



A Guide on Word Embeddings in NLP

Taken from: <https://www.turing.com/kb/guide-on-word-embeddings-in-nlp>

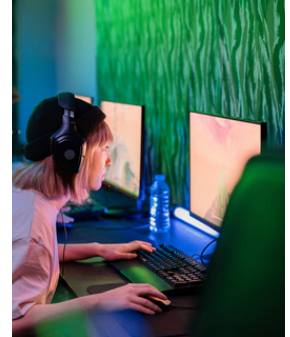


A Guide on Word Embeddings in NLP

Word embedding in NLP is an important term that is used for representing words for text analysis in the form of real-valued vectors. It is an advancement in NLP that has improved the ability of computers to understand text-based content in a better way. It is considered one of the most significant breakthroughs of deep learning for solving challenging natural language processing problems.

In this approach, words and documents are represented in the form of numeric vectors allowing similar words to have similar vector representations.

The extracted features are fed into a machine learning model so as to work with text data and preserve the semantic and syntactic information. This information once received in its converted form is used by NLP algorithms that easily digest these learned representations and process textual information.



Due to the perks this technology brings on the table, the popularity of ML NLP is surging making it one of the most chosen fields by the developers.

Now that you have a basic understanding of the topic, let us start from scratch by introducing you to word embeddings, its techniques, and applications.

1. What is word embedding?

Word embedding or word vector is an approach with which we represent documents and words. It is defined as a numeric vector input that allows words with similar meanings to have the same representation. It can approximate meaning and represent a word in a lower dimensional space.





These can be trained much faster than the hand-built models that use graph embeddings like WordNet. For instance, a word embedding with 50 values holds the capability of representing 50 unique features. Many people choose pre-trained word embedding models like Flair, fastText, SpaCy, and others. We will discuss it further in the article. Let's move on to learn it briefly with an example of the same.

1.1. The problem

2. Applications Of Artificial Intelligence in Education

Given a supervised learning task to predict which tweets are about real disasters and which ones are not (classification). Here the independent variable would be the tweets (text) and the target variable would be the binary values (1: Real Disaster, 0: Not real Disaster). Now, Machine Learning and Deep Learning algorithms only take numeric input. So, how do we convert tweets to their numeric values? We will dive deep into the techniques to solve such problems, but first let's look at the solution provided by word embedding.

1.2. The solution

Word Embeddings in NLP is a technique where individual words are represented as real-valued vectors in a lower-dimensional space and captures inter-word semantics. Each word is represented by a real-valued vector with tens or hundreds of dimensions.



2. Term frequency-inverse document frequency (TF-IDF)

Term frequency-inverse document frequency is the machine learning algorithm that is used for word embedding for text. It comprises two metrics, namely term frequency (TF) and inverse document frequency (IDF).

This algorithm works on a statistical measure of finding word relevance in the text that can be in the form of a single document or various documents that are referred to as corpus.

The term frequency (TF) score measures the frequency of words in a particular document. In simple words, it means that the occurrence of words is counted in the documents.

The inverse document frequency or the IDF score measures the rarity of the words in the text. It is given more importance over the term frequency score because even though the TF score gives more weightage to frequently occurring words, the IDF score focuses on rarely used words in the corpus that may hold significant information.



TF-IDF algorithm finds application in solving simpler natural language processing and machine learning problems for tasks like information retrieval, stop words removal, keyword extraction, and basic text analysis.

However, it does not capture the semantic meaning of words efficiently in a sequence.

Now let's understand it further with an example. We will see how vectorization is done in TF-IDF.

$$\text{tf-idf}_{i,j} = \text{Term Frequency}_{i,j} \times \text{Inverse Document Frequency}_i$$

Where,

$$\text{Term Frequency}_{i,j} = \frac{\text{Term } i \text{ frequency in document } j}{\text{Total no. of terms in document } j}$$

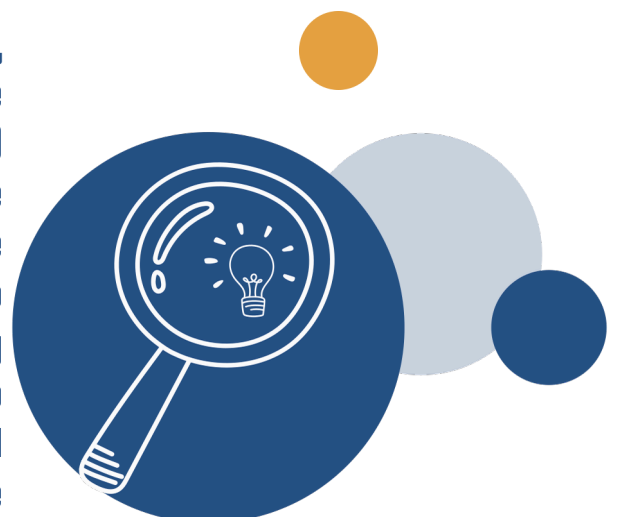
$$\text{Inverse Document Frequency}_i = \log \left(\frac{\text{Total documents}}{\text{No. of documents containing term } i} \right)$$

