Published in Towards Data Science

Rabeh Ayari, PhD Follow
Mar 2, 2020 · 9 min read ★ · Listen

NLP: Word Embedding Techniques Demystified

Bag-Of-Words vs TF-IDF vs Word2Vec vs Doc2Vec vs Doc2VecC

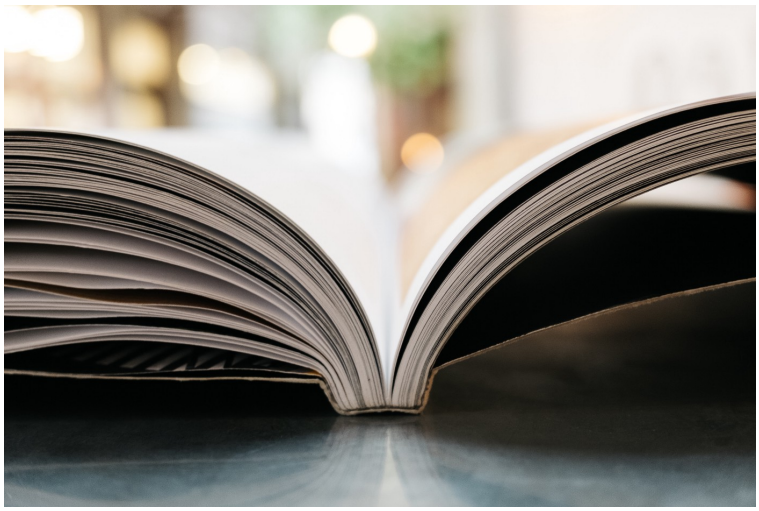

Photo by Jonas Jacobsson on Unsplash

Word Embedding is a technique of word representation that allows words with similar meaning to be understood by machine learning algorithms. Technically speaking, it is a mapping of words into vectors of real numbers using the neural network, probabilistic model, or dimension reduction on word co-occurrence matrix.

Word embedding can be learned using a variety of language models. For example, a ***'dog'*** will be represented by the vector ***[0.75, 0.22, 0.66, 0.97]***. If all the words in a dictionary are encoded in such a way, it becomes possible to compare the vectors of the words with each other, for example by measuring the *cosine distance or the euclidean distance* between the vectors. A good representation of words will then make it possible to find that the word ***'pet'*** is closer to the word ***'dog'*** than it is to the word ***'soccer'*** or **'engine'**. Therefore, these representations allow us to hope that, in the vector space where the embedding is made, we will have the equation $\boldsymbol{king - man + woman = queen}$ or even the equation $\boldsymbol{London - England + Italy = Rome}$.

Word embeddings are also very useful in mitigating the curse of dimensionality, a very recurring problem in artificial intelligence. Without word embedding, the unique identifiers representing the words generate scattered data, isolated points in a vast sparse representation. With word embedding, on the other hand, the space becomes much more limited in terms of dimensionality with a widely richer amount of semantic information. With such numerical features, it is easier for a computer to perform different mathematical operations like matrix factorization, dot product, etc, which are

article, we will be covering: B*ag-Of-Words, TF-IDF, Word2Vec, Doc2vec and Doc2vecC.*

1. Bag-of-Words

Bag-Of-Words (*a.k.a.* BOW) is a popular basic approach to generate document representation. A text is represented as a bag containing plenty of words. The grammar and word order are neglected while the frequency is kept the same. A feature generated by bag-of-words is a vector where ***n*** is the number of words in the input documents vocabulary.

For instance, there are two documents:

- doc_1: ***"I love Italian food and Tunisian food."***
- doc_2: ***"I love the food here."***

The vocabulary of these two documents is:

["I", "love", "Italian", "food", "and", "Tunisian", "here",]

This vocabulary will produce feature vectors of length **8** (*i.e.* Vocabulary Cardinality). Given such vocabulary, the bag-of-word feature representation of these two documents are:

- BOW(doc_1): ***[1,1,1,2,1,1,0,0]***
- BOW(doc_2): ***[1,1,0,1,0,0,1,1]***

Using this technique we create the bag-of-words representation of each document in our dataset. If our dataset contains a big number of unique words some of that are not used very frequently, which is usually the case. So, among those, we chose ***N*** most frequent words and create a feature vector of dimension ***Nx1***. These feature vectors are then used for any machine learning task.

