GENSIM

* Gensim is billed as a Natural Language Processing package that does ‘Topic Modeling for Humans’. But it is practically much more than that. It is a leading and a state-of-the-art package for processing texts, working with word vector models (such as Word2Vec, FastText etc) and for building topic models.
* If you are unfamiliar with [topic modeling](https://www.machinelearningplus.com/nlp/topic-modeling-gensim-python/), it is a technique to extract the underlying topics from large volumes of text. Gensim provides algorithms like LDA and LSI and the necessary sophistication to build high-quality topic models.
* You may argue that topic models and word embedding are available in other packages like scikit, R etc. But the width and scope of facilities to build and evaluate topic models are unparalleled in gensim, plus many more convenient facilities for text processing.
* It is a great package for processing texts, working with word vector models (such as Word2Vec, FastText etc) and for building topic models.
* Also, another significant advantage with gensim is: it lets you handle large text files without having to load the entire file in memory.

## Core Concepts

The core concepts of gensim are:

1. [Document](https://radimrehurek.com/gensim/auto_examples/core/run_core_concepts.html#core-concepts-document): some text.
2. [Corpus](https://radimrehurek.com/gensim/auto_examples/core/run_core_concepts.html#core-concepts-corpus): a collection of documents.
3. [Vector](https://radimrehurek.com/gensim/auto_examples/core/run_core_concepts.html#core-concepts-vector): a mathematically convenient representation of a document.
4. [Model](https://radimrehurek.com/gensim/auto_examples/core/run_core_concepts.html#core-concepts-model): an algorithm for transforming vectors from one representation to another.

## *What is a Dictionary and Corpus?*

* In order to work on text documents, Gensim requires the words (aka tokens) be converted to unique ids. In order to achieve that, Gensim lets you create a Dictionary object that maps each word to a unique id.
* By converting your text/sentences to a [list of words] and pass it to the corpora.Dictionary () object.

## Why is the dictionary object needed and where can it be used?

The dictionary object is typically used to create a ‘bag of words’ Corpus. It is this Dictionary and the bag-of-words (Corpus) that are used as inputs to topic modeling and other models that Gensim specializes in.

## *What sort of text inputs can gensim handle?*

The input text typically comes in 3 different forms:

1. As sentences stored in python’s native list object
2. As one single text file, small or large.
3. In multiple text files.

 When your text input is large, you need to be able to create the dictionary object without having to load the entire text file. The good news is Gensim lets you read the text and update the dictionary, one line at a time, without loading the entire text file into system memory.

A ‘token’ typically means a ‘word’. A ‘document’ can typically refer to a ‘sentence’ or ‘paragraph’ and a ‘corpus’ is typically a ‘collection of documents as a bag of words’. That is, for each document, a corpus contains each word’s id and its frequency count in that document. As a result, information of the order of words is lost.

## How to create a Dictionary from a list of sentences?

In gensim, the dictionary contains a map of all words (tokens) to its unique id.

You can create a dictionary from a paragraph of sentences, from a text file that contains multiple lines of text and from multiple such text files contained in a directory. For the second and third cases, we will do it without loading the entire file into memory so that the dictionary gets updated as you read the text line by line.

1. Let’s start with the ‘List of sentences’ input.

When you have multiple sentences, you need to convert each sentence to a list of words. A list comprehension is a common way to do this.

Gensim will use this dictionary to create a bag-of-words corpus where the words in the documents are replaced with its respective id provided by this dictionary. If you get new documents in the future, it is also possible to update an existing dictionary to include the new words.

## *How to create a Dictionary from one or more text files?*

You can also create a dictionary from a text file or from a directory of text files. The below example reads a file line-by-line and uses gensim’s simple\_preprocess to process one line of the file at a time. The advantage here is it let’s you read an entire text file without loading the file in memory all at once.

## How to read one-line-at-a-time from multiple files?

Assuming you have all the text files in the same directory, you need to define a class with an \_\_iter\_\_ method. The \_\_iter\_\_() method should iterate through all the files in a given directory and yield the processed list of word tokens.

## *How to create a bag of words corpus in gensim?*

The next important object you need to familiarize with in order to work in gensim is the Corpus (a Bag of Words). That is, it is a corpus object that contains the word id and its frequency in each document. You can think of it as gensim’s equivalent of a Document-Term matrix.

Once you have the updated dictionary, all you need to do to create a bag of words corpus is to pass the tokenized list of words to the Dictionary.doc2bow()

How to interpret the above corpus?

The (0, 1) in line 1 means, the word with id=0 appears once in the 1st document.  
Likewise, the (4, 4) in the second list item means the word with id 4 appears 4 times in the second document.

## How to create a bag of words corpus from a text file?

Reading words from a python list is quite straightforward because the entire text was in-memory already. However, you may have a large file that you don’t want to load the entire file in memory.

You can import such files one line at a time by defining a class and the \_\_iter\_\_ function that iteratively reads the file one line at a time and yields a corpus object.

But how to create the corpus object?

The \_\_iter\_\_() from BoWCorpus reads a line from the file, process it to a list of words using simple\_preprocess() and pass that to the dictionary.doc2bow().Also, notice that I am using the smart\_open() from [smart\_open](https://github.com/RaRe-Technologies/smart_open) package because, it lets you open and read large files line-by-line from a variety of sources such as S3, HDFS, WebHDFS, HTTP, or local and compressed files.

## *How to create the TFIDF matrix (corpus) in gensim?*

The Term Frequency – Inverse Document Frequency(TF-IDF) is also a bag-of-words model but unlike the regular corpus, TFIDF down weights tokens (words) that appears frequently across documents.

How is TFIDF computed?

Tf-Idf is computed by multiplying a local component like term frequency (TF) with a global component, that is, inverse document frequency (IDF) and optionally normalizing the result to unit length.

As a result of this, the words that occur frequently across documents will get down weighted.

So, how to get the TFIDF weights?

By training the corpus with models.TfidfModel (). Then, apply the corpus within the square brackets of the trained tfidf model.

## How to use gensim downloader API to load datasets?

Gensim provides an inbuilt API to download popular text datasets and word embedding models. Using the API to download the dataset is as simple as calling the api.load () method with the right data or model name

## *How to create bigrams and trigrams using Phraser models?*

What are bigrams and trigrams? Why do they matter?

In paragraphs, certain words always tend to occur in pairs (bigram) or in groups of threes (trigram). Because the two words combined together form the actual entity. For example: The word ‘French’ refers the language or region and the word ‘revolution’ can refer to the planetary revolution. But combining them, ‘French Revolution’, refers to something completely different.