To create TF-IDF vectors, we use Scikit-learn's TF-IDF Vectorizer. After applying it to the previous 4 sample tweets, we obtain -

	ablaze	accident	car	caught	fire	jam	kind	sadly	set	swear	true	up	world
0	0.00	0.00	0.00	0.0	0.0	0.00	0.67	0.53	0.00	0.00	0.53	0.0	0.00
1	0.47	0.00	0.00	0.0	0.0	0.47	0.00	0.00	0.47	0.37	0.00	0.0	0.47
2	0.00	0.59	0.47	0.0	0.0	0.00	0.00	0.00	0.00	0.47	0.47	0.0	0.00
3	0.00	0.00	0.64	0.4	0.4	0.00	0.00	0.32	0.00	0.00	0.00	0.4	0.00

2.1. Output of TfidfVectorizer

The rows represent each document, the columns represent the vocabulary, and the values of $\text{tf-idf}(i,j)$ are obtained through the above formula. This matrix obtained can be used along with the target variable to train a machine learning/deep learning model. Let us now discuss two different approaches to word embeddings. We'll also look at the hands-on part!



3. Bag of words (BOW)

A bag of words is one of the popular word embedding techniques of text where each value in the vector would represent the count of words in a document/sentence. In other words, it extracts features from the text. We also refer to it as vectorization. To get you started, here's how you can proceed to create BOW.

- ✓ In the first step, you have to tokenize the text into
- ✓ Next, the sentences tokenized in the first step have further tokenized words.
- ✓ Eliminate any stop words or
- ✓ Then, convert all the words to
- ✓ Finally, move to create a frequency distribution chart of the words.
- ✓ We will discuss BOW with proper examples in the continuous bag of word selection below.

4. Word2Vec

The Word2Vec method was developed by Google in 2013. Presently, we use this technique for all advanced natural language processing (NLP) problems. It was invented for training word embeddings and is based on a distributional hypothesis.



In this hypothesis, it uses skip-grams or a continuous bag of words (CBOW).

These are basically shallow neural networks that have an input layer, an output layer, and a projection layer. It reconstructs the linguistic context of words by considering both the order of words in history as well as the future.

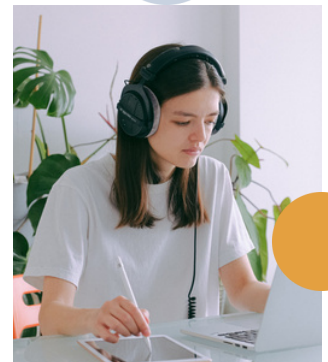
The method involves iteration over a corpus of text to learn the association between the words. It relies on a hypothesis that the neighboring words in a text have semantic similarities with each other. It assists in mapping semantically similar words to geometrically close embedding vectors.

It uses the cosine similarity metric to measure semantic similarity. Cosine similarity is equal to $\text{Cos}(\text{angle})$ where the angle is measured between the vector representation of two words/documents.

So if the cosine angle is one, it means that the words are overlapping.

And if the cosine angle is a right angle or 90° , It means words hold no contextual similarity and are independent of each other.

To summarize, we can say that this metric assigns similar vector representations to the same boards.



4.1. Two variants of Word2Vec

Word2Vec has two neural network-based variants: Continuous Bag of Words (CBOW) and Skip-gram.