2. Term Frequency-Inverse Document Frequency

The Term Frequency-Inverse Document Frequency (*a.k.a.* TF-IDF) is another way to represent a document based on its words. With TF-IDF, words are given weights by TF-IDF importance instead of only frequency. TF-IDF provides a statistical measure used to evaluate the importance of the words with respect to the document in a collection or corpus. There are many commonly used words for each dataset that appear many times in the document but do not provide any important information. The weight increases in proportion to the number of occurrences of the word in the document. It also varies according to the frequency of the word in the corpus. Variants of the original formula are often used in search engines to assess the relevance of a document based on the user's search criteria.

By definition, TF-IDF embedding is composed by two terms: the first computes the normalized Term Frequency (TF), *a.k.a.* the occurrence a word appears in a document, divided by the total number of words in that document; the second term is the Inverse Document Frequency (IDF) which computes the logarithm of the number of the documents in the corpus divided by the

As shown in the figure below, *scikit-learn* library includes in the submodule `sklearn.feature_extraction.text` gathering utilities to build feature vectors from text documents in which we will find `CountVectorizer`. `CountVectorizer` easily convert a collection of raw documents to a matrix of TF-IDF numerical features.

```
from sklearn.feature_extraction.text import TfidfVectorizer
corpus = [
    'This is the first document.',
    'This document is the second document.',
    'And this is the third one.',
    'Is this the first document?',
]
vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(corpus)
print(vectorizer.get_feature_names())

print(X.shape)
```

TfidfVectorizer using Sklearn

3. Word2Vec

One of the most efficient techniques to represent a word is Word2Vec. Word2vec is a computationally efficient predictive model for learning word embeddings from raw text. It plots the words in a multi-dimensional vector space, where similar words tend to be close to each other. The surrounding words of a word provide the context to that word.

Let's start with a high-level insight about where we're going. Word2Vec uses a well-known trick in deep learning. In this trick, we train a simple neural network with a single hidden layer to perform a *fake task*, without a real need of the results of the task we trained it on. Instead, the main objective consists of learning a representation of the input data which are actually the ***'word vectors'*** gathered from the learned weights of the hidden layer. Such an approach may remind us of the way we train auto-encoders to perform dimensionality reduction.

Word2Vec can rely on either one of two model architectures in order to produce a distributed representation of input words: ***Continuous Bag-of-Words (CBoW)*** or **Continuous Skip-Gram** as shown in the figure below. Vector representation extracts semantic relationships based on the co-occurrence of words in the dataset.

The CBoW and skip-gram models are trained using a binary classification to discriminate between the real target word and the other words in the same context. The accuracy at which the model predicts the words depends on how many times the model sees these words within the same context throughout the dataset. The hidden representation is changed by more words and context co-occurrences during the training process, which allows the model to have more future successful predictions, leading to a better representation of word and context in the vector space. Skip gram is much slower than CBOW, but performs more accurately with infrequent words.

Word2Vec Training Model Architecture

Let's go deeper into details to understand the difference between CBOW architecture and Continuous skip-gram.

- In the first option , ***CBOW***; The model predicts the current word from a window of surrounding context words. The order of context words does not influence prediction (bag-of-words assumption). For example, if you gave the trained network the input words *"Soccer", "NBA", "Game"* and *"Tennis"*, the output probabilities are going to be much higher for words like *"Player"* and *"Tennis"* then for unrelated words like *"cheese"* and *"elections"*.
- Whereas, in the second option of using the ***continuous skip-gram architecture;*** the model uses the current word to predict the surrounding window of context words. The skip-gram architecture weighs nearby context words more heavily than more distant context words. The output probabilities are going to relate to how likely it is to find each vocabulary word near our input word. For example, if you gave the trained network the input word *"Europe"*, the output probabilities are going to be much higher for words like *"Belgium"* and *"Continent"* than for unrelated words like *"fruits"* and *"cats"*.