It’s quite important to form bigrams and trigrams from sentences, especially when working with bag-of-words models.

How to create the bigrams?

It’s quite easy and efficient with gensim’s Phrases model. The created Phrases model allows indexing, so, just pass the original text (list) to the built Phrases model to form the bigrams

Trigram-Simply rinse and repeat the same procedure to the output of the bigram model. Once you’ve generated the bigrams, you can pass the output to train a new Phrases model. Then, apply the bigrammed corpus on the trained trigram model.

## How to train Word2Vec model using gensim?

A word embedding model is a model that can provide numerical vectors for a given word. Using the Gensim’s downloader API, you can download pre-built word embedding models like word2vec, fasttext, GloVe and ConceptNet. These are built on large corpuses of commonly occurring text data such as wikipedia, google news etc.

However, if you are working in a specialized niche such as technical documents, you may not able to get word embeddings for all the words. So, in such cases its desirable to train your own model.

Gensim’s Word2Vec implementation let’s you train your own word embedding model for a given corpus

## *How to update an existing Word2Vec model with new data?*

On an existing Word2Vec model, call the build\_vocab() on the new datset and then call the train() method. build\_vocab() is called first because the model has to be apprised of what new words to expect in the incoming corpus.

## How to extract word vectors using pre-trained Word2Vec and FastText models?

Gensim lets you download state of the art pretrained models through the downloader API.

## *How to create document vectors using Doc2Vec?*

Unlike Word2Vec, a Doc2Vec model provides a vectorised representation of a group of words taken collectively as a single unit. It is not a simple average of the word vectors of the words in the sentence.

## How to compute similarity metrics like cosine similarity and soft cosine similarity?

Soft cosine similarity is similar to [cosine similarity](https://en.wikipedia.org/wiki/Cosine_similarity) but in addition considers the semantic relationship between the words through its vector representation.

To compute soft cosines, you will need a word embedding model like Word2Vec or FastText. First, compute the similarity\_matrix. Then convert the input sentences to bag-of-words corpus and pass them to the softcossim() along with the similarity matrix.

*How to summarize text documents?*

Gensim implements the textrank summarization using the summarize() function in the summarization module. All you need to do is to pass in the text string along with either the output summarization ratio or the maximum count of words in the summarized output.

Text Summarization

Demonstrates summarizing text by extracting the most important sentences from it. This module automatically summarizes the given text, by extracting one or more important sentences from the text. In a similar way, it can also extract keywords. Review the performance of the summarizer in terms of speed.

This module also supports **keyword** extraction. Keyword extraction works in the same way as summary generation (i.e. sentence extraction), in that the algorithm tries to find words that are important or seem representative of the entire text. They keywords are not always single words; in the case of multi-word keywords, they are typically all nouns.

Cosine Similarity

Cosine similarity is a metric used to measure how similar the documents are irrespective of their size. Mathematically, it measures the cosine of the angle between two vectors projected in a multi-dimensional space.

The cosine similarity is advantageous because even if the two similar documents are far apart by the Euclidean distance (due to the size of the document), chances are they may still be oriented closer together. The smaller the angle, higher the cosine similarity.

A commonly used approach to match similar documents is based on counting the maximum number of common words between the documents.

But this approach has an inherent flaw. That is, as the size of the document increases, the numbers of common words tend to increase even if the documents talk about different topics.

The cosine similarity helps overcome this fundamental flaw in the ‘count-the-common-words’ or Euclidean distance approach.

## What is Cosine Similarity and why is it advantageous?

Cosine similarity is a metric used to determine how similar the documents are irrespective of their size.

Mathematically, it measures the cosine of the angle between two vectors projected in a multi-dimensional space. In this context, the two vectors I am talking about are arrays containing the word counts of two documents.

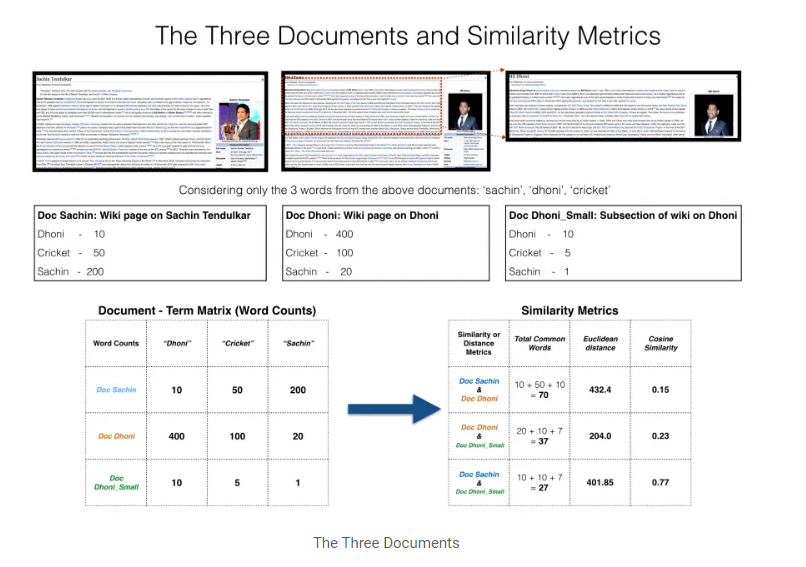
As a similarity metric, how does cosine similarity differ from the number of common words?

When plotted on a multi-dimensional space, where each dimension corresponds to a word in the document, the cosine similarity captures the orientation (the angle) of the documents and not the magnitude. If you want the magnitude, compute the Euclidean distance instead.

The cosine similarity is advantageous because even if the two similar documents are far apart by the Euclidean distance because of the size (like, the word ‘cricket’ appeared 50 times in one document and 10 times in another) they could still have a smaller angle between them. Smaller the angle, higher the similarity.

## Cosine Similarity Example

Let’s suppose you have 3 documents based on a couple of star cricket players – Sachin Tendulkar and Dhoni. Two of the documents (A) and (B) are from the wikipedia pages on the respective players and the third document (C) is a smaller snippet from Dhoni’s wikipedia page.



As you can see, all three documents are connected by a common theme – the game of Cricket.

Our objective is to quantitatively estimate the similarity between the documents.