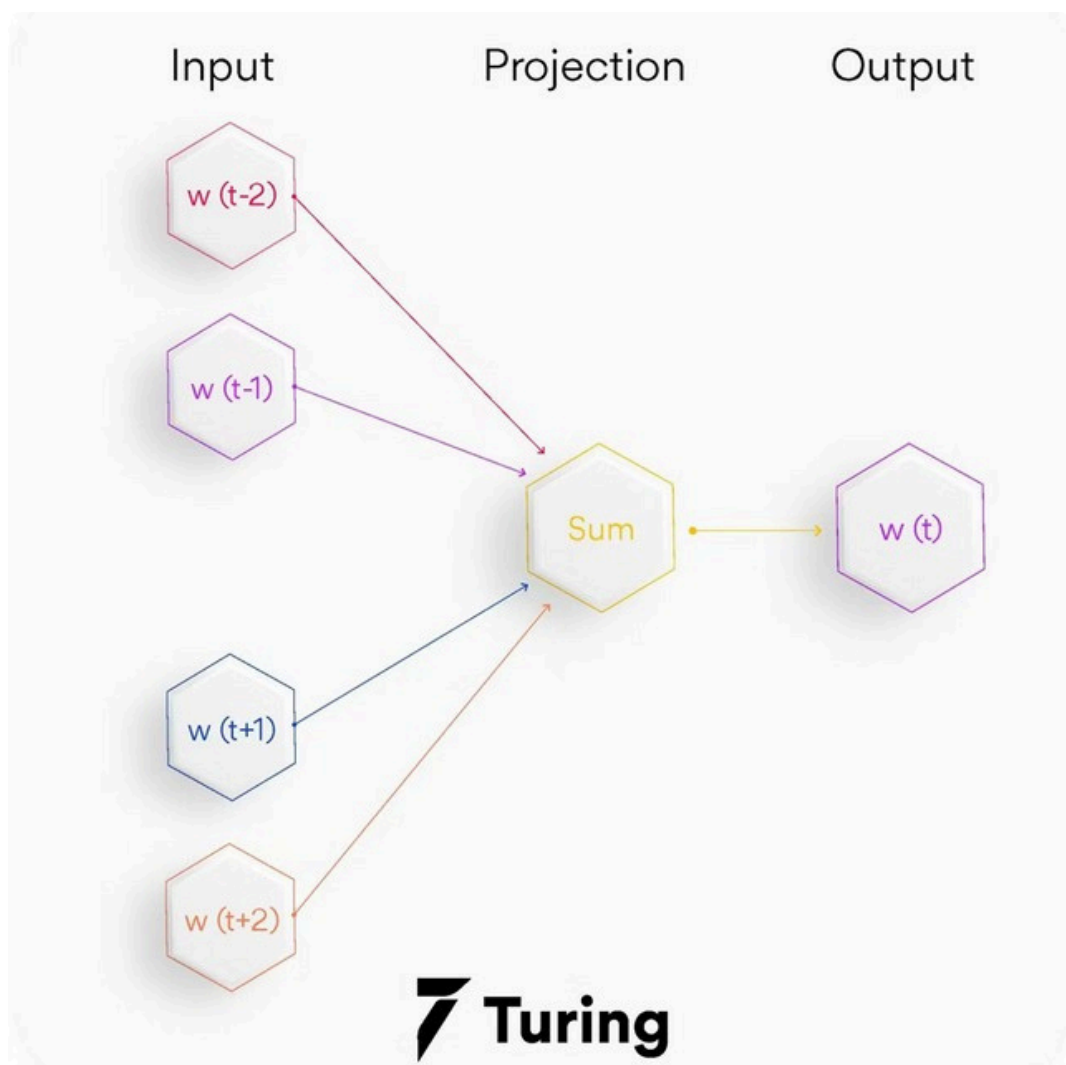
1. CBOW –

The continuous bag of words variant includes various inputs that are taken by the neural network model. Out of this, it predicts the targeted word that closely relates to the context of different words fed as input.

It is fast and a great way to find better numerical representation for frequently occurring words. Let us understand the concept of context and the current word for CBOW.



In CBOW, we define a window size. The middle word is the current word and the surrounding words (past and future words) are the context. CBOW utilizes the context to predict the current words. Each word is encoded using One Hot Encoding in the defined vocabulary and sent to the CBOW neural network.



The hidden layer is a standard fully-connected dense layer. The output layer generates probabilities for the target word from the vocabulary. As we have discussed earlier about the bag of words (BOW) and it being also termed as vectorizer, we will take an example here to clarify it further. Let's take a small part of disaster tweets, 4 tweets, to understand how BOW works:-



'kind true sadly',
'swear jam set world ablaze',
'swear true car accident',
'car sadly car caught up fire'

To create BOW, we use Scikit-learn's CountVectorizer, which tokenizes a collection of text documents, builds a vocabulary of known words, and encodes new documents using that vocabulary.

