# **Language is Not just a Jumbled Bag of words:**

# What are the Word Embedding?

Word embedding are the texts converted into numbers may be different numerical representation of same text.

# Why do we need Word Embedding?

As it turns out, many Machine Learning algorithms and almost all Deep Learning Architectures are incapable of processing *strings*or *plain text*in their raw form. They require numbers as inputs to perform any sort of job, be it classification, regression etc. in broad terms. And with the huge amount of data that is present in the text format, it is imperative to extract knowledge out of it and build applications. Some real world applications of text applications are – sentiment analysis of reviews by Amazon etc., document or news classification or clustering by Google etc.

**Type of Word Embedding:**

* Frequency Based Embedding.
* Prediction based Embedding

There are generally three types of vectors that we encounter under this category.

* One Hot Encoding
* Count Vector
* TF-IDF Vector
* Co-Occurrence Vector
* One Hot Encoding
* See below Example:
* “Word Embeddings are Word converted into numbers”
* A *dictionary*may be the list of all unique words in the **sentence.**So, a dictionary may look like –
* ["Word","Embeddings","are" ,"Converted" ,"into" ,"numbers"]
* A ve*ctor*representation of a word may be a one-hot encoded vector where 1 stands for the position where the word exists and 0 everywhere else. The vector representation of “numbers”in this format according to the above dictionary is [0,0,0,0,0,1] and of converted is[0,0,0,1,0,0].

#### Count Vector

Consider a Corpus C of D documents {d1,d2…..dD} and N unique tokens extracted out of the corpus C. The N tokens will form our dictionary and the size of the Count Vector matrix M will be given by D X N. Each row in the matrix M contains the frequency of tokens in document D(i).

Let us understand this using a simple example.

D1: He is a lazy boy. She is also lazy.

D2: Neeraj is a lazy person.

The dictionary created may be a list of unique tokens(words) in the corpus =[‘He’,’She’,’lazy’,’boy’,’Neeraj’,’person’]

Here, D=2, N=6

The count matrix M of size 2 X 6 will be represented as –

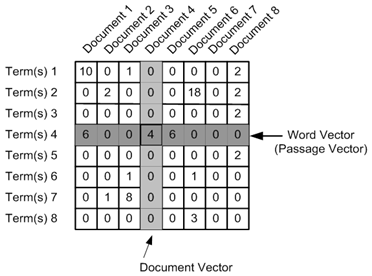
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | He | She | lazy | boy | Neeraj | person |
| D1 | 1 | 1 | 2 | 1 | 0 | 0 |
| D2 | 0 | 0 | 1 | 0 | 1 | 1 |

Now, a column can also be understood as word vector for the corresponding word in the matrix M. For example, the word vector for ‘lazy’ in the above matrix is [2,1] and so on.Here, the *rows* correspond to the *documents* in the corpus and the *columns* correspond to the *tokens* in the dictionary. The second row in the above matrix may be read as – D2 contains ‘lazy’: once, ‘Neeraj’: once and ‘person’ once.

Now there may be quite a few variations while preparing the above matrix M. The variations will be generally in-

1. The way dictionary is prepared.  
   Why? Because in real world applications we might have a corpus which contains millions of documents. And with millions of document, we can extract hundreds of millions of unique words. So basically, the matrix that will be prepared like above will be a very sparse one and inefficient for any computation. So an alternative to using every unique word as a dictionary element would be to pick say top 10,000 words based on frequency and then prepare a dictionary.
2. The way count is taken for each word.  
   We may either take the frequency (number of times a word has appeared in the document) or the presence(has the word appeared in the document?) to be the entry in the count matrix M. But generally, frequency method is preferred over the latter.

Below is a representational image of the matrix M for easy understanding.



#### 2.1.2 TF-IDF vectorization

This is another method which is based on the frequency method but it is different to the count vectorization in the sense that it takes into account not just the occurrence of a word in a single document but in the entire corpus. So, what is the rationale behind this? Let us try to understand.

