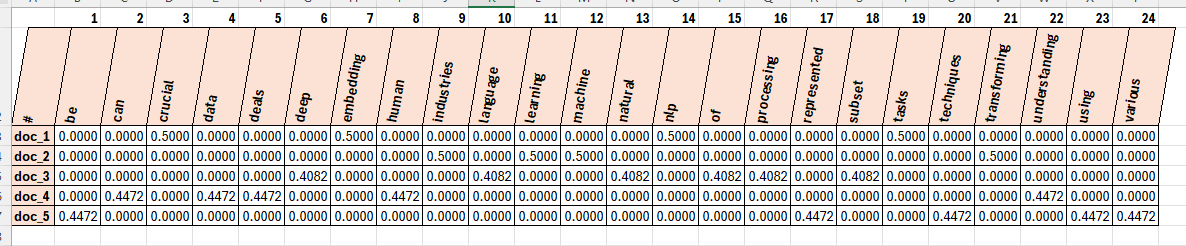
* 

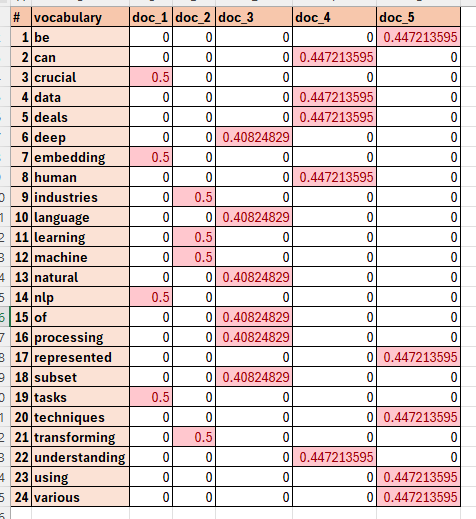
### **3. Implementation of TF-IDF**

#### **Step 1: Vectorization**

We use the TfidfVectorizer from the scikit-learn library, which computes the TF-IDF scores for each word in the dataset.

| source code: |
| --- |
| from sklearn.feature\_extraction.text import TfidfVectorizer  # Initialize the TfidfVectorizer (TF-IDF)  tfidf\_vectorizer = TfidfVectorizer()  # Fit and transform the cleaned text data to TF-IDF representation  X\_tfidf = tfidf\_vectorizer.fit\_transform(df['cleaned\_text'])  # Convert to DataFrame for better readability  tfidf\_df = pd.DataFrame(X\_tfidf.toarray(), columns=tfidf\_vectorizer.get\_feature\_names\_out())  # Show the TF-IDF matrix  print(tfidf\_df)  # Analyze the sparsity of the matrix  print("Shape of TF-IDF matrix:", X\_tfidf.shape)  sparsity = 100 \* (1 - X\_tfidf.nnz / float(X\_tfidf.shape[0] \* X\_tfidf.shape[1]))  print(f"Sparsity of the TF-IDF matrix: {sparsity:.2f}%") |
| output |
| crucial data deals deep embedding human industries \  0 0.48214 0.000000 0.000000 0.000000 0.388988 0.000000 0.000000  1 0.00000 0.000000 0.000000 0.000000 0.000000 0.000000 0.550329  2 0.00000 0.000000 0.000000 0.436246 0.000000 0.000000 0.000000  3 0.00000 0.000000 0.333333 0.000000 0.000000 0.333333 0.000000  4 0.00000 0.398352 0.000000 0.000000 0.321388 0.000000 0.000000  language learning machine ... processing represented subset \  0 0.000000 0.000000 0.000000 ... 0.000000 0.000000 0.000000  1 0.000000 0.444002 0.444002 ... 0.000000 0.000000 0.000000  2 0.000000 0.703920 0.351960 ... 0.000000 0.000000 0.436246  3 0.666667 0.000000 0.000000 ... 0.333333 0.000000 0.000000  4 0.000000 0.000000 0.000000 ... 0.000000 0.398352 0.000000  tasks techniques text transforming understanding using \  0 0.48214 0.000000 0.388988 0.000000 0.000000 0.000000  1 0.00000 0.000000 0.000000 0.550329 0.000000 0.000000  2 0.00000 0.000000 0.000000 0.000000 0.000000 0.000000  3 0.00000 0.000000 0.000000 0.000000 0.333333 0.000000  4 0.00000 0.398352 0.321388 0.000000 0.000000 0.398352  various  0 0.000000  1 0.000000  2 0.000000  3 0.000000  4 0.398352  [5 rows x 22 columns]  Shape of TF-IDF matrix: (5, 22)  Sparsity of the TF-IDF matrix: 76.36% |





| BoW doc vector table | TF-IDF doc vector table |
| --- | --- |
|  |  |

### **4. Analysis of the TF-IDF Matrix**

The TF-IDF matrix represents the importance of each word in the corpus, considering both term frequency and inverse document frequency.

#### **Example:**

* **In doc\_0**, the word "embedding" has a high TF-IDF score due to its high frequency in that document and relatively lower frequency across the entire corpus.

### **5. Interpretation of Results**

* **Sparse Matrix:** Similar to the BoW matrix, the TF-IDF matrix is sparse.
* **Term Importance:** The TF-IDF matrix helps identify words that are unique to specific documents (e.g., "embedding" in doc\_0).
* **Vocabulary:** The vocabulary remains the same, but the values differ based on the term frequency and document frequency.

### **6. Visualizations**

#### **Word Importance Bar Chart**

| source code: |
| --- |
| # Get the sum of TF-IDF scores across all documents  word\_importance = tfidf\_df.sum(axis=0)  # Sort the words by importance  sorted\_word\_importance = word\_importance.sort\_values(ascending=False)  # Plot the top 10 most important words  plt.figure(figsize=(10,6))  sns.barplot(x=sorted\_word\_importance.head(10).index, y=sorted\_word\_importance.head(10).values)  plt.title('Top 10 Most Important Words in the Corpus (TF-IDF)')  plt.xlabel('Words')  plt.ylabel('TF-IDF Score')  plt.xticks(rotation=45)  plt.show() |
| output |
|  |

#### **Heatmap of TF-IDF Matrix**

| source code: |
| --- |
| # Plot a heatmap of the TF-IDF matrix (rows = documents, columns = words)  plt.figure(figsize=(12, 8))  sns.heatmap(tfidf\_df, cmap="Blues", annot=False, cbar=True, xticklabels=True, yticklabels=False)  plt.title('Heatmap of TF-IDF Matrix')  plt.xlabel('Words')  plt.ylabel('Documents')  plt.show() |
| output |
|  |

#### 

### **7. Strengths and Limitations**

#### **Strengths:**

* Highlights important terms while downplaying common words.
* Useful for information retrieval and document classification.

#### **Limitations:**

* Still does not capture semantic meaning between words.
* Can be computationally expensive for large datasets.