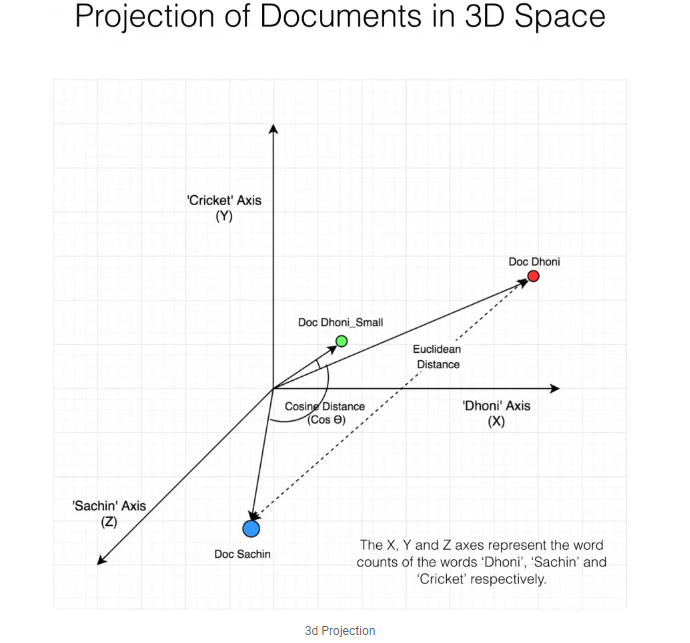
For ease of understanding, let’s consider only the top 3 common words between the documents: ‘Dhoni’, ‘Sachin’ and ‘Cricket’.

You would expect Doc A and Doc C, that is the two documents on Dhoni would have a higher similarity over Doc A and Doc B, because, Doc C is essentially a snippet from Doc A itself.

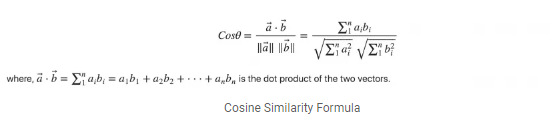
However, if we go by the number of common words, the two larger documents will have the most common words and therefore will be judged as most similar, which is exactly what we want to avoid.

The results would be more congruent when we use the cosine similarity score to assess the similarity.

Let’s project the documents in a 3-dimensional space, where each dimension is a frequency count of either: ‘Sachin’, ‘Dhoni’ or ‘Cricket’. When plotted on this space, the 3 documents would appear something like this.



As you can see, Doc Dhoni\_Small and the main Doc Dhoni are oriented closer together in 3-D space, even though they are far apart by magnitiude. It turns out, the closer the documents are by angle, the higher is the Cosine Similarity (Cos theta).



As you include more words from the document, it’s harder to visualize a higher dimensional space. But you can directly compute the cosine similarity using this math formula.

## How to Compute Cosine Similarity in Python?

To compute the cosine similarity, you need the word count of the words in each document. The CountVectorizer or the TfidfVectorizer from scikit learn. The output of this comes as a sparse\_matrix.

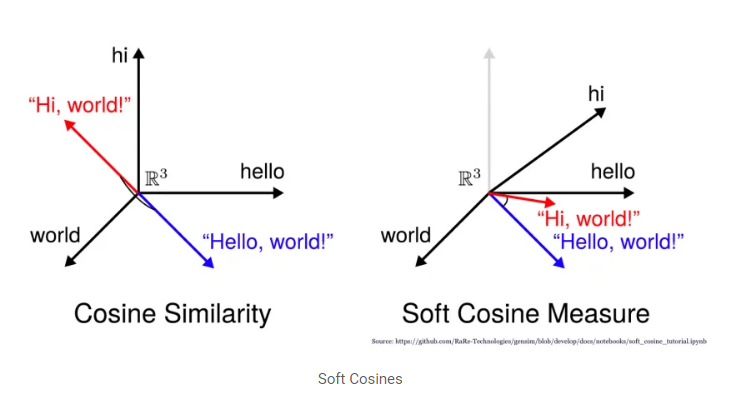
Even better, we could have used the TfidfVectorizer () instead of CountVectorizer (), because it would have downweighted words that occur frequently across documents.

Then, use cosine\_similarity() to get the final output. It can take the document term matrix as a pandas data frame as well as a sparse matrix as inputs.

## Soft Cosine Similarity

Suppose if you have another set of documents on a completely different topic, say ‘food’, you want a similarity metric that gives higher scores for documents belonging to the same topic and lower scores when comparing docs from different topics.

In such case, we need to consider the semantic meaning should be considered. That is, words similar in meaning should be treated as similar. For Example, ‘President’ vs ‘Prime minister’, ‘Food’ vs ‘Dish’, ‘Hi’ vs ‘Hello’ should be considered similar. For this, converting the words into respective word vectors, and then, computing the similarities can address this problem.



Soft cosines can be a great feature if you want to use a similarity metric that can help in clustering or classification of documents.

Word Embeddings

## What is Word Embedding?

## Word Embedding is a type 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.

* It is language modelling and feature learning technique. Word embedding is a way to perform mapping using a neural network. There are various word embedding models available such as word2vec (Google), Glove (Stanford) and fastest (Facebook).
* Word Embedding is also called as distributed semantic model or distributed represented or semantic vector space or vector space model. As you read these names, you come across the word semantic which means categorizing similar words together. For example fruits like apple, mango, banana should be placed close whereas books will be far away from these words. In a broader sense, word embedding will create the vector of fruits which will be placed far away from vector representation of books.

## Where is Word Embedding used?

Word embedding helps in feature generation, document clustering, text classification, and natural language processing tasks. Let us list them and have some discussion on each of these applications.

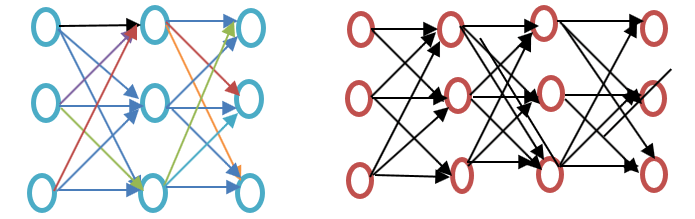
* **Compute similar words:** Word embedding is used to suggest similar words to the word being subjected to the prediction model. Along with that it also suggests dissimilar words, as well as most common words.
* **Create a group of related words:** It is used for semantic grouping which will group things of similar characteristic together and dissimilar far away.
* **Feature for text classification:** Text is mapped into arrays of vectors which is fed to the model for training as well as prediction. Text-based classifier models cannot be trained on the string, so this will convert the text into machine trainable form. Further its features of building semantic help in text-based classification.
* **Document clustering** is another application where word embedding is widely used
* **Natural language processing:** There are many applications where word embedding is useful and wins over feature extraction phases such as parts of speech tagging, sentimental analysis, and syntactic analysis.

