FAQ - Word Embeddings

[Previous](https://olympus.mygreatlearning.com/courses/111352/modules/items/5520638?pb_id=17388)

[Next](https://olympus.mygreatlearning.com/courses/111352/modules/items/5520641?pb_id=17388)

#### **1. What are the three different Word2Vec models, and how do they differ?**

Three different Word2Vec models are as follows:

**1. Continuous Bag of Words (CBOW)**

* CBOW is a neural network architecture for word embeddings where the model predicts the target word based on its surrounding context words. It takes the average of the word vectors in the context and attempts to maximize the probability of predicting the target word.

**2. Skip-gram**

* Skip-gram is a neural network architecture that works in the reverse way compared to CBOW. It predicts context words given a target word. By capturing the relationships between a target word and its context words, Skip-gram is particularly effective in scenarios where context is sparse. It excels at capturing fine-grained semantic relationships and is often preferred when working with smaller datasets or in applications where detailed semantic information is crucial.

Both COW and Skip-gram model uses the softmax function to compute the probability distribution over the entire vocabulary, making the training computationally expensive.

**3. Negative Sampling**

* Negative Sampling is a technique used in word embedding training to address computational efficiency issues, especially when dealing with large vocabularies. Instead of predicting the actual context words, Negative Sampling transforms the task into a binary classification problem. It randomly samples a small set of negative (non-context) words for each training instance, and the model is trained to distinguish true context words from these negatives. This approach significantly reduces the computational cost compared to traditional softmax-based training (CBOW and Skip-gram), making it suitable for large-scale datasets and vocabularies.

#### **2. What is the difference between Word2Vec and GloVe models?**

Word2Vec and GloVe are popular techniques used in natural language processing (NLP) to represent words as vectors in a continuous vector space. They are both used for word embedding, a process that converts words into numerical vectors while preserving their semantic relationships. However, they have some differences in their approaches:

**Training Approach:**

* **Word2Vec:** Word2Vec uses a neural network to predict the probability of a word given its context (Continuous Bag of Words, CBOW) or the probability of the context given a word (Skip-gram).
* **GloVe (Global Vectors for Word Representation):** GloVe, on the other hand, focuses on the global statistics of the corpus. It constructs a co-occurrence matrix for words based on the frequency of their co-occurrence in a given context window.

**Context Window:**

* **Word2Vec:** In Word2Vec, the context window is a parameter that defines the number of words considered as context for predicting the target word.
* **GloVe:** GloVe does not explicitly use a context window. Instead, it builds a global co-occurrence matrix that represents the overall word co-occurrence statistics.

**Training Efficiency:**

* **Word2Vec**: It employs a neural network architecture (either Skip-gram or CBOW), which involves learning a large number of parameters. The complexity of this network increases with the size of the vocabulary. Training the neural network requires iterative optimization, and multiple passes through the dataset are often necessary for convergence. These factors contribute to the computational expense, particularly for large vocabularies where the number of parameters and the amount of data to process are substantial.
* **GloVe**: GloVe, in contrast, takes a different approach by formulating the word embedding task as a matrix factorization problem based on global word co-occurrence statistics. This matrix factorization is computationally efficient compared to training a neural network. It simplifies the training process and can be parallelized effectively. The use of global co-occurrence statistics allows GloVe to capture semantic relationships without the need for a complex neural network architecture. As a result, the training of GloVe models is often considered more efficient, especially for large datasets and vocabularies.

#### **3. What is cosine similarity? How to interpret it in the context of NLP?**

Cosine similarity is a metric used to measure the similarity between two vectors by calculating the cosine of the angle between them. In the context of NLP (Natural Language Processing), cosine similarity is often employed to quantify the similarity between two text documents or between the vector representations of words.

**Calculation:**

* Cosine similarity is calculated using the dot product of two vectors divided by the product of their magnitudes. For two vectors A and B, the cosine similarity (cosθ) is given by the formula

LaTeX: \text{cosine similarity }(A, B) = \frac{A \cdot B}{||A|| \cdot ||B||}

**Interpretation in NLP:**

* **Similarity Measure:** A higher cosine similarity indicates that the vectors are more similar, while a lower cosine similarity suggests greater dissimilarity.
* **Range:** The cosine similarity ranges from -1 to 1. A similarity of 1 implies that the vectors are identical, 0 indicates orthogonality (no similarity), and -1 implies complete dissimilarity with opposite directions.