	ablaze	accident	car	caught	fire	jam	kind	sadly	set	swear	true	up	world
0	0	0	0	0	0	0	1	1	0	0	1	0	0
1	1	0	0	0	0	1	0	0	1	1	0	0	1
2	0	1	1	0	0	0	0	0	0	1	1	0	0
3	0	0	2	1	1	0	0	1	0	0	0	1	0

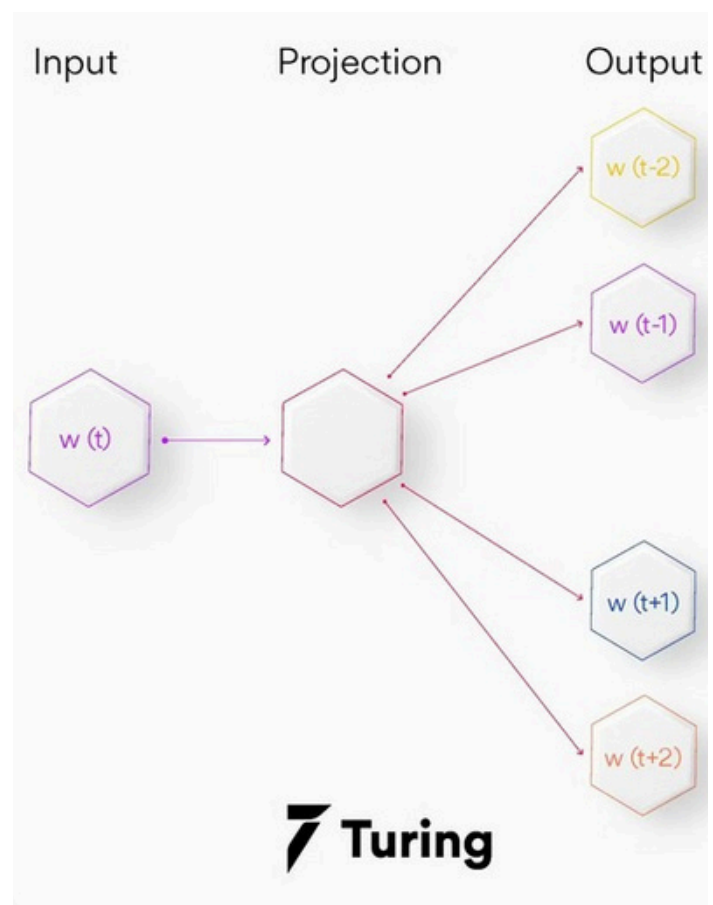
Output of Count Vectorizer

Here the rows represent each document (4 in our case), the columns represent the vocabulary (unique words in all the documents) and the values represent the count of the words of the respective rows. In the same way, we can apply CountVectorizer to the complete training data tweets (11,370 documents) and obtain a matrix that can be used along with the target variable to train a machine learning/deep learning model.



2. Skip-gram –

is a slightly different word embedding technique in comparison to CBOW as it does not predict the current word based on the context. Instead, each current word is used as an input to a log-linear classifier along with a continuous projection layer. This way, it predicts words in a certain range before and after the current word. This variant takes only one word as an input and then predicts the closely related context words. That is the reason it can efficiently represent rare words.





The end goal of Word2Vec (both variants) is to learn the weights of the hidden layer. The hidden consequences will be used as our word embeddings!! Let's now see the code for creating custom word embeddings using Word2Vec-

Import Libraries

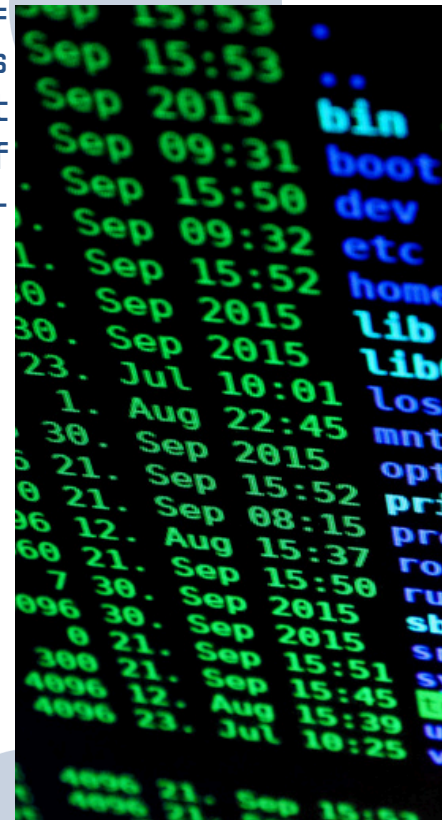
```
from gensim.models import  
Word2Vec import nltk import re  
from nltk.corpus import  
stopwords
```