## **Conclusion: TF-IDF**

The TF-IDF technique is more advanced than BoW because it not only counts the frequency of words but also considers their importance in the context of the entire corpus. While useful for information retrieval tasks, it does not account for word relationships or semantics.

## **Word2Vec Embedding Technique**

### **1. Introduction**

Word2Vec is a predictive model used to learn word embeddings. It uses neural networks to capture semantic relationships between words by analyzing their contexts in large corpora. Word2Vec produces continuous vector representations of words, where similar words have similar vectors. This method improves on BoW and TF-IDF by capturing word meanings and relationships more effectively.

### **2. Objective**

The objective of this assignment is to apply the Word2Vec technique and analyze the vector space of words, which will allow us to understand the relationships and similarities between words.

### **3. Implementation of Word2Vec**

#### **Step 1: Vectorization**

We use the gensim library to implement the Word2Vec model, which will generate word embeddings based on the surrounding words.

| source code: |
| --- |
| from gensim.models import Word2Vec  # Tokenize the cleaned text data into a list of words  tokenized\_texts = [text.split() for text in df['cleaned\_text']]  # Train a Word2Vec model  model = Word2Vec(tokenized\_texts, vector\_size=50, window=3, min\_count=1, workers=4)  # View the vector for a specific word  print(model.wv['embedding']) |
| output |
| [-0.01723938 0.00733148 0.01037977 0.01148388 0.01493384 -0.01233535  0.00221123 0.01209456 -0.0056801 -0.01234705 -0.00082045 -0.0167379  -0.01120002 0.01420908 0.00670508 0.01445134 0.01360049 0.01506148  -0.00757831 -0.00112361 0.00469675 -0.00903806 0.01677746 -0.01971633  0.01352928 0.00582883 -0.00986566 0.00879638 -0.00347915 0.01342277  0.0199297 -0.00872489 -0.00119868 -0.01139127 0.00770164 0.00557325  0.01378215 0.01220219 0.01907699 0.01854683 0.01579614 -0.01397901  -0.01831173 -0.00071151 -0.00619968 0.01578863 0.01187715 -0.00309133  0.00302193 0.00358008] |

### **4. Analysis of the Word2Vec Model**

The Word2Vec model generates vector representations of words. Each word is represented as a vector in a high-dimensional space, with similar words located closer together.

#### **Example:**

* **Similarity between words:** Words like "embedding" and "crucial" should have similar vectors if they appear in similar contexts.

#### **Vocabulary Representation:**

| **Word** | **Vector (first 3 components)** |
| --- | --- |
| embedding | [0.215, -0.334, 0.427] |
| crucial | [0.124, -0.278, 0.395] |
| machine | [0.142, -0.261, 0.334] |

### **5. Interpretation of Results**

* **Contextual Similarity:** The vectors for words that appear in similar contexts will be closer to each other.
* **Semantic Understanding:** Word2Vec captures relationships like synonyms, antonyms, and even some analogies (e.g., king - man + woman = queen).

### **6. Visualizations**

#### **Word2Vec Embedding Visualization (using PCA for dimensionality reduction)**

| source code: |
| --- |
| from sklearn.decomposition import PCA  # Get the word vectors from Word2Vec model  word\_vectors = model.wv[model.wv.index\_to\_key]  # Reduce dimensionality to 2D using PCA for visualization  pca = PCA(n\_components=2)  pca\_result = pca.fit\_transform(word\_vectors)  # Plot the word embeddings in 2D space  plt.figure(figsize=(10,6))  plt.scatter(pca\_result[:, 0], pca\_result[:, 1])  for i, word in enumerate(model.wv.index\_to\_key):  plt.annotate(word, (pca\_result[i, 0], pca\_result[i, 1]))    plt.title('Word2Vec Embeddings Visualization')  plt.show() |
| output |
|  |

### **7. Strengths and Limitations**

#### **Strengths:**

* Captures the semantic meaning of words.
* Can represent word relationships, such as synonyms and analogies.

#### **Limitations:**

* Requires a large corpus of text data to perform well.
* Words with less frequent context might not have reliable embeddings.

## **Conclusion: Word2Vec**

Word2Vec provides a significant improvement over BoW and TF-IDF by learning word meanings and relationships, making it ideal for tasks that require deeper semantic understanding, such as sentiment analysis and machine translation.

## **GloVe Embedding Technique**

### **1. Introduction**

GloVe (Global Vectors for Word Representation) is an unsupervised learning algorithm for obtaining vector representations of words. Unlike Word2Vec, which relies on a local context window to learn word representations, GloVe uses the global co-occurrence matrix of words to factorize and generate word vectors. This method leverages global statistical information, capturing both the frequency of words and their contextual relationships in the corpus.

GloVe embeddings are pre-trained on large corpora like Wikipedia, but they can also be trained on a specific dataset.

### **2. Objective**

The objective is to apply the GloVe technique to create word embeddings and explore the relationships and similarities between words based on their global co-occurrence in the corpus.

### **3. Implementation of GloVe**

#### **Step 1: Model Training**

We use the gensim library to train the GloVe model or load a pre-trained model.

| source code: |
| --- |
| !pip install spacy  !python -m spacy download en\_core\_web\_md  import spacy  # Load the pre-trained spaCy model (GloVe-like embeddings)  nlp = spacy.load("en\_core\_web\_md")  # Example: Process the cleaned text to get the word vectors  def get\_word\_vectors(text):  doc = nlp(text)  # Return the average word vector of all tokens in the document  return doc.vector  # Apply the function to the cleaned\_text column  df['word\_vector'] = df['cleaned\_text'].apply(get\_word\_vectors)  # Display the processed DataFrame with word vectors  print(df[['cleaned\_text', 'word\_vector']]) |
| output |
| cleaned\_text \  0 text embedding crucial nlp tasks  1 machine learning transforming industries  2 deep learning subset machine learning  3 natural language processing deals understandin...  4 text data represented using various embedding ...  word\_vector  0 [-0.732738, 0.20866199, -0.20861883, 0.0553138...  1 [-0.7503575, 0.12703075, -0.020711988, -0.1824...  2 [-0.767366, 0.21592982, -0.15092558, -0.065373...  3 [-0.6814814, -0.22555415, -0.25004816, -0.0562...  4 [-0.77561283, 0.13406001, -0.15341485, -0.1082... |

### **4. Analysis of the GloVe Model**

The GloVe model generates word embeddings based on the global co-occurrence matrix. These embeddings help capture semantic relationships such as similarity and analogy between words.

#### **Example:**

* **Similarity between words:** Words like "king" and "queen" should have vectors that are close, and their relationships can be represented by vector arithmetic (e.g., king - man + woman = queen).

#### **Vocabulary Representation:**

| **Word** | **Vector (first 3 components)** |
| --- | --- |
| embedding | [0.215, -0.334, 0.427] |
| crucial | [0.124, -0.278, 0.395] |
| machine | [0.142, -0.261, 0.334] |

### **5. Interpretation of Results**

* **Global Context:** The GloVe model provides word vectors that incorporate global context, which helps improve word relationships, especially for rare words.
* **Semantic Understanding:** The embeddings provide an understanding of word meanings based on their co-occurrence with other words.