Now we have got some knowledge of word embedding. Some light is also thrown on different models to implement word embedding. This whole tutorial is focused on one of the models (word2vec).

## What is word2vec?

Word2vec is the technique/model to produce word embedding for better word representation. It captures a large number of precise syntactic and semantic word relationship. It is a shallow two-layered neural network. Before going further, please see the difference between shallow and deep neural network:

The shallow neural network consists of the only a hidden layer between input and output whereas deep neural network contains multiple hidden layers between input and output. Input is subjected to nodes whereas the hidden layer, as well as the output layer, contains neurons.

[](https://www.guru99.com/images/1/111318_0826_WordEmbeddi1.png)

**Figure: Shallow vs. Deep learning**

word2vec is a two-layer network where there is input one hidden layer and output.

Word2vec was developed by a group of researcher headed by Tomas Mikolov at Google. Word2vec is better and more efficient that latent semantic analysis model.

## What word2vec does?

Word2vec represents words in vector space representation. Words are represented in the form of vectors and placement is done in such a way that similar meaning words appear together and dissimilar words are located far away. This is also termed as a semantic relationship. Neural networks do not understand text instead they understand only numbers. Word Embedding provides a way to convert text to a numeric vector.

Word2vec reconstructs the linguistic context of words. Before going further let us understand, what is linguistic context? In general life scenario when we speak or write to communicate, other people try to figure out what is objective of the sentence. For example, "What is the temperature of India", here the context is the user wants to know "temperature of India" which is context. In short, the main objective of a sentence is context. Word or sentence surrounding spoken or written language (disclosure) helps in determining the meaning of context. Word2vec learns vector representation of words through the contexts.

## Why Word2vec?

### Before Word Embedding

It is important to know which approach is used before word embedding and what are its demerits and then we will move to the topic of how demerits are overcome by Word embedding using word2vec approach. Finally, we will move how word2vec works because it is important to understand it's working.

### Approach for Latent Semantic Analysis

This is the approach which was used before word embedding. It used the concept of Bag of words where words are represented in the form of encoded vectors. It is a sparse vector representation where the dimension is equal to the size of vocabulary. If the word occurs in the dictionary, it is counted, else not. To understand more, please see the below program.

In Latent Semantic approach, the row represents unique words whereas the column represents the number of time that word appears in the document. It is a representation of words in the form of the document matrix. Term-Frequency inverse document frequency (TFIDF) is used to count the frequency of words in the document which is the frequency of the term in the document/ frequency of the term in the entire corpus.

## Shortcoming of Bag of Words method

* It ignores the order of the word, for example, this is bad = bad is this.
* It ignores the context of words. Suppose If I write the sentence "He loved books. Education is best found in books". It would create two vectors one for "He loved books" and other for "Education is best found in books." It would treat both of them orthogonal which makes them independent, but in reality, they are related to each other

To overcome these limitation word embedding was developed and word2vec is an approach to implement such.

## Word Embedding Approaches

One of the reasons that Natural Language Processing is a difficult problem to solve is the fact that, unlike human beings, computers can only understand numbers. We have to represent words in a numeric format that is understandable by the computers. Word embedding refers to the numeric representations of words.

Several word embedding approaches currently exist and all of them have their pros and cons. We will discuss three of them here:

1. Bag of Words
2. TF-IDF Scheme
3. Word2Vec

## Bag of Words

The bag of words approach is one of the simplest word embedding approaches. The following are steps to generate word embeddings using the bag of words approach.

We will see the word embeddings generated by the bag of words approach with the help of an example. Suppose you have a [corpus](https://en.wikipedia.org/wiki/Text_corpus) with three sentences.

* S1 = I love rain
* S2 = rain rain go away
* S3 = I am away

To convert above sentences into their corresponding word embedding representations using the bag of words approach, we need to perform the following steps:

1. Create a dictionary of unique words from the corpus. In the above corpus, we have following unique words: [I, love, rain, go, away, am]
2. Parse the sentence. For each word in the sentence, add 1 in place of the word in the dictionary and add zero for all the other words that don't exist in the dictionary. For instance, the bag of words representation for sentence S1 (I love rain), looks like this: [1, 1, 1, 0, 0, 0]. Similarly for S2 and S3, bag of word representations are [0, 0, 2, 1, 1, 0] and [1, 0, 0, 0, 1, 1], respectively.

Notice that for S2 we added 2 in place of "rain" in the dictionary; this is because S2 contains "rain" twice.

## Pros and Cons of Bag of Words

Bag of words approach has both pros and cons. The main advantage of the bag of words approach is that you do not need a very huge corpus of words to get good results. You can see that we build a very basic bag of words model with three sentences. Computationally, a bag of words model is not very complex.

A major drawback of the bag of words approach is the fact that we need to create huge vectors with empty spaces in order to represent a number (sparse matrix) which consumes memory and space. In the example previous, we only had 3 sentences. Yet you can see three zeros in every vector.

Imagine a corpus with thousands of articles. In such a case, the number of unique words in a dictionary can be thousands. If one document contains 10% of the unique words, the corresponding embedding vector will still contain 90% zeros.

Another major issue with the bag of words approach is the fact that it doesn't maintain any context information. It doesn't care about the order in which the words appear in a sentence. For instance, it treats the sentences "Bottle is in the car" and "Car is in the bottle" equally, which are totally different sentences.

## A type of bag of words approach, known as n-grams, can help maintain the relationship between words. N-gram refers to a contiguous sequence of n words. For instance, 2-grams for the sentence "You are not happy", are "You are", "are not" and "not happy". Although the n-grams approach is capable of capturing relationships between words, the size of the feature set grows exponentially with too many n-grams.

## Bag of words (BOW)

Bag of words is a simple and popular technique for feature extraction from text. **Bag of word model processes the text to find how many times each word appeared in the sentence. This is also called as vectorization.**

Steps for creating BOW

* Tokenize the text into sentences
* Tokenize sentences into words
* Remove punctuation or stop words
* Convert the words to lower text
* Create the frequency distribution of words

## What is the problem with bag of words?

In the bag of words model, each document is represented as a word-count vector. These counts can be binary counts, a word may occur in the text or not or will have absolute counts. The size of the vector is equal to the number of elements in the vocabulary. If most of the elements are zero then the bag of words will be a sparse matrix.