Preprocess the Text

#Word2Vec inputs a corpus of documents split into constituent words. corpus = [] for i in range(0,len(X)): tweet = re.sub("[^a-zA-Z]", " ",X[i]) tweet = tweet.lower() tweet = tweet.split() corpus.append(tweet) Here is the exciting part! Let's try to see the most similar words (vector representations) of some random words from the tweets - model.wv.most_similar('disaster')

Output -

```
[('alert', 0.9997072219848633),  
 ('military', 0.9997004270553589),  
 ('murder', 0.999669075012207),  
 ('ash', 0.9996492862701416),  
 ('base', 0.9996055364608765),  
 ('oil', 0.9996046423912048),  
 ('along', 0.99959397315979),  
 ('based', 0.9995934963226318),  
 ('inundation', 0.9995468854904175),  
 ('west', 0.9995380640029907)]
```





List of tuples of words and their predicted probability. The embedding vector of 'disaster' –

```
array([-0.5968882 , -0.33086956, -0.32643065, -0.3670732 ,  0.628059 ,
       -0.3692328 , -0.37902787, -0.12308089, -0.38124698, -0.03940517,
        0.2260839 ,  0.10852845, -0.2873811 , -0.42781743,  0.06604357,
       -0.07114276, -0.29775023, -0.99628943, -0.54497653, -0.11718027,
       -0.15935768,  0.09587188, -0.2503798 ,  0.06768776,  0.3311586 ,
        0.43098116,  0.06936899,  0.24311952,  0.14515282,  0.19245838,
        0.10462623, -0.45676082,  0.5662387 ,  0.69908774,  0.48064467,
        0.27378514, -0.45430255,  0.17282294, -0.40275463, -0.38083532,
        0.47487524,  0.31950948, -0.1109335 ,  0.2165357 ,  0.034114 ,
        0.05689918,  0.20939653,  0.15209009, -0.24204595,  0.03478364,
        0.1616051 , -0.5827333 , -0.47017908,  0.26226178, -0.11884775,
        0.40180743, -0.5173988 , -0.19270805,  0.660391 , -0.24518126,
       -0.42860952, -0.22274768,  0.4887834 ,  0.49302152,  0.38799986,
       -0.041193 , -0.38600504, -0.37632987,  0.04570564,  0.50462466,
       -0.14396502,  0.33490512, -0.15964787, -0.21363072, -0.25445372,
        0.52389127,  0.5747422 , -0.25075617, -0.5339069 ,  0.2582965 ,
       -0.16139959,  0.09748188,  0.04540966, -0.27768216, -0.51260656,
       -0.06189002, -0.54032195, -0.21863565,  0.06233869,  0.13287479,
        0.49741864,  0.1772418 ,  0.02064824, -0.04775626, -0.16804916,
        0.4643644 ,  0.5546319 ,  0.68051434,  0.7790246 ,  0.5617202 ],
      dtype=float32)
```