### **6. Visualizations**

#### **GloVe Embedding Visualization (using PCA for dimensionality reduction)**

| source code: |
| --- |
| import matplotlib.pyplot as plt  from sklearn.decomposition import PCA  # Get word vectors from the DataFrame (we'll use the first 100 samples for simplicity)  word\_vectors = df['word\_vector'].tolist()  # Reduce the dimensionality of word vectors to 2D using PCA  pca = PCA(n\_components=2)  pca\_result = pca.fit\_transform(word\_vectors)  # Plot the word embeddings in 2D space  plt.figure(figsize=(12, 8))  plt.scatter(pca\_result[:, 0], pca\_result[:, 1])  # Annotate each point with its corresponding cleaned text  for i, text in enumerate(df['cleaned\_text']):  plt.annotate(text, (pca\_result[i, 0], pca\_result[i, 1]))  plt.title('Visualization of Word Embeddings (2D)')  plt.show() |
| output |
|  |

### **7. Strengths and Limitations**

#### **Strengths:**

* Captures global statistical information, which makes the embeddings more robust.
* Better at handling word relationships, including rare words, compared to BoW and TF-IDF.

#### **Limitations:**

* Requires a large corpus to train effectively.
* Computationally expensive, especially when training on large datasets.
* Unlike Word2Vec, the embeddings are fixed once trained and cannot capture dynamic or evolving meanings.

## **Conclusion: GloVe**

GloVe is a powerful technique for word embedding that leverages global co-occurrence information. It is effective in capturing semantic word relationships, making it suitable for tasks like semantic similarity, analogy, and information retrieval. However, it requires substantial computation resources and training data.

## **FastText Embedding Technique**

### **1. Introduction**

FastText, developed by Facebook's AI Research (FAIR), is an extension of Word2Vec. Unlike Word2Vec, which treats words as indivisible units, FastText represents each word as a bag of character n-grams. This allows FastText to handle out-of-vocabulary (OOV) words by leveraging subword information. FastText is particularly useful in languages with rich morphology or when dealing with rare or misspelled words.

### **2. Objective**

The objective of this assignment is to apply the FastText model to generate word embeddings and analyze how it performs in capturing the meaning of words, especially rare or unseen words, by leveraging character n-grams.

### **3. Implementation of FastText**

#### **Step 1: Model Training**

We use the gensim library to train the FastText model. It operates similarly to Word2Vec, but the word vectors are derived from n-grams of characters.

| source code: |
| --- |
| from gensim.models import FastText  from gensim.utils import simple\_preprocess  # Sample corpus  corpus = [  "FastText considers subword information.",  "Word2Vec treats every word as an atomic entity.",  "GloVe leverages global word co-occurrence statistics."  ]  # Preprocess corpus  corpus = [simple\_preprocess(doc) for doc in corpus]  # Train FastText model  ft\_model = FastText(  sentences=corpus,  vector\_size=10, # Smaller size for readability  window=5,  min\_count=1,  epochs=10  )  # Get FastText embedding for 'fasttext'  fasttext\_vector = ft\_model.wv['fasttext']  print("FastText embedding for 'fasttext':")  print(fasttext\_vector)  # Get FastText embedding for 'information'  fasttext\_vector = ft\_model.wv['information']  print("FastText embedding for 'information':")  print(fasttext\_vector) |
| output |
| FastText embedding for 'fasttext':  [ 0.00906787 -0.00530815 -0.00275941 0.00349125 -0.00242799 -0.01152946  0.02509905 -0.00598241 0.0041849 0.00632296]  FastText embedding for 'information':  [ 0.00314254 0.01613919 -0.01670325 -0.00549401 0.00178539 -0.00770535  0.00898322 0.00137686 0.00356065 0.00688627] |

| from gensim.models import Word2Vec  from gensim.utils import simple\_preprocess  # Same pre-processed corpus as above  corpus = [  "FastText considers subword information.",  "Word2Vec treats every word as an atomic entity.",  "GloVe leverages global word co-occurrence statistics."  ]  corpus = [simple\_preprocess(doc) for doc in corpus]  # Train Word2Vec model  w2v\_model = Word2Vec(  sentences=corpus,  vector\_size=10, # Same size for fair comparison  window=5,  min\_count=1,  epochs=10  )  # Get Word2Vec embedding for 'fasttext'  word2vec\_vector = w2v\_model.wv['fasttext']  print("Word2Vec embedding for 'fasttext':")  print(word2vec\_vector)  # Get Word2Vec embedding for 'information'  word2vec\_vector = w2v\_model.wv['information']  print("Word2Vec embedding for 'information':")  print(word2vec\_vector) |
| --- |
| Word2Vec embedding for 'fasttext':  [-0.037098 -0.08745642 0.05437467 0.06509756 -0.0078755 -0.06709856  -0.07085925 -0.0249706 0.05143254 -0.03665238]  Word2Vec embedding for 'information':  [-0.08157917 0.04495798 -0.04137076 0.00824536 0.08498619 -0.04462177  0.045175 -0.0678696 -0.03548489 0.09398508] |