In deep learning, we would have sparse matrix as we will be working with huge amount of training data. Sparse representations are harder to model both for computational reasons as well as for informational reasons.

**Huge amount of weights:**Huge input vectors means a huge number of weights for a neural network.

**Computationally intensive:**More weights means more computation required to train and predict.

**Lack of meaningful relations and no consideration for order of words:**BOW is a collection of words that appear in the text or sentences with the word counts. Bag of words does not take into consideration the order in which they appear.

## TF-IDF Scheme

The TF-IDF scheme is a type of bag words approach where instead of adding zeros and ones in the embedding vector, you add floating numbers that contain more useful information compared to zeros and ones. The idea behind TF-IDF scheme is the fact that words having a high frequency of occurrence in one document, and less frequency of occurrence in all the other documents, are more crucial for classification.

TF-IDF is a product of two values: Term Frequency (TF) and Inverse Document Frequency (IDF).

Term frequency refers to the number of times a word appears in the document and can be calculated as:

Term frequency = (Number of Occurrences of a word)/ (Total words in the document)

For instance, if we look at sentence S1 from the previous section i.e. "I love rain", every word in the sentence occurs once and therefore has a frequency of 1. On the contrary, for S2 i.e. "rain rain go away", the frequency of "rain" is two while for the rest of the words, it is 1.

IDF refers to the log of the total number of documents divided by the number of documents in which the word exists, and can be calculated as:

IDF(word) = Log((Total number of documents)/(Number of documents containing the word))

For instance, the IDF value for the word "rain" is 0.1760, since the total number of documents is 3 and rain appears in 2 of them, therefore log(3/2) is 0.1760. On the other hand, if you look at the word "love" in the first sentence, it appears in one of the three documents and therefore its IDF value is log(3), which is 0.4771.

## Pros and Cons of TF-IDF

Though TF-IDF is an improvement over the simple bag of words approach and yields better results for common NLP tasks, the overall pros and cons remain the same. We still need to create a huge sparse matrix, which also takes a lot more computation than the simple bag of words approach.

## Word Embedding is solution to these problems

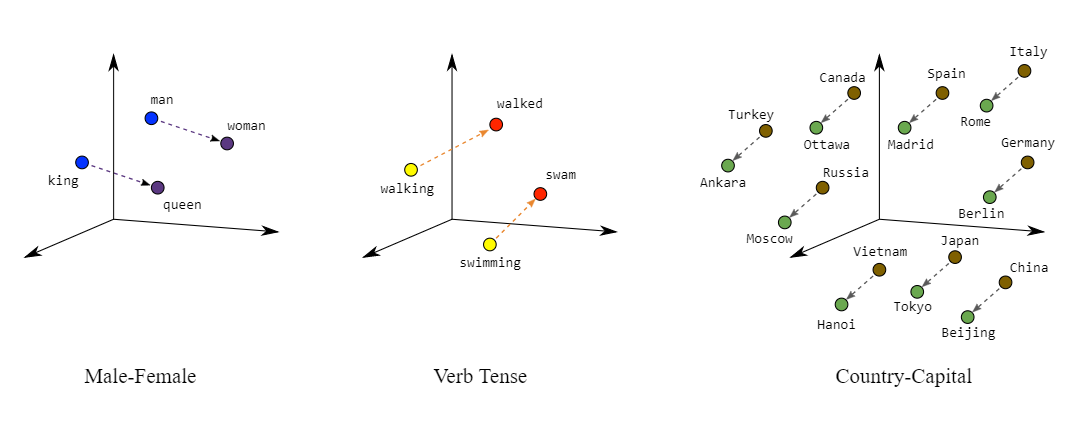
**Embeddings translate large sparse vectors into a lower-dimensional space that preserves semantic relationships**.

Word embeddings is a technique where individual words of a domain or language are represented as real-valued vectors in a lower dimensional space.

**Sparse Matrix problem with BOW is solved by mapping high-dimensional data into a lower-dimensional space.**

Lack of meaningful relationship issue of BOW is solved by placing **vectors of semantically similar items close to each other**. This way words that have similar meaning have similar distances in the vector space as shown below.

king is to queen as man is to woman” encoded in the vector space as well as verb Tense and Country and their capitals are encoded in low dimensional space preserving the semantic relationships.



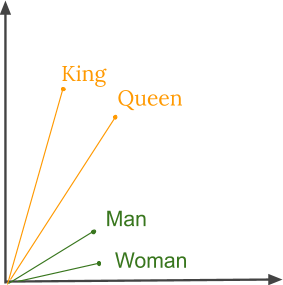
## Using standard Dimensionality reduction techniques

**Standard dimensionality reduction techniques like Principal Component Analysis(PCA) can be used to create word embeddings**. PCA tries to find highly correlated dimensions that can be collapsed into a single dimension using the BOW.

## What is Word Embedding?

Humans have always excelled at understanding languages. It is easy for humans to understand the relationship between words but for computers, this task may not be simple. For example, we humans understand the words like king and queen, man and woman, tiger and tigress have a certain type of relation between them.

Word embeddings are basically a form of word representation that bridges the human understanding of language to that of a machine. They have learned representations of text in an n-dimensional space where words that have the same meaning have a similar representation. Meaning that two similar words are represented by almost similar vectors that are very closely placed in a vector space. These are essential for solving most [Natural language processing](https://www.mygreatlearning.com/blog/natural-language-processing-infographic/) problems.



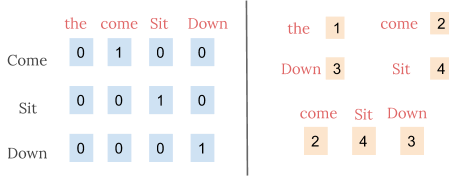
Thus when using word embeddings, all individual words are represented as real-valued vectors in a predefined vector space. Each word is mapped to one vector and the vector values are learned in a way that resembles a neural network.

Word2Vec is one of the most popular techniques to learn word embeddings using shallow neural network. It was developed by Tomas Mikolov in 2013 at Google.

## Why Word Embeddings are used?

As we know the machine learning models cannot process text so we need to figure out a way to convert these textual data into numerical data. Previously techniques like [Bag of Words](https://www.mygreatlearning.com/blog/bag-of-words/) and [TF-IDF](https://www.mygreatlearning.com/blog/bag-of-words/) have been discussed that can help achieve use this task. Apart from this, we can use two more techniques such as one-hot encoding, or we can use unique numbers to represent words in a vocabulary. The latter approach is more efficient than one-hot encoding as instead of a sparse vector, we now have a dense one. Thus this approach even works when our vocabulary is large.