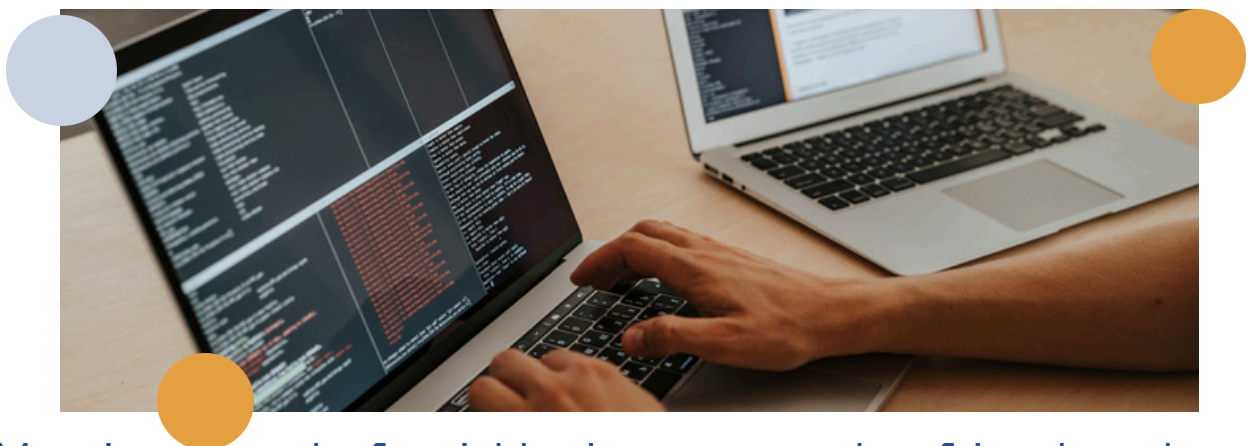
List of tuples of words and their predicted probability. The embedding vector of 'disaster' –

```
array([-0.5968882 , -0.33086956, -0.32643065, -0.3670732 ,  0.628059 ,
       -0.3692328 , -0.37902787, -0.12308089, -0.38124698, -0.03940517,
        0.2260839 ,  0.10852845, -0.2873811 , -0.42781743,  0.06604357,
       -0.07114276, -0.29775023, -0.99628943, -0.54497653, -0.11718027,
       -0.15935768,  0.09587188, -0.2503798 ,  0.06768776,  0.3311586 ,
        0.43098116,  0.06936899,  0.24311952,  0.14515282,  0.19245838,
        0.10462623, -0.45676082,  0.5662387 ,  0.69908774,  0.48064467,
        0.27378514, -0.45430255,  0.17282294, -0.40275463, -0.38083532,
        0.47487524,  0.31950948, -0.1109335 ,  0.2165357 ,  0.034114 ,
        0.05689918,  0.20939653,  0.15209009, -0.24204595,  0.03478364,
        0.1616051 , -0.5827333 , -0.47017908,  0.26226178, -0.11884775,
        0.40180743, -0.5173988 , -0.19270805,  0.660391 , -0.24518126,
       -0.42860952, -0.22274768,  0.4887834 ,  0.49302152,  0.38799986,
       -0.041193 , -0.38600504, -0.37632987,  0.04570564,  0.50462466,
       -0.14396502,  0.33490512, -0.15964787, -0.21363072, -0.25445372,
        0.52389127,  0.5747422 , -0.25075617, -0.5339069 ,  0.2582965 ,
       -0.16139959,  0.09748188,  0.04540966, -0.27768216, -0.51260656,
       -0.06189002, -0.54032195, -0.21863565,  0.06233869,  0.13287479,
        0.49741864,  0.1772418 ,  0.02064824, -0.04775626, -0.16804916,
        0.4643644 ,  0.5546319 ,  0.68051434,  0.7790246 ,  0.5617202 ],
      dtype=float32)
```

dimensionality = 100

5. Challenges with the bag of words and TF-IDF

Now let's discuss the challenges with the two text vectorization techniques we have discussed till now. In BOW, the size of the vector is equal to the number of elements in the vocabulary. If most of the values in the vector are zero then the bag of words will be a sparse matrix. Sparse representations are harder to model both for computational reasons and also for informational reasons. Also, in BOW there is a lack of meaningful relations and no consideration for the order of words. Here's more that adds to the challenge with this word embedding technique.



Massive amount of weights: Large amounts of input vectors invite massive amounts of weight for a neural network. **No meaningful relations or consideration for word order:** The bag of words does not consider the order in which the words appear in the sentences or a text. **Computationally intensive:** With more weight comes the need for more computation to train and predict. While the TF-IDF model contains the information on the more important words and the less important ones, it does not solve the challenge of high dimensionality and sparsity, and unlike BOW it also makes no use of semantic similarities between words.

6. GloVe: Global Vector for word representation

The GloVe method of word embedding in NLP was developed at Stanford by Pennington, et al. It is referred to as global vectors because the global corpus statistics were captured directly by the model. It finds great performance in word analogy and named entity recognition problems.

This technique reduces the computational cost of training the model because of a simpler least square cost or error function that further results in different and improved word embeddings. It leverages local context window methods like the skip-gram model of Mikolov and Global Matrix factorization methods for generating low dimensional word representations.

Latent semantic analysis (LSA) is a Global Matrix factorization method that does not do well on word analogy but leverages statistical information indicating a sub-optimal vector space structure.



On the contrary, the skip-gram method performs better on the analogy task. However, it does not utilize the statistics of the corpus properly because of no training on global co-occurrence counts.

So, unlike Word2Vec, which creates word embeddings using local context, GloVe focuses on global context to create word embeddings which gives it an edge over Word2Vec. In GloVe, the semantic relationship between the words is obtained using a co-occurrence matrix.