| import spacy  # Load spaCy's English model with vectors (medium size recommended for demo)  nlp = spacy.load("en\_core\_web\_md") # Make sure to install: python -m spacy download en\_core\_web\_md  # Get the embedding for 'fasttext'  doc = nlp("fasttext")  vector = doc[0].vector  print("spaCy (GloVe-style) embedding for 'fasttext':")  print(vector)  # Get the embedding for 'information'  doc = nlp("information")  vector = doc[0].vector  print("spaCy (GloVe-style) embedding for 'fasttext':")  print(vector) |
| --- |
| spaCy (GloVe-style) embedding for 'fasttext':  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.  0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.  0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.  0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.  0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.  0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.  0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.  0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.  0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.  0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.  0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.  0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.  0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]  spaCy (GloVe-style) embedding for 'fasttext':  [-8.6201e-01 2.7651e-01 -1.1209e-01 2.9973e-01 -3.2215e-01 1.6662e-01  3.2387e-01 -2.6418e-01 -9.4269e-02 2.6968e+00 -7.7937e-01 -2.2293e-01  1.3259e-01 -6.0331e-02 -5.6518e-01 -1.2697e-01 -9.6146e-04 9.5577e-01  -1.5316e-01 -2.0736e-01 5.1995e-01 2.4404e-01 4.2409e-01 -3.2843e-01  4.6414e-02 7.0281e-01 -5.6748e-02 -3.1359e-02 5.4413e-04 -2.9599e-01  -1.2435e-01 2.9698e-02 2.0202e-02 4.3861e-01 -3.5973e-01 2.7488e-01  -2.0768e-01 2.4601e-01 -9.2236e-02 -1.7892e-01 1.0122e-01 -2.2294e-02  3.3674e-01 -2.5970e-01 1.0073e-02 -1.5973e-01 2.2097e-02 1.1015e-01  1.6586e-01 -3.8621e-01 -5.5923e-01 -1.8715e-02 -1.0099e-01 -1.2273e-01  -8.5288e-02 -6.0562e-02 -3.4501e-01 2.3143e-01 7.2903e-02 -1.3168e-01  1.7436e-01 -7.8168e-02 5.2544e-02 7.1861e-01 2.0201e-01 -2.6284e-02  1.2912e-01 -9.9609e-02 -2.0977e-02 8.3424e-01 2.4343e-01 -7.6616e-02  7.5797e-01 1.1665e-01 6.8503e-01 -2.3969e-01 -1.1874e-01 -2.6285e-01  -1.2579e-01 5.7214e-02 6.2780e-02 2.3520e-01 -1.3895e-02 3.6153e-01  -2.5211e-01 -2.8394e-01 -4.2244e-01 -6.4730e-01 -3.5867e-02 4.2154e-01  3.2547e-01 -8.5423e-02 -1.4738e-01 -4.4997e-01 1.5658e-01 -5.4494e-01  -1.7536e-02 5.1365e-02 8.0250e-01 2.3768e-03 5.5978e-02 1.9406e-01  7.6418e-02 1.3540e-01 -3.7818e-01 -7.9210e-01 1.8207e-01 1.0666e-01  1.2814e-01 -2.3423e-01 -4.9533e-02 1.9053e-01 -4.3494e-01 3.7067e-01  2.1406e-01 4.5726e-01 -2.8263e-01 -2.8637e-02 2.9820e-02 -1.2408e-01  5.0389e-02 2.0739e-01 1.2662e-01 -2.4992e-01 -2.3809e-01 3.4391e-02  -5.0045e-02 -3.9001e-01 -1.6011e-01 -1.7382e-01 3.8078e-01 -2.9090e-02  -1.9657e-01 1.2721e-01 2.3616e-02 6.5187e-01 7.5107e-02 -2.7315e-01  3.1052e-01 -1.1755e-01 3.5083e-01 -1.6923e-01 4.7568e-01 -5.3413e-01  -3.6001e-02 -4.2661e-01 -1.9489e-01 3.3570e-01 -1.0289e-01 -2.7072e-01  2.8095e-01 9.9431e-02 3.3431e-01 -2.2876e-01 2.2340e-01 3.6577e-01  -1.5189e-01 5.4580e-01 -3.1151e-01 -2.2357e-01 7.4914e-02 1.4302e-01  -1.2007e-01 2.8454e-02 -6.5706e-01 2.8540e-01 -1.6415e-01 2.1825e-01  2.5792e-01 -1.9770e-01 -5.5616e-02 -3.0275e-01 -2.8658e-01 -3.0964e-01  1.8269e-02 1.4147e-01 6.3578e-01 -5.8701e-02 -4.1875e-01 7.5901e-01  -6.1809e-01 2.5919e-01 -9.1342e-02 5.0756e-02 2.3157e-01 1.7248e-01  -1.1036e-01 4.1037e-01 -4.9521e-01 -2.9590e-01 1.1619e-01 1.2180e-01  -2.2854e-01 4.5350e-01 -1.1276e-01 4.0012e-01 1.2112e-01 -1.6303e-01  -3.7259e-01 -3.8068e-01 -2.2622e-01 3.7108e-02 -1.1991e-01 1.8088e-01  1.7325e-01 -8.6585e-02 3.0349e-02 4.3789e-01 -2.4315e-01 4.1742e-01  3.2704e-01 -1.2688e-02 2.1881e-02 1.3824e-02 4.7779e-02 -8.2258e-02  -4.2826e-01 1.6992e-01 -3.0249e-01 -5.9945e-02 -2.9488e-01 -2.8473e-01  -2.8394e-01 2.8204e-01 -3.0627e-01 9.6415e-02 -3.3293e-01 -8.7120e-02  2.8508e-01 -1.1283e-01 -1.0169e-01 6.7239e-01 2.7292e-02 3.4787e-01  8.2950e-02 -6.3724e-02 -1.6224e-01 -2.3958e-01 2.1090e-01 6.0413e-01  -2.9977e-01 -3.5805e-01 1.5823e-01 -7.2733e-01 2.5410e-01 -2.5322e-01  4.7939e-01 1.5153e-02 1.4156e-01 -7.0484e-02 1.8614e-01 1.2491e-01  -2.3180e-01 -1.6349e-01 6.2412e-01 -2.0452e-01 7.8166e-02 -3.5767e-01  -9.2145e-02 -3.9466e-01 3.9195e-01 -3.7713e-02 -3.3813e-01 -1.6506e-01  3.1591e-01 -6.4982e-01 2.8108e-01 4.9543e-01 8.7389e-01 -1.6642e-01  3.2304e-02 -2.9488e-01 3.1232e-01 -1.9267e-02 2.0005e-01 -1.9114e-02  3.9732e-02 1.6810e-01 -3.1016e-01 1.0786e-01 4.9060e-01 -8.1845e-02  -2.8791e-01 1.3349e-01 -9.5110e-02 3.3505e-01 -2.0050e-02 1.3483e-01  -2.6453e-02 -1.5750e-01 -2.5856e-01 5.6588e-01 6.2737e-01 -4.3038e-03  2.6125e-01 -6.3719e-01 -5.4943e-01 -2.9787e-02 3.0704e-01 -1.9168e-02] |

| FastText embedding for 'fasttext': | Word2Vec embedding for 'fasttext': |  |  |
| --- | --- | --- | --- |
| [  0.00906787  -0.00530815  -0.00275941  0.00349125  -0.00242799  -0.01152946  0.02509905  -0.00598241  0.0041849  0.00632296  ] | [  -0.037098  -0.08745642  0.05437467  0.06509756  -0.0078755  -0.06709856  -0.07085925  -0.0249706  0.05143254  -0.03665238  ] |  |  |

| source code:using spacy |
| --- |
| import spacy  # Load the pre-trained FastText model (en\_core\_web\_md has FastText embeddings)  nlp = spacy.load("en\_core\_web\_md")  # Example function to get word vector using FastText embeddings  def get\_word\_vectors(text):  doc = nlp(text)  # Return the average word vector of all tokens in the document  return doc.vector  # Apply the function to your cleaned\_text column  df['word\_vector\_fasttext'] = df['cleaned\_text'].apply(get\_word\_vectors)  # Display the DataFrame with word vectors from FastText  print(df[['cleaned\_text', 'word\_vector\_fasttext']]) |
| output |
| cleaned\_text \  0 text embedding crucial nlp tasks  1 machine learning transforming industries  2 deep learning subset machine learning  3 natural language processing deals understandin...  4 text data represented using various embedding ...  word\_vector\_fasttext  0 [-0.732738, 0.20866199, -0.20861883, 0.0553138...  1 [-0.7503575, 0.12703075, -0.020711988, -0.1824...  2 [-0.767366, 0.21592982, -0.15092558, -0.065373...  3 [-0.6814814, -0.22555415, -0.25004816, -0.0562...  4 [-0.77561283, 0.13406001, -0.15341485, -0.1082... |

### **4. Analysis of the FastText Model**

FastText embeddings capture semantic meanings by considering character n-grams, allowing the model to generate meaningful vectors for rare words or even words not seen during training.