In the below example, we assume we have a small vocabulary containing just four words, using the two techniques we represent the sentence ‘Come sit down’.

*One-hot encoding vs integer encoding*

However, the integer-encoding is arbitrary as it does not capture any relationship between words. It can be challenging for a model to interpret, for example, a linear classifier learns a single weight for each feature. Because there is no relationship between the similarity of any two words and the similarity of their encodings, this feature-weight combination is not meaningful.

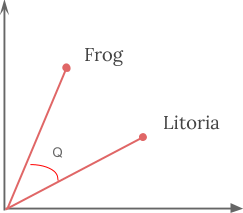
Thus by using word embeddings, words that are close in meaning are grouped near to one another in vector space. For example, while representing a word such as frog, the nearest neighbour of a frog would be frogs, toads, Litoria. This implies that it is alright for a classifier to not see the word Litoria and only frog during training, and the classifier would not be thrown off when it sees Litoria during testing because the two-word vectors are similar. Also, word embeddings learn relationships. Vector differences between a pair of words can be added to another word vector to find the analogous word. For example, “man” -“woman” + “queen” ≈ “king”.

## What is word2Vec?

Word2vec is a method to efficiently create word embeddings by using a two-layer neural network. It was developed by Tomas Mikolov, et al. at Google in 2013 as a response to make the neural-network-based training of the embedding more efficient and since then has become the de facto standard for developing pre-trained word embedding.

The input of word2vec is a text corpus and its output is a set of vectors known as feature vectors that represent words in that corpus. While Word2vec is not a deep neural network, it turns text into a numerical form that deep neural networks can understand.

The Word2Vec objective function causes the words that have a similar context to have similar embeddings. Thus in this vector space, these words are really close. Mathematically, the cosine of the angle (Q) between such vectors should be close to 1, i.e. angle close to 0.



## How Word2vec works?

Word2vec learns word by predicting its surrounding context. For example, let us take the word "He **loves**Football."

We want to calculate the word2vec for the word: loves.

Suppose

loves = Vin. P(Vout / Vin) is calculated

where,

Vin is the input word.

P is the probability of likelihood.

Vout is the output word.

Word **loves**moves over each word in the corpus. Syntactic as well as the Semantic relationship between words is encoded. This helps in finding similar and analogies words.

All random features of the word **loves** is calculated. These features are changed or update concerning neighbour or context words with the help of a back propagation method.

Another way of learning is that if the context of two words are similar or two words have similar features, then such words are related.

## Word2vec Architecture

There are two architectures used by word2vec

1. Continuous Bag of words (CBOW)
2. skip gram

Before going further, let us discuss why these architectures or models are important from word representation point of view. Learning word representation is essentially unsupervised, but targets/labels are needed to train the model. Skip-gram and CBOW convert unsupervised representation to supervised form for model training.

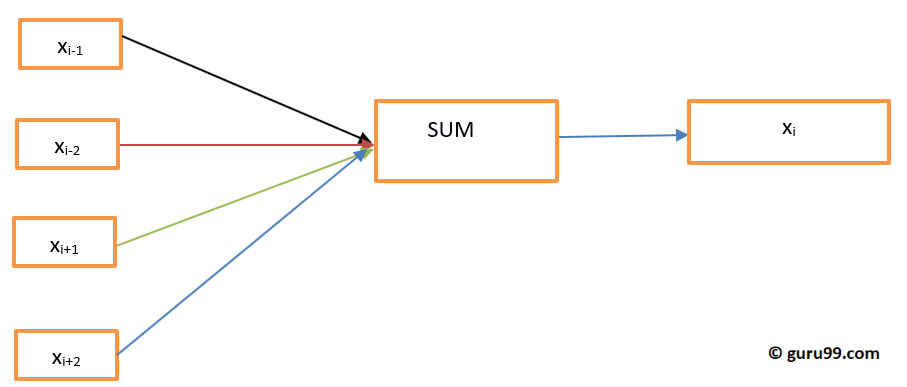
In CBOW, the current word is predicted using the window of surrounding context windows. For example, if wi-1,wi-2,wi+1,wi+2are given words or context, this model will provide wi

Skip-Gram performs opposite of CBOW which implies that it predicts the given sequence or context from the word. You can reverse the example to understand it. If wi is given, this will predict the context or wi-1,wi-2,wi+1,wi+2.

Word2vec provides an option to choose between CBOW (continuous Bag of words) and skim-gram. Such parameters are provided during training of the model. One can have the option of using negative sampling or hierarchical softmax layer.

### Continuous Bag of Words

Let us draw a simple diagram to understand the continuous bag of word architecture.

[](https://www.guru99.com/images/1/111318_0826_WordEmbeddi3.png)

Let us calculate the equations mathematically. Suppose V is the vocabulary size and N is the hidden layer size. Input is defined as { xi-1, xi-2, xi+1, xi+2}. We obtain the weight matrix by multiplying V \* N. Another matrix is obtained by multiplying input vector with the weight matrix. This can also be understood by the following equation.

h=xitW

where xit∧ W are the input vector and weight matrix respectively,

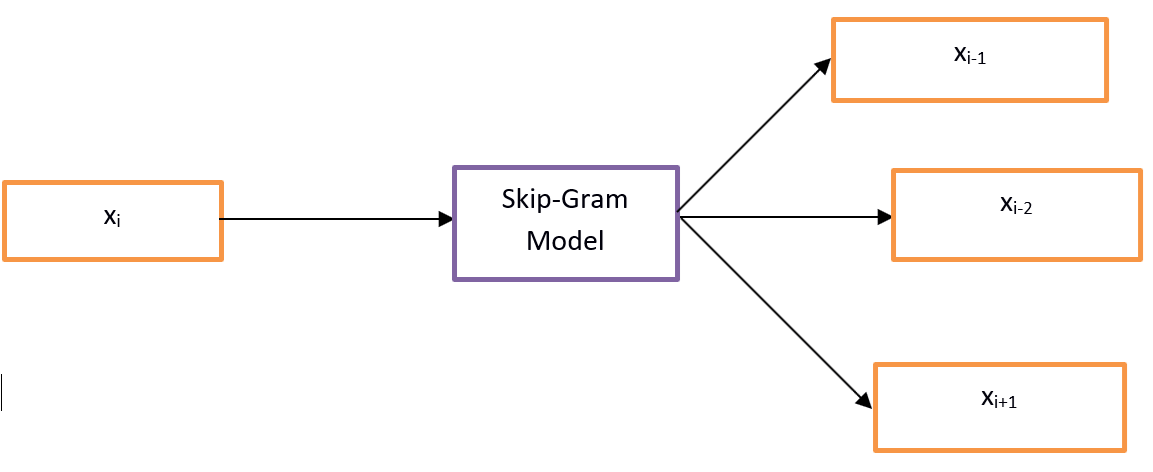
To calculate the match between context and the next word, please refer to the below equation

u=predictedrepresentation\*h

where predictedrepresentation is obtained model∧h in the above equation.