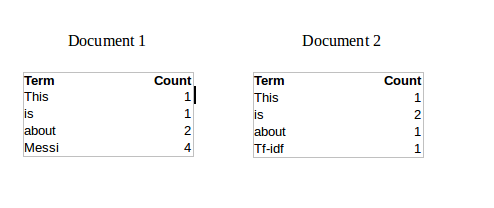
Common words like ‘is’, ‘the’, ‘a’ etc. tend to appear quite frequently in comparison to the words which are important to a document. For example, a document **A** on Lionel Messi is going to contain more occurences of the word “Messi” in comparison to other documents. But common words like “the” etc. are also going to be present in higher frequency in almost every document.

Ideally, what we would want is to down weight the common words occurring in almost all documents and give more importance to words that appear in a subset of documents.

TF-IDF works by penalising these common words by assigning them lower weights while giving importance to words like Messi in a particular document.

So, how exactly does TF-IDF work?

Consider the below sample table which gives the count of terms(tokens/words) in two documents.



Now, let us define a few terms related to TF-IDF.

TF = (Number of times term t appears in a document)/(Number of terms in the document)

So, TF(This,Document1) = 1/8

TF(This, Document2)=1/5

It denotes the contribution of the word to the document i.e words relevant to the document should be frequent. eg: A document about Messi should contain the word ‘Messi’ in large number.

IDF = log(N/n), where, N is the number of documents and n is the number of documents a term t has appeared in.

where N is the number of documents and n is the number of documents a term t has appeared in.

So, IDF(This) = log(2/2) = 0.

So, how do we explain the reasoning behind IDF? Ideally, if a word has appeared in all the document, then probably that word is not relevant to a particular document. But if it has appeared in a subset of documents then probably the word is of some relevance to the documents it is present in.

Let us compute IDF for the word ‘Messi’.

IDF(Messi) = log(2/1) = 0.301.

Now, let us compare the TF-IDF for a common word ‘This’ and a word ‘Messi’ which seems to be of relevance to Document 1.

TF-IDF(This,Document1) = (1/8) \* (0) = 0

TF-IDF(This, Document2) = (1/5) \* (0) = 0

TF-IDF(Messi, Document1) = (4/8)\*0.301 = 0.15

As, you can see for Document1, TF-IDF method heavily penalises the word ‘This’ but assigns greater weight to ‘Messi’. So, this may be understood as ‘Messi’ is an important word for Document1 from the context of the entire corpus.

Disadvantage:

* One-hot encoding represents similarity and difference at the *document* level, but because all words are rendered equidistant, it is not able to encode per-word similarity. Moreover, because all words are equally distant, *word form* becomes incredibly important
* The disadvantage is that for high cardinality, the feature space can really blow up quickly and you start fighting with the curse of dimensionality.
* Meaning: Discarding word order ignores the context, and in turn meaning of words in the document (semantics). Context and meaning can offer a lot to the model, that if modeled could tell the difference between the same words differently arranged (“this is interesting” vs “is this interesting”), synonyms (“old bike” vs “used bike”), and much more.
* Vocabulary: The vocabulary requires careful design, most specifically in order to manage the size, which impacts the sparsity of the document representations.
* Sparsity: Sparse representations are harder to model both for computational reasons (space and time complexity) and also for information reasons, where the challenge is for the models to harness so little information in such a large representational space.
* max\_features: Instead of using all words, max number of word can be chosen to reduce the model complexity and size.

In inverted indexing, originally, you have a list of documents associated to a word (where the word appear), and if you use TF-IDF, that word will be associated with the TF-IDF values of its document list. Then the term vectors are used to calculate the similarity between two documents, or two textual units in general. The two term vectors can also be used to calculate the "relationship" between their corresponding words. But this relationship is not semantically or syntactically related, it's just about the level of common occurrence in the textual units to be learnt from.

For example, the "similarity" of two words "neural" and "network" (computed by cosine distance) formed from inverted index of a deep learning corpus might be very large (almost 1), meaning they are very similar, but in fact, they are commonly occurred in this kind of corpus, not similar in meaning.

For example, in a big text corpus, there are two sentences:

Sentence1 :"BMW is a German car manufacturer"

Sentence2 :"BMW is a German automobile manufacturer".

We, and the computer, can infer "car" and "automobile" are synonyms.

Other two sentences:

Sentence1 :"We will go there this Thursday"

Sentence2 :"I will go there this Sunday"

Thursday and Sunday have the same syntactic role and they are somehow semantically related (days of the week)

.