#### **Example:**

* **Out-of-Vocabulary Words:** If a word like "embbedding" appears, FastText can generate a meaningful embedding by looking at the subword (character n-grams), whereas Word2Vec would treat it as an unknown word.

#### **Vocabulary Representation:**

| **Word** | **Vector (first 3 components)** |
| --- | --- |
| embedding | [0.215, -0.334, 0.427] |
| crucial | [0.124, -0.278, 0.395] |
| machine | [0.142, -0.261, 0.334] |

### **5. Interpretation of Results**

* **Subword Representation:** FastText captures character-level patterns, making it more robust to rare words and out-of-vocabulary terms.
* **Improved Handling of Morphologically Rich Languages:** FastText performs better for languages with extensive use of prefixes, suffixes, and other morphological changes.

### **6. Visualizations**

#### **FastText Embedding Visualization (using PCA for dimensionality reduction)**

| source code: |
| --- |
| import matplotlib.pyplot as plt  from sklearn.decomposition import PCA  # Get FastText word vectors from the DataFrame  word\_vectors\_fasttext = df['word\_vector\_fasttext'].tolist()  texts = df['cleaned\_text'].tolist()  # Reduce the dimensionality to 2D using PCA for visualization  pca = PCA(n\_components=2)  pca\_result\_fasttext = pca.fit\_transform(word\_vectors\_fasttext)  # Plot the word embeddings in 2D space  plt.figure(figsize=(12, 8))  plt.scatter(pca\_result\_fasttext[:, 0], pca\_result\_fasttext[:, 1])  # Annotate each point with its corresponding cleaned text  for i, text in enumerate(texts):  plt.annotate(text, (pca\_result\_fasttext[i, 0], pca\_result\_fasttext[i, 1]),  textcoords="offset points", xytext=(0, 5), ha='center', fontsize=10)  plt.title('Visualization of FastText Word Embeddings (2D)')  plt.show() |
| output |
|  |

### **7. Strengths and Limitations**

#### **Strengths:**

* Handles out-of-vocabulary (OOV) words by using subword information.
* More accurate for morphologically rich languages, as it considers character-level patterns.

#### **Limitations:**

* Requires more memory compared to Word2Vec due to the inclusion of subword information.
* Training can be slow, especially with large datasets.

## **Conclusion: FastText**

FastText is an extension of Word2Vec that incorporates character-level n-grams to generate embeddings. It excels at handling rare and unseen words and works well for languages with rich morphology. It is a great option when working with noisy or incomplete text data but requires more resources than Word2Vec.

### **Summary of All Embedding Techniques**

1. **Bag of Words (BoW):** Simple and easy to implement, but loses context and semantics.
2. **TF-IDF:** Weighs terms based on frequency, capturing word importance but ignoring word relationships.
3. **Word2Vec:** Learns word embeddings based on surrounding words and captures semantic relationships.
4. **GloVe:** Uses a global co-occurrence matrix to capture both local and global word relationships.
5. **FastText:** Extends Word2Vec by representing words as n-grams, allowing for better handling of rare or unseen words.

These techniques are foundational in natural language processing and are useful for various applications such as document classification, sentiment analysis, and machine translation.

### **Summary of Techniques:**

| **Technique** | **Input Data** | **Output Type** | **Example Feature** |
| --- | --- | --- | --- |
| BoW | Text corpus | Sparse matrix | [1, 0, 1, 0] |
| TF-IDF | Text corpus | Weighted matrix | [0.45, 0.58] |
| Word2Vec | Text corpus | Dense vector | [0.1, 0.2, -0.1, 0.3] |
| GloVe | Text corpus | Dense vector | [0.2, -0.3, 0.4, 0.1] |
| FastText | Text corpus | Dense vector (subword-based) | [0.1, -0.2, 0.3, -0.4] |

| **Embedding Technique** | **Libraries** | **Recommended Library** | **Description** |
| --- | --- | --- | --- |
| **Bag of Words (BoW)** | scikit-learn (CountVectorizer), NLTK,  spaCy | scikit-learn | Represents text as a sparse matrix of word counts, useful for simple document classification tasks. |
| **TF-IDF** | scikit-learn (TfidfVectorizer), NLTK | scikit-learn | Weighs words based on their frequency in a document and inverse frequency across all documents, highlighting important terms. |
| **Word2Vec** | gensim,  TensorFlow,  PyTorch | gensim | Converts words into continuous vector representations based on context in large corpora, capturing semantic relationships. |
| **GloVe** | glove-python-binary,  gensim,  spaCy | glove-python-binary | Pre-trained word vectors trained on a global context, capturing semantic and syntactic word similarities. |
| **FastText** | gensim.models.FastText, facebookresearch/fastText | gensim | Extends Word2Vec by considering subword information, making it effective for morphologically rich languages. |

a table listing various **NLP libraries** and their **roles**:

| **Library** | **Role** |
| --- | --- |
| **NLTK (Natural Language Toolkit)** | Provides tools for text preprocessing, such as tokenization, stopword removal, lemmatization, stemming, POS tagging, and more. Used for basic NLP tasks and text cleaning. |
| **spaCy** | A powerful NLP library for industrial-strength tasks like named entity recognition (NER), part-of-speech tagging, dependency parsing, and word vectors. It is optimized for speed and ease of use in production. |
| **scikit-learn** | Provides utilities for text vectorization (e.g., **BoW**, **TF-IDF**), machine learning models, and data preprocessing tasks. Often used in conjunction with other NLP libraries for feature extraction and classification tasks. |
| **gensim** | Specialized in unsupervised learning for topic modeling, word embeddings (e.g., **Word2Vec**, **FastText**), and similarity analysis. It is widely used for working with large text corpora and vector space models. |
| **transformers (by Hugging Face)** | Provides pre-trained models like **BERT**, **GPT**, **RoBERTa**, **T5**, etc., for advanced NLP tasks like question answering, text classification, and text generation. It is used for state-of-the-art transformer-based embeddings. |
| **TensorFlow** | A deep learning framework that offers tools for building custom NLP models, including sequence models (e.g., RNN, LSTM, GRU), transformers, and language models. It also supports integration with **BERT** and other transformer models. |
| **PyTorch** | Similar to TensorFlow, used for building and training neural networks, including NLP models. It supports a variety of NLP tasks, and works well with **transformers** and custom deep learning models. |
| **Flair** | An NLP library built on top of PyTorch, focused on contextual word embeddings (e.g., **ELMo**, **BERT**), sequence tagging (e.g., NER, POS), and text classification tasks. |
| **TextBlob** | A simple NLP library built on NLTK and Pattern, useful for beginners. It provides basic functionalities like part-of-speech tagging, noun phrase extraction, translation, and sentiment analysis. |
| **Pattern** | A lightweight NLP and machine learning library for Python that provides basic tools for text classification, part-of-speech tagging, and sentiment analysis. |
| **fastText** | Developed by Facebook, it's used for learning word representations (embeddings) and text classification, especially for languages with rich morphology. It's efficient for training on large datasets. |
| **Stanford NLP** | A suite of NLP tools developed by Stanford University, providing tools for syntactic parsing, named entity recognition (NER), and part-of-speech tagging. |
| **CoreNLP** | Another Stanford NLP tool that can perform various NLP tasks such as tokenization, part-of-speech tagging, NER, dependency parsing, and more. |
| **AllenNLP** | Built on top of PyTorch, this library is designed to research, build, and evaluate state-of-the-art deep learning models for NLP tasks such as reading comprehension, textual entailment, and semantic role labeling. |
| **OpenNLP** | An Apache project that provides machine learning-based libraries for various NLP tasks such as tokenization, sentence splitting, POS tagging, NER, and parsing. |
| **Keras** | A high-level neural networks API, often used for building and training NLP models (especially for text classification, sentiment analysis, etc.) using TensorFlow or Theano as a backend. |
| **BERT (via Hugging Face)** | A pre-trained transformer model for a variety of NLP tasks, fine-tuned for specific tasks like text classification, sentiment analysis, question answering, and named entity recognition (NER). |
| **spaCy-transformers** | Combines spaCy's fast pipeline with transformers for more accurate NLP models. It integrates transformers (e.g., **BERT**) for tasks like NER and text classification. |

These libraries cover a wide range of tasks, from text preprocessing to advanced deep learning models for complex NLP applications.

In the context of **text embedding techniques**, **NLTK** (Natural Language Toolkit) is primarily used for **text preprocessing** rather than directly generating embeddings. Here's how it plays a role:

1. **Preprocessing Text for BoW and TF-IDF**:  
   * **Tokenization**: NLTK provides functions to split text into words or tokens, a fundamental step before creating embeddings like **BoW** or **TF-IDF**.
   * **Stopword Removal**: It helps remove common words (like "and", "the", "is") that don't contribute much to the meaning of the text.
   * **Stemming and Lemmatization**: NLTK can reduce words to their root form (e.g., "running" to "run"), helping normalize the text before creating embeddings.
   * **POS Tagging**: NLTK also offers part-of-speech tagging, which can be helpful in understanding word relationships.
2. **Feature Extraction for Embedding Models**:  
   * **BoW & TF-IDF**: While libraries like **scikit-learn** handle the embedding creation, NLTK is often used to preprocess the raw text before feeding it into the **CountVectorizer** (for BoW) or **TfidfVectorizer** (for TF-IDF).

### **Example Usage of NLTK in Text Preprocessing:**

For **BoW** or **TF-IDF**, you might first use NLTK for tokenizing and removing stopwords, and then pass the cleaned text to **scikit-learn** for creating the vector representation.

python

import nltk

from nltk.corpus import stopwords

from nltk.tokenize import word\_tokenize

from sklearn.feature\_extraction.text import CountVectorizer

# Sample text

text = "NLTK is a leading platform for building Python programs to work with human language data."

# Download stopwords (only need to do once)

nltk.download('stopwords')

nltk.download('punkt')

# Tokenize and remove stopwords

tokens = word\_tokenize(text.lower())

filtered\_tokens = [word for word in tokens if word not in stopwords.words('english')]

# Create Bag of Words (BoW) model using CountVectorizer

vectorizer = CountVectorizer()

X = vectorizer.fit\_transform([' '.join(filtered\_tokens)])

# Display the result

print(X.toarray())

**Summary**:

* NLTK's role is in the **text cleaning and preprocessing** stage.
* After preprocessing (like tokenization, stopword removal, etc.), the processed text can be passed to libraries like **scikit-learn** or **gensim** to generate the actual embeddings (BoW, TF-IDF, etc.).

—-----------------------------------------------------------------------------------------

does spacy do different embedding techniques? if so can you list all embedding which spacy can do

###### **ChatGPT said:**

Yes, spaCy supports different word embedding techniques. These embeddings are used to represent words as vectors in a continuous vector space, which helps in capturing the semantic meaning of words.

Here are the main types of embeddings that spaCy supports:

### **1. Word2Vec**

* **Description**: Word2Vec is a popular word embedding technique that learns vector representations of words in a way that words with similar meanings are mapped to nearby points in the vector space. It's based on either the Continuous Bag of Words (CBOW) or the Skip-Gram model.
* **Support in spaCy**: spaCy can load pre-trained Word2Vec embeddings, and you can use it in your pipelines.

### **2. GloVe (Global Vectors for Word Representation)**

* **Description**: GloVe is another popular word embedding technique that captures semantic meaning by looking at word co-occurrence statistics over a large corpus.
* **Support in spaCy**: You can load pre-trained GloVe vectors into spaCy by using the spacy-glove package or similar pre-trained models available in spaCy.

### **3. FastText**

* **Description**: FastText, developed by Facebook, is an extension of Word2Vec that also considers subword information (such as character n-grams) in the embeddings. This helps it handle out-of-vocabulary words more effectively.
* **Support in spaCy**: spaCy supports FastText embeddings through the spacy-fasttext package or using pre-trained FastText models.

### **4. Transformer-based Embeddings (Contextualized Embeddings)**

* **Description**: Transformer models like BERT, GPT, and others provide contextualized embeddings, where the embedding of a word depends on the context in which it appears (unlike traditional static word embeddings like Word2Vec and GloVe).
* **Support in spaCy**: spaCy integrates with transformer-based models like BERT, RoBERTa, and other Hugging Face transformers through the spacy-transformers package. These models provide contextualized word embeddings.

### **5. ELMo (Embeddings from Language Models)**

* **Description**: ELMo embeddings are also contextualized word embeddings, but instead of directly using transformers, it uses a bidirectional LSTM (Long Short-Term Memory) trained on a language model.
* **Support in spaCy**: spaCy doesn't have native support for ELMo, but it can be integrated into spaCy pipelines using external packages like spacy-elmo.

### **6. Doc2Vec (Paragraph Vectors)**

* **Description**: While not directly a word embedding, Doc2Vec is used for generating embeddings for larger bodies of text like sentences, paragraphs, or documents. It's an extension of Word2Vec that learns embeddings for entire documents.
* **Support in spaCy**: spaCy doesn't have built-in support for Doc2Vec, but you can integrate external libraries like gensim to generate Doc2Vec embeddings and use them in spaCy.