### Skip-Gram Model

Skip-Gram approach is used to predict a sentence given an input word. To understand it better let us draw the diagram.

[](https://www.guru99.com/images/1/111318_0826_WordEmbeddi4.png)

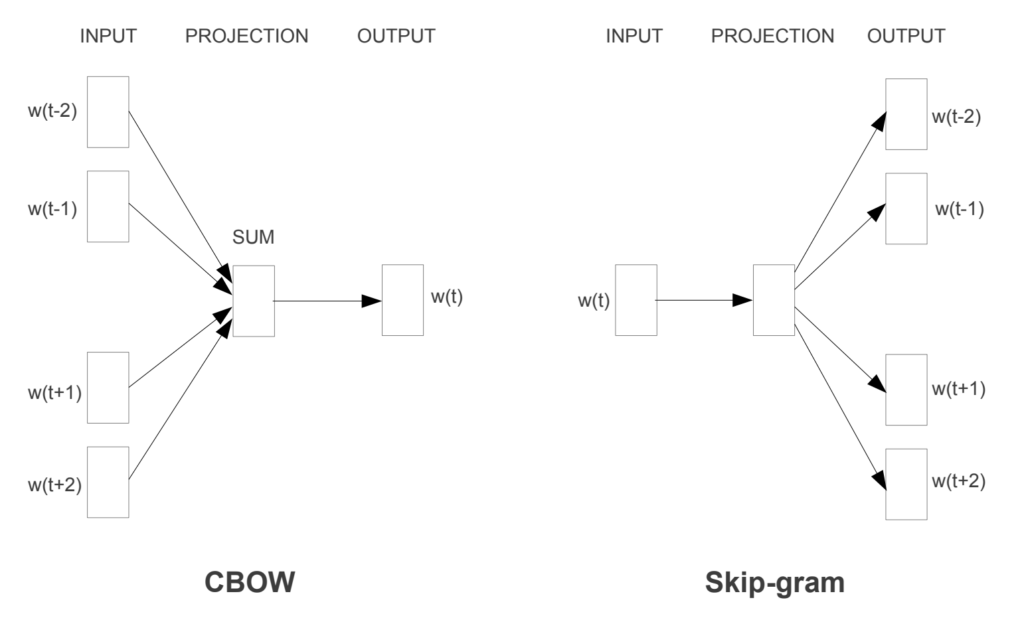
**Figure Skip-Gram Model**

One can treat it as the reverse of the Continuous bag of word model where the input is the word and model provides the context or the sequence. We can also conclude that the target is fed to the input and output layer is replicated multiple times to accommodate the chosen number of context words. Error vector from all the output layer is summed up to adjust weights via a backpropagation method.

### Which model to choose?

CBOW is several times faster than skip gram and provides a better frequency for frequent words whereas skip gram needs a small amount of training data and represents even rare words or phrases.

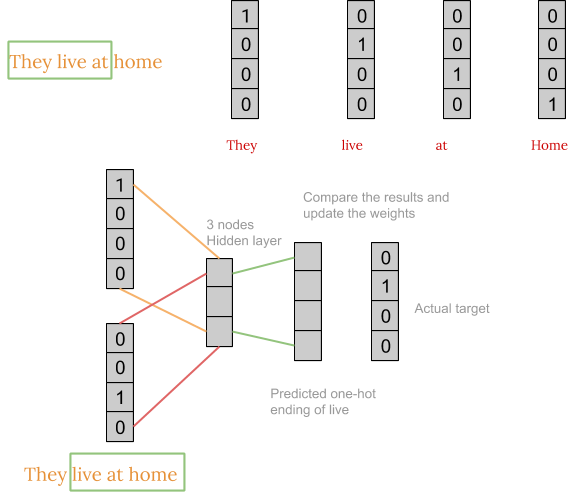
Word2vec is not a single algorithm but a combination of two techniques – CBOW (Continuous bag of words) and Skip-gram model. Both of these are shallow neural networks which map word(s) to the target variable which is also a word(s). Both of these techniques learn weights which act as word vector representations.



## Continuous Bag-of-Words model (CBOW)

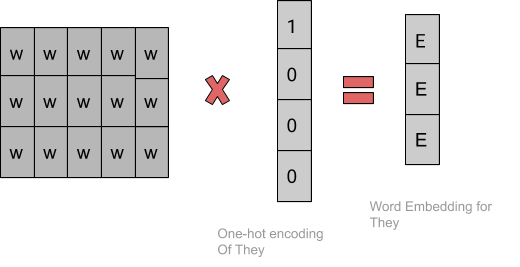
CBOW predicts the probability of a word to occur given the words surrounding it. We can consider a single word or a group of words. But for simplicity, we will take a single context word and try to predict a single target word.

The English language contains almost 1.2 million words, making it impossible to include so many words in our example. So I ‘ll consider a small example in which we have only four words i.e. live, home, they and at. For simplicity, we will consider that the corpus contains only one sentence, that being, ‘They live at home’.



First, we convert each word into a one-hot encoding form. Also, we’ll not consider all the words in the sentence but ll only take certain words that are in a window. For example for a window size equal to three, we only consider three words in a sentence. The middle word is to be predicted and the surrounding two words are fed into the neural network as context. The window is then slid and the process is repeated again.

Finally, after training the network repeatedly by sliding the window a shown above, we get weights which we use to get the embeddings as shown below.