In both models, when we say ***'surrounding'***, there is actually a ***'window size'*** parameter to the algorithm. The size of the context window limits how many words before and after a given word would be included as context words of the given word. For example, window size of *3* will include the *3* words to the left and the *3* words to the right for each observed word in the sentence as context. Increasing the window size increases the training time as more word-context pairs need to be trained. Also it may capture context words that are not relevant to the current word. Decreasing the context words can capture relations between words and stop words which is not preferred.

4. Doc2Vec

Doc2Vec is another widely used technique that creates an embedding of a document irrespective to its length. While Word2Vec computes a feature vector for every **word** in the corpus, Doc2Vec computes a feature vector for every **document** in the corpus. Doc2vec model is based on Word2Vec, with only adding another vector *(paragraph ID)* to the input. The Doc2Vec model, by analogy with Word2Vec, can rely on one of two model architectures which are: Distributed Memory version of Paragraph Vector **(PV-DM)** and Distributed Bag of Words version of Paragraph Vector **(PV-DBOW).**

Doc2Vec Model Architecture

The above diagram is based on the CBOW model, but instead of using just nearby words to predict the word, we also added another feature vector, which is document-unique. So when training the word vectors *W*, the document vector *D* is trained as well, and at the end of training, it holds a numeric representation of the document.

The inputs consist of word vectors and document Id vectors. The word vector is a one-hot vector with a dimension `1xV`. The document Id vector has a dimension of `1xC`, where C is the number of total documents. The dimension of the weight matrix W of the hidden layer is `NxV`. The dimension of the weight matrix D of the hidden layer is `CxN`.

5. Doc2vecC

Doc2vecC tackles the problem of Doc2vec by including a global context through capturing the semantic meanings of the document. The architecture is very similar to Word2vec. In Doc2VecC, the average of the word embeddings in a document is used to represent the global context. But, unlike Word2vec, here at every iteration, words are randomly sampled from the document (document corruption). Vectors from those words are then averaged to obtain a document vector. This document vector is then used to predict the current word using the local context words as well as the global context. The global context is generated through an unbiased drop-out corruption. This corrupted document is represented as:

Then, the probability of observing a word *w_t* given the local context *c_t* and global context *x_i* is given as:

Once we have learned *U* using neural networks, each document can be simply represented as an average of the word embedding in that document. Hence, the document vector i is given as :

The output of the model shares both global and local semantics of the dataset. Averaging helps us obtain vectors for unseen documents as well, hence tackling the problem presented by Doc2vec. Since only a fraction of the words are used for training, the training time is much lower in comparison to Doc2vec. In any document, words derive context from their neighbouring words. Hence to provide more context, we append our data set with publicly available data sets that are subsets of textual data for businesses, reviews, and user data for personal, educational, and academic purposes.

. . .

Know your author

Rabeh Ayari is a Principal Data Scientist working on applied AI problems for credit risk modelling & fraud analytics and conducting original research in machine learning. My areas of expertise include data analysis using deep neural networks, machine learning, data visualization, feature engineering, linear/non-linear regression, classification, discrete optimization, operations research, evolutionary algorithm and linear programming. Feel free to drop me a message here!

Read more from Towards Data Science

Recommended from Medium

Prachetshah
Starting with ML with Simple Linear Regression in Python. Creating Salary Price Predicto...

Ajay n Jain
Machine Learning, Let's Learn

Musstafa
LeNet in depth

Francesca Lazz... in Microsoft Azu...
How to accelerate DevOps with Machine Learning lifecycle management

Khoa Ngu... in Towards Data Scie...
Detecting Heart Abnormalities Using 1D CNN on Data You Cannot See

Nicole Sim
Understanding Multi-class Classification Confusion Matrix in Python

Heliya Hasani
Welcome To Modern Optimization Series With Heliya Hasani !

Avinash Na... in Python in Plain En...
Understanding Logistic Regression and Building Model in Python