Consider two sentences –

- **I am a data science enthusiast I am looking for**
- **a data science job**

The co-occurrence matrix involved in GloVe would look like this for the above sentences –

	I	am	a	data	science	enthusiast	looking	for	job
I	0	2	0	0	0	0	0	0	0
am	2	0	1	0	0	0	1	0	0
a	0	1	0	2	0	0	0	1	0
data	0	0	2	0	2	0	0	0	0
science	0	0	0	2	0	1	0	0	1
enthusiast	0	0	0	0	1	0	0	0	0
looking	0	1	0	0	0	0	0	1	0
for	0	0	1	0	0	0	1	0	0
job	0	0	0	0	1	0	0	0	0

Window Size = 1

Each value in this matrix represents the count of co-occurrence with the corresponding word in row/column. Observe here – this co-occurrence matrix is created using global word co-occurrence count (no. of times the words appeared consecutively; for window size=1). If a text corpus has 1m unique words, the co-occurrence matrix would be 1m x 1m in shape. The core idea behind GloVe is that the word co-occurrence is the most important statistical information available for the model to ‘learn’ the word representation.



Let's now see an example from Stanford's GloVe paper of how the co-occurrence probability ratios work in GloVe. "For example, consider the co-occurrence probabilities for target words ice and steam with various probe words from the vocabulary. Here are some actual probabilities from a corpus of 6 billion words:"

Probability and Ratio	$k = \text{solid}$	$k = \text{gas}$	$k = \text{water}$	$k = \text{fashion}$
$P(k \text{ice})$	1.9×10^{-4}	6.6×10^{-5}	3.0×10^{-3}	1.7×10^{-5}
$P(k \text{steam})$	2.2×10^{-5}	7.8×10^{-4}	2.2×10^{-3}	1.8×10^{-5}
$P(k \text{ice})/P(k \text{steam})$	8.9	8.5×10^{-2}	1.36	0.96

Her

$P(j|i)$ = Probability(j is in the context of i)

Mathematically -

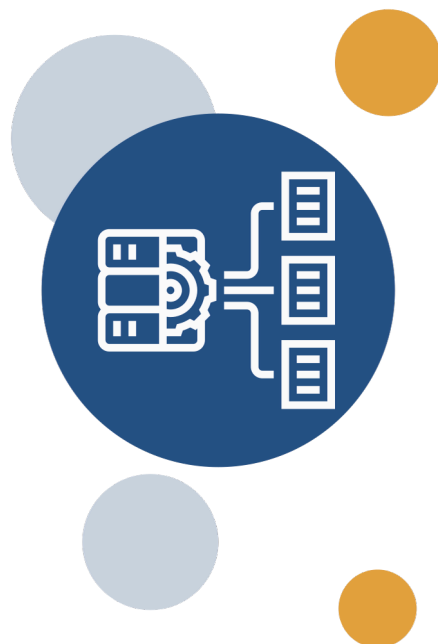
$$P(j|i) = \frac{x_{ij}}{x_i}$$

Where,

x_{ij} = Co-occurrence Matrix[i][j]

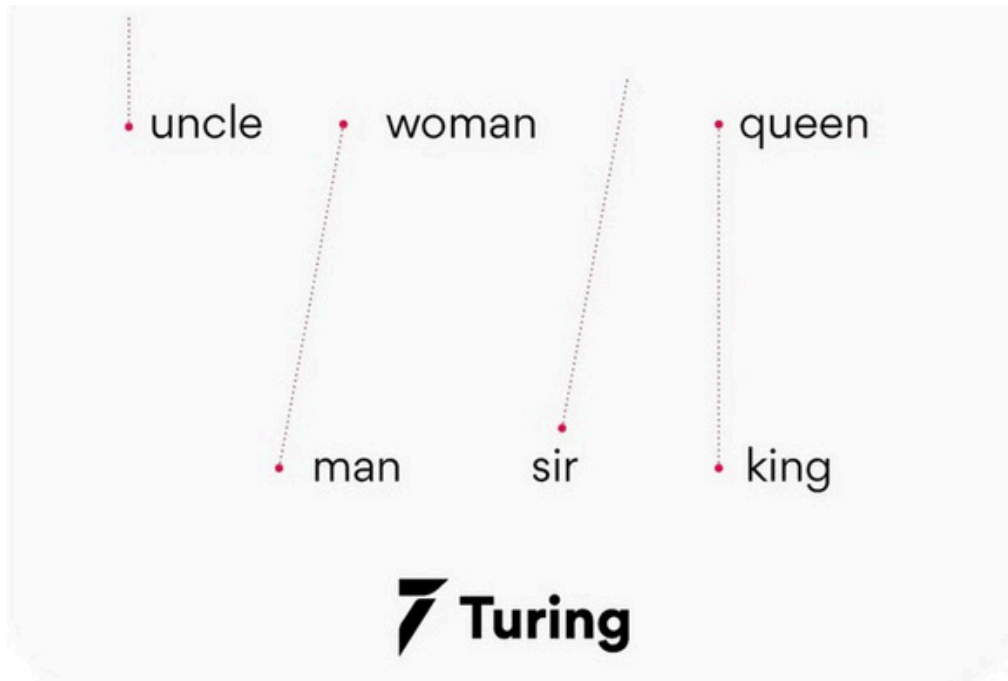
$$x_i = \sum_1^k x_{ik} \text{ (number of times any word appears in the context of word i)}$$