**Application:**

* **Document Similarity:** In document analysis, cosine similarity is used to determine how similar two documents are based on the words they contain.
* **Word Embeddings:** In the context of word embeddings (e.g., Word2Vec or GloVe), cosine similarity measures the semantic similarity between words based on their vector representations. Words with similar meanings tend to have higher cosine similarities.

#### **4. How do the average\_vectorizer functions work?**

**Word2vec**

The function is defined as follows:

def average\_vectorizer\_Word2Vec(doc):

# Initializing a feature vector for the sentence

feature\_vector = np.zeros((vec\_size,), dtype="float64")

# Creating a list of words in the sentence that are present in the model vocabulary

words\_in\_vocab = [word for word in doc.split() if word in words]

# adding the vector representations of the words

for word in words\_in\_vocab:

feature\_vector += np.array(word\_vector\_dict[word])

# Dividing by the number of words to get the average vector

if len(words\_in\_vocab) != 0:

feature\_vector /= len(words\_in\_vocab)

return feature\_vector

The average\_vectorizer\_Word2Vec function is designed to generate an average word vector representation for a given document using pre-trained Word2Vec word embeddings. Here's a breakdown of how the function works:

* **Initialization:** 
  + **feature\_vector**: A NumPy array initialized as a zero vector with a specified size (vec\_size). This array will be used to store the cumulative vector representation of words in the document.
* **Words in Vocabulary:** 
  + **words\_in\_vocab**: A list comprehension iterates through each word in the input document (doc.split()), checking if the word is present in the pre-trained Word2Vec model's vocabulary (words).
* **Vector Addition:** For each word in the document that is present in the Word2Vec model's vocabulary, the function adds the corresponding vector representation to the feature\_vector using np.array(word\_vector\_dict[word]). This step aggregates the vector representations of all the words in the document.
* **Normalization (Averaging):** After summing up the vectors, the function checks if there are words in the vocabulary to avoid division by zero. If there are words in the vocabulary (len(words\_in\_vocab) != 0), it divides the feature\_vector by the number of words in the vocabulary. This step normalizes the cumulative vector representation to obtain the average vector.
* **Return:** The resulting normalized feature\_vector is then returned as the average word vector representation for the input document.

**GloVe**

The function is defined as follows:

def average\_vectorizer\_GloVe(doc):

# Initializing a feature vector for the sentence

feature\_vector = np.zeros((vec\_size,), dtype="float64")

# Creating a list of words in the sentence that are present in the model vocabulary

words\_in\_vocab = [word for word in doc.split() if word in glove\_words]

# adding the vector representations of the words

for word in words\_in\_vocab:

feature\_vector += np.array(glove\_word\_vector\_dict[word])

# Dividing by the number of words to get the average vector

if len(words\_in\_vocab) != 0:

feature\_vector /= len(words\_in\_vocab)

return feature\_vector

The average\_vectorizer\_GloVe function is similar to the average\_vectorizer\_Word2Vec function but is specifically tailored for generating an average word vector representation using pre-trained GloVe (Global Vectors for Word Representation) word embeddings. Let's break down how this function works:

* **Initialization:**
  + **feature\_vector**: A NumPy array initialized as a zero vector with a specified size (vec\_size). This array will be used to store the cumulative vector representation of words in the document.
* **Words in Vocabulary:**
  + words\_in\_vocab: A list comprehension iterates through each word in the input document (doc.split()), checking if the word is present in the pre-trained GloVe model's vocabulary (glove\_words).
* **Vector Addition:** For each word in the document that is present in the GloVe model's vocabulary, the function adds the corresponding vector representation to the feature\_vector using np.array(glove\_word\_vector\_dict[word]). This step aggregates the vector representations of all the words in the document.
* **Normalization (Averaging):** After summing up the vectors, the function checks if there are words in the vocabulary to avoid division by zero. If there are words in the vocabulary (len(words\_in\_vocab) != 0), it divides the feature\_vector by the number of words in the vocabulary. This step normalizes the cumulative vector representation to obtain the average vector.
* **Return:** The resulting normalized feature\_vector is then returned as the average word vector representation for the input document.