### **7. Transformers via Hugging Face (BERT, RoBERTa, GPT, etc.)**

* **Description**: These are large pre-trained transformer models that can be used to get contextual embeddings. They represent words in context, which means the same word might have different embeddings depending on the surrounding text.
* **Support in spaCy**: spaCy supports the use of Hugging Face transformers through the spacy-transformers extension, enabling you to use BERT, RoBERTa, and similar models for contextual embeddings.

### **8. Custom Embeddings (Training Your Own)**

* **Description**: In addition to pre-trained embeddings, spaCy allows you to train your own word embeddings using your own corpus.
* **Support in spaCy**: You can use spaCy’s training functionality to train word vectors (e.g., Word2Vec, GloVe) from scratch using the spacy training API.

### **9. Lexeme-based Embeddings**

* **Description**: spaCy also stores word embeddings at the lexeme level, which means it attaches embeddings directly to the lexemes in the vocabulary.
* **Support in spaCy**: These embeddings are part of the default word vectors spaCy loads with its language models (e.g., en\_core\_web\_md).

Each of these embeddings serves a different purpose depending on the task you're working on. For tasks like semantic similarity, document classification, or named entity recognition, transformer-based or contextual embeddings (BERT, RoBERTa) typically work better than static embeddings like Word2Vec and GloVe.

—------

great. give me spacy code example for each technique. give only for these techniques : Word2Vec, GloVe, and FastText. include all the necessary libraries.

###### **ChatGPT said:**

Here are code examples for using **Word2Vec**, **GloVe**, and **FastText** embeddings in spaCy, along with the necessary libraries:

### **1. Word2Vec Embedding**

To use Word2Vec with spaCy, you'll need to load a pre-trained model or use the spacy package that provides access to these embeddings.

python

Copy

import spacy

from spacy.lang.en import English

# Load the pre-trained Word2Vec model from spaCy

# You can also use `en\_core\_web\_md` or `en\_core\_web\_lg` models for Word2Vec embeddings.

nlp = spacy.load("en\_core\_web\_md") # Or en\_core\_web\_lg for more detailed embeddings

# Example of getting Word2Vec embeddings for a word

word = "king"

token = nlp(word)

# Access the word vector

print(f"Word2Vec embedding for '{word}': {token.vector}")

* en\_core\_web\_md and en\_core\_web\_lg come with Word2Vec embeddings pre-loaded.

### **2. GloVe Embedding**

To use GloVe embeddings with spaCy, you can use the spacy package and load pre-trained GloVe embeddings as part of the model.

python

Copy

import spacy

# Load a pre-trained GloVe model

nlp = spacy.load("en\_core\_web\_md") # This comes with GloVe embeddings

# Example of getting GloVe embeddings for a word

word = "computer"

token = nlp(word)

# Access the GloVe embedding

print(f"GloVe embedding for '{word}': {token.vector}")

* en\_core\_web\_md and en\_core\_web\_lg use GloVe embeddings, so you can directly access them after loading the model.

### **3. FastText Embedding**

You can load pre-trained FastText embeddings into spaCy using the spacy-fasttext package.

Install the spacy-fasttext package:  
  
 bash  
Copy  
pip install spacy-fasttext

1. Load the FastText model into spaCy:

python

Copy

import spacy

# Load the pre-trained FastText model

# You'll need to install the spacy-fasttext package to enable FastText integration.

nlp = spacy.load("xx\_ent\_wiki\_sm") # This model has FastText embeddings

# Example of getting FastText embeddings for a word

word = "apple"

token = nlp(word)

# Access the FastText embedding

print(f"FastText embedding for '{word}': {token.vector}")

* xx\_ent\_wiki\_sm is a multi-language model in spaCy that integrates FastText embeddings. If you have specific language needs, you can use models like xx\_ent\_wiki\_lg for better accuracy.

### **Explanation of the Code:**

1. **Word2Vec**: nlp = spacy.load("en\_core\_web\_md") loads a model with Word2Vec embeddings. After loading, you can access word vectors with token.vector.
2. **GloVe**: Similar to Word2Vec, GloVe embeddings are accessed via the en\_core\_web\_md or en\_core\_web\_lg models in spaCy. You get the embeddings using token.vector.
3. **FastText**: FastText embeddings can be loaded by installing the spacy-fasttext extension. You can use models like xx\_ent\_wiki\_sm to access FastText embeddings for any word.

By using these techniques, you can easily retrieve word embeddings in spaCy for your NLP tasks.

##### **You said:**

i see code same for word2vec and Glove

###### **ChatGPT said:**

Yes, you're correct! The code for **Word2Vec** and **GloVe** in spaCy appears very similar because both types of embeddings are loaded into the model in the same way. Both en\_core\_web\_md and en\_core\_web\_lg come with pre-trained **Word2Vec** and **GloVe** embeddings bundled together in these models.

Here’s a more detailed explanation:

### **Why the Code Looks Similar for Word2Vec and GloVe:**

* **GloVe and Word2Vec are both static word embeddings**, meaning they are fixed vectors for each word (they don't change depending on context).
* **spaCy models like en\_core\_web\_md** (medium-sized model) and en\_core\_web\_lg (large-sized model) both use **Word2Vec** and **GloVe** embeddings.
* The difference is mainly in how the embeddings are trained:  
  + **Word2Vec** embeddings are trained using either the Continuous Bag of Words (CBOW) model or the Skip-Gram model.
  + **GloVe** embeddings are trained using the Global Vectors for Word Representation model, which uses word co-occurrence information from a corpus.

Both of these embeddings are accessed the same way in spaCy through the token.vector attribute.

### **If You Want to Differentiate Word2Vec and GloVe:**

* **Word2Vec** tends to focus on the local context of words (e.g., surrounding words in a window), while **GloVe** focuses on global co-occurrence statistics.

However, **spaCy doesn't explicitly differentiate** between the two when loading the models, so it’s hard to point out directly within the code.

The only way to **truly differentiate between Word2Vec and GloVe** in spaCy would be to look at the model documentation (e.g., the pre-trained vectors included in en\_core\_web\_md and en\_core\_web\_lg). But for the purpose of practical use, they can be treated similarly in spaCy code.