Let's take $k = \text{solid}$ i.e, words related to ice but unrelated to steam. The expected P_{ik} / P_{jk} ratio will be large. Similarly, for words k which are related to steam but not to ice, say $k = \text{gas}$, the ratio will be small. For words like water or fashion, which are either related to both ice and steam or neither to both respectively, the ratio should be approximately one.





The probability ratio is able to better discriminate between two relevant words. Hence in GloVe, the starting point for word vector learning is ratios of co-occurrence probabilities rather than the probabilities themselves.



Enough of the theory. Time for the code!

```
import Libraries
import nltk
import re
from nltk.corpus import stopwords
from glove import Corpus, Glove
Text Preprocessing
#GloVe inputs a corpus of documents splitted into constituent words
corpus = []
for i in range(0, len(X)):
    tweet = re.sub("[^a-zA-Z]", " ", X[i])
    tweet = tweet.lower()
    tweet = tweet.split()
    corpus.append(tweet)
```

Train the word Embeddings

```
corpus = Corpus()
corpus.fit(text_corpus, window = 5)
glove = Glove(no_components=100,
learning_rate=0.05)
#no_components = dimensionality
of word embeddings = 100
glove.fit(corpus.matrix, epochs=100,
no_threads=4, verbose=True)
glove.add_dictionary(corpus.diction
ary) Find most similar -
glove.most_similar("storm", number=1
0)
```

Output -

```
[('dust', 0.9220579668543765),
 ('brendan', 0.904046738220378),
 ('violent', 0.8719569025522304),
 ('fecal', 0.8671525955282415),
 ('goddess', 0.8236152015268149),
 ('occasionally', 0.7480771946676805),
 ('monster', 0.6818300005559692),
 ('comin', 0.6613716746320002),
 ('system', 0.6500671375524384)]
```

List of tuples of words and their predicted probability



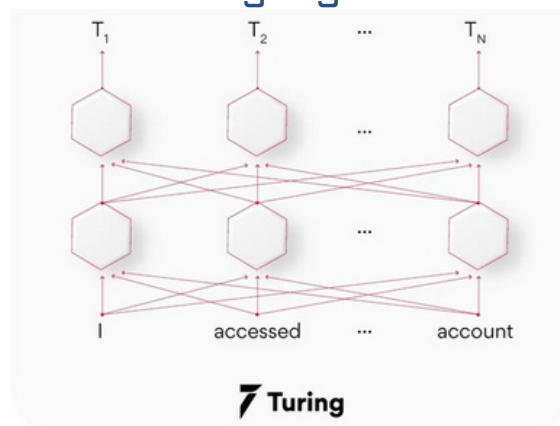
7. BERT (Bidirectional encoder representations from transformers)

This natural language processing (NLP) based language algorithm belongs to a class known as transformers. It comes in two variants namely BERT-Base, which includes 110 million parameters, and BERT-Large, which has 340 million parameters.



It relies on an attention mechanism for generating high-quality word embeddings that are contextualized. So when the embedding goes through the training process, they are passed through each BERT layer so that its attention mechanism can capture the word associations based on the words on the left and those on the right. It is an advanced technique in comparison to the discussed above as it creates better word embedding. The credit goes to the pre-trained model on Wikipedia data sets and massive word corpus. This technique can be further improved for task-specific data sets by fine-tuning the embeddings.

It finds great application in language translation tasks.



8. Conclusion

Word embeddings can train deep learning models like GRU, LSTM, and Transformers, which have been successful in NLP tasks such as sentiment classification, name entity recognition, speech recognition, etc. Here's a final checklist for a recap.

- Bag of words: Extracts features from the text
- TF-IDF: Information retrieval, keyword
- extraction Word2Vec: Semantic analysis task
- GloVe: Word analogy, named entity recognition
- tasks BERT: language translation, question answering system