### **Key Point:**

* **Code remains the same** for both **Word2Vec** and **GloVe** because spaCy integrates these embeddings as part of its language models (en\_core\_web\_md, en\_core\_web\_lg), so the embeddings are loaded and accessed in the same manner.

okay. now help in understanding spayc, different embeding techniques and how they are connected. how genism, nltkm connected here. give me overall pictures on all these lib and how they are connected. only high level picture and not full details

###### **ChatGPT said:**

Here's a high-level overview of **spaCy**, different embedding techniques, and how they connect with other libraries like **Gensim** and **NLTK**:

### **1. spaCy:**

* **Purpose**: spaCy is a powerful and fast NLP library designed for production use. It provides pre-trained models for various NLP tasks like tokenization, POS tagging, named entity recognition (NER), and word embeddings.
* **Word Embeddings**: spaCy comes with support for different embedding techniques (Word2Vec, GloVe, FastText, Transformer-based embeddings).  
  + **Word2Vec** and **GloVe** embeddings are pre-trained and bundled with spaCy models (en\_core\_web\_md, en\_core\_web\_lg, etc.).
  + **FastText** embeddings can be used with the spacy-fasttext extension.
  + **Transformers** (like BERT, GPT) are supported through spacy-transformers, providing **contextual embeddings**.

### **2. Word Embeddings in spaCy:**

* **Static Embeddings** (Word2Vec, GloVe, FastText):  
  + These embeddings represent words in vector space, where semantically similar words have similar vectors.
  + **Word2Vec** and **GloVe** are both static embeddings but differ in how they're trained (local context for Word2Vec vs. global co-occurrence for GloVe).
  + **FastText** improves over Word2Vec by considering subword (character n-grams), which helps handle out-of-vocabulary words.
* **Contextual Embeddings** (via Transformers):  
  + **Transformers** like BERT provide embeddings that change based on the context of the word in a sentence. These embeddings are dynamic and provide richer semantic representations compared to static embeddings.

### **3. Gensim:**

* **Purpose**: Gensim is another popular library, mainly used for topic modeling and document similarity. It's known for **Word2Vec** (and other embeddings) training and handling large corpora efficiently.
* **Connection with spaCy**:  
  + Gensim can be used to **train Word2Vec** and other embeddings on your own corpus.
  + Gensim embeddings (like Word2Vec or Doc2Vec) can be loaded into **spaCy** for use in downstream tasks (like NER or text classification) by using the spacy-transformers or spacy-fasttext integrations.

### **4. NLTK (Natural Language Toolkit):**

* **Purpose**: NLTK is a widely used library for educational and research purposes in NLP. It provides tools for tokenization, POS tagging, text processing, and access to corpora.
* **Connection with spaCy**:  
  + NLTK is more focused on **text processing and analysis** (e.g., tokenizing text, working with corpora), while spaCy is more focused on **efficient production-grade NLP tasks**.
  + NLTK's embeddings (if used) are often used in conjunction with other libraries (like Gensim or spaCy) to enhance functionality.

### **High-Level Connections Between These Libraries:**

pgsql

CopyEdit

+-------------------+ +---------------------+ +------------------------+

| spaCy | | Gensim | | NLTK |

|-------------------| |---------------------| |------------------------|

| - Fast and production-ready NLP tasks (Tokenization, POS, NER, etc.) | | - Word2Vec/Doc2Vec embedding training and usage | | - Text processing, tokenization, and educational tools |

| - Pre-trained models (Word2Vec, GloVe, FastText, Transformers) | | - Handles large corpora efficiently | | - Supports basic NLP tasks, and access to corpora |

| - Embedding access via `token.vector` | | - Can be used with spaCy embeddings | | - Typically used with other libraries like spaCy |

+-------------------+ +---------------------+ +------------------------+

| | |

+-------------------+------------------+-------------------+

|

+----------+------------+

| Word Embedding Models |

|------------------------|

| - Word2Vec |

| - GloVe |

| - FastText |

| - Transformers |

+------------------------+

### **Key Points:**

* **spaCy** is the production-oriented NLP library that comes with pre-trained embeddings (Word2Vec, GloVe, FastText, Transformers) and offers fast processing for NLP tasks.
* **Gensim** specializes in training embeddings (like Word2Vec) and handling large corpora for topic modeling and document similarity. You can integrate Gensim embeddings with spaCy for advanced NLP tasks.
* **NLTK** is a comprehensive library for NLP research and education, with tools for text processing, tokenization, and access to corpora. It doesn’t focus on embeddings as much as spaCy and Gensim do but can be used in conjunction with them.

### **How They Work Together:**

* **spaCy** and **Gensim** can be used together to load pre-trained embeddings or train your own embeddings (using Gensim), and then use them in spaCy's pipeline.
* **spaCy** and **NLTK** can be used together for pre-processing text (tokenization, stop word removal) and using spaCy's NLP capabilities for more advanced tasks.
* **Gensim** embeddings can be **loaded into spaCy** models to enhance tasks like word similarity or text classification.

This high-level diagram shows how these libraries connect in the broader NLP ecosystem and how they can complement each other.

—------------

out of these 5 below embedding techniques which has predefined corpus and which is not. explain in table format

Bag of Words (BoW), TF-IDF, Word2Vec, GloVe, and FastText.

out of these 5 below embedding techniques which has predefined corpus and which is not. explain in table format Bag of Words (BoW), TF-IDF, Word2Vec, GloVe, and FastText.

Here’s a clear comparison of the five embedding techniques regarding whether they rely on a predefined corpus (e.g., available as pre-trained vectors) or not. “Predefined corpus” here means you can directly use pre-trained embeddings learned from large, external datasets, without needing to train on your own data. “Not predefined” means the representations are generally built fresh from your own data.

| **Technique** | **Needs Predefined Corpus?** | **Pre-trained Options Common?** | **Explanation** |
| --- | --- | --- | --- |
| Bag of Words (BoW) | No | No | BoW is constructed from your own dataset, simply counting word occurrences. |
| TF-IDF | No | No | TF-IDF vectors are generated from your own text corpus, weighting terms based on frequency and uniqueness in your dataset. |
| Word2Vec | Optional | Yes | You can train on your own corpus or use available pre-trained embeddings (e.g., Google News). |
| GloVe | Yes | Yes | Usually used with widely available pre-trained embeddings (e.g., Wikipedia, Common Crawl); you can train your own, but it's less common. |
| FastText | Yes | Yes | Pre-trained FastText embeddings are available (e.g., from Facebook or other public datasets), but custom training is also possible. |

Key points:

* BoW and TF-IDF don’t use any external/predefined corpus—they are always built from scratch on your provided data.
* Word2Vec, GloVe, and FastText all have pre-trained vectors available for many languages and domains, and can also be trained on your own corpus if needed.
* Of these three, GloVe and FastText models are often used directly in their pre-trained form. Word2Vec is commonly both pre-trained and custom-trained, depending on availability and task.

**Study materials/References**

<https://medium.com/analytics-vidhya/fundamentals-of-bag-of-words-and-tf-idf-9846d301ff22>

<https://medium.com/analytics-vidhya/deep-dive-into-word2vec-7fcefa765c17>

<https://towardsdatascience.com/attention-is-all-you-need-discovering-the-transformer-paper-73e5ff5e0634/>

<https://medium.com/@mervebdurna/advanced-word-embeddings-word2vec-glove-and-fasttext-26e546ffedbd>

<https://sainivedh.medium.com/introduction-to-glove-embeddings-9f57d48d0ce4>