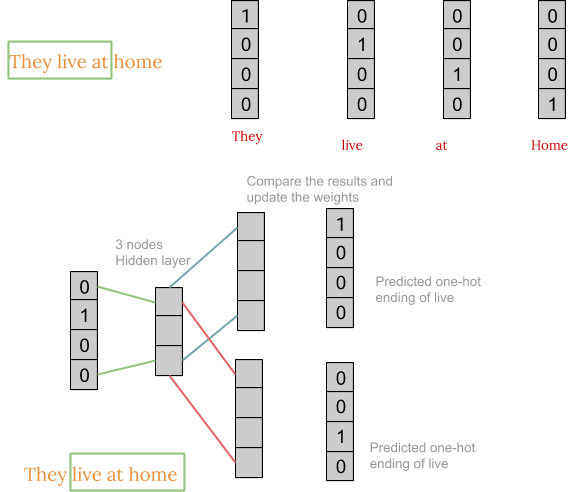


Usually, we take a window size of around 8-10 words and have a vector size of 300.

## Skip-gram model

The Skip-gram model architecture usually tries to achieve the reverse of what the CBOW model does. It tries to predict the source context words (surrounding words) given a target word (the centre word)

The working of the skip-gram model is quite similar to the CBOW but there is just a difference in the architecture of its neural network and the way the weight matrix is generated  as shown in the figure below:



After obtaining the weight matrix, the steps to get word embedding is same as CBOW.

Turns out for large corpus with higher dimensions, it is better to use skip-gram but is slow to train. Whereas CBOW is better for small corpus and is faster to train too.

## Pros and Cons of Word2Vec

Word2Vec has several advantages over bag of words and IF-IDF scheme. Word2Vec retains the semantic meaning of different words in a document. The context information is not lost. Another great advantage of Word2Vec approach is that the size of the embedding vector is very small. Each dimension in the embedding vector contains information about one aspect of the word. We do not need huge sparse vectors, unlike the bag of words and TF-IDF approaches.

## GloVe

GloVe (Global Vectors for Word Representation) is an alternate method to create word embeddings. It is based on matrix factorization techniques on the word-context matrix. A large matrix of co-occurrence information is constructed and you count each “word” (the rows), and how frequently we see this word in some “context” (the columns) in a large corpus. Usually, we scan our corpus in the following manner: for each term, we look for context terms within some area defined by a window-size before the term and a window-size after the term. Also, we give less weight for more distant words.

The number of “contexts” is, of course, large, since it is essentially combinatorial in size. So then we factorize this matrix to yield a lower-dimensional matrix, where each row now yields a vector representation for each word. In general, this is done by minimizing a “reconstruction loss”. This loss tries to find the lower-dimensional representations which can explain most of the variance in the high-dimensional data.

We use both GloVe and Word2Vec to convert our text into embeddings and both exhibit comparable performances. Although in real applications we train our model over Wikipedia text with a window size around 5- 10. The number of words in the corpus is around 13 million, hence it takes a huge amount of time and resources to generate these embeddings. To avoid this we can use the pre-trained word vectors that are already trained and we can easily use them.

## The relation between Word2vec and NLTK

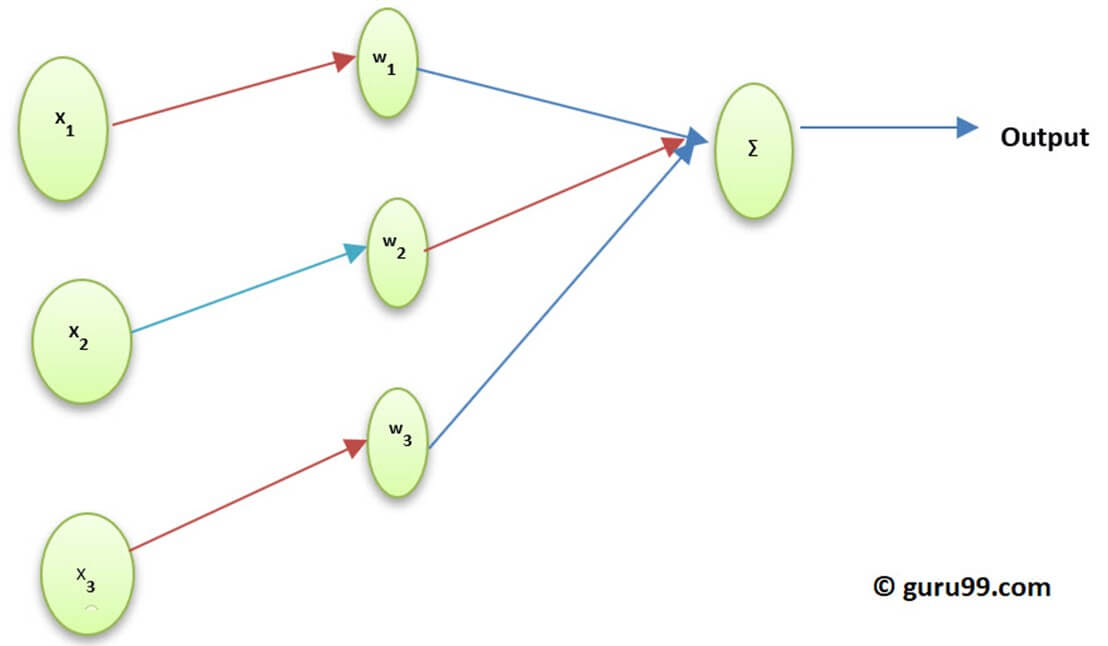
[NLTK](https://www.guru99.com/nltk-tutorial.html) is natural Language toolkit. It is used for preprocessing of the text. One can do different operations such as parts of speech tagging, lemmatizing, stemming, stop words removal, removing rare words or least used words. It helps in cleaning the text as well as helps in preparing the features from the effective words. In the other way, word2vec is used for semantic (closely related items together) and syntactic (sequence) matching. Using word2vec, one can find similar words, dissimilar words, dimensional reduction, and many others. Another important feature of word2vec is to convert the higher dimensional representation of the text into lower dimensional of vectors.

### Where to use NLTK and Word2vec?

If one has to accomplish some general-purpose tasks as mentioned above like tokenization, POS tagging and parsing one must go for using NLTK whereas for predicting words according to some context, topic modeling, or document similarity one must use Word2vec.

Activators and Word2Vec

The activation function of the neuron defines the output of that neuron given a set of inputs. Biologically inspired by an activity in our brains where different neurons are activated using different stimuli. Let us understand the activation function through the following diagram.

[](https://www.guru99.com/images/1/111318_0826_WordEmbeddi8.jpg)

**Figure Understanding Activation function**

Here x1,x2,..x4 is the node of the neural network.

w1, w2, w3 is the weight of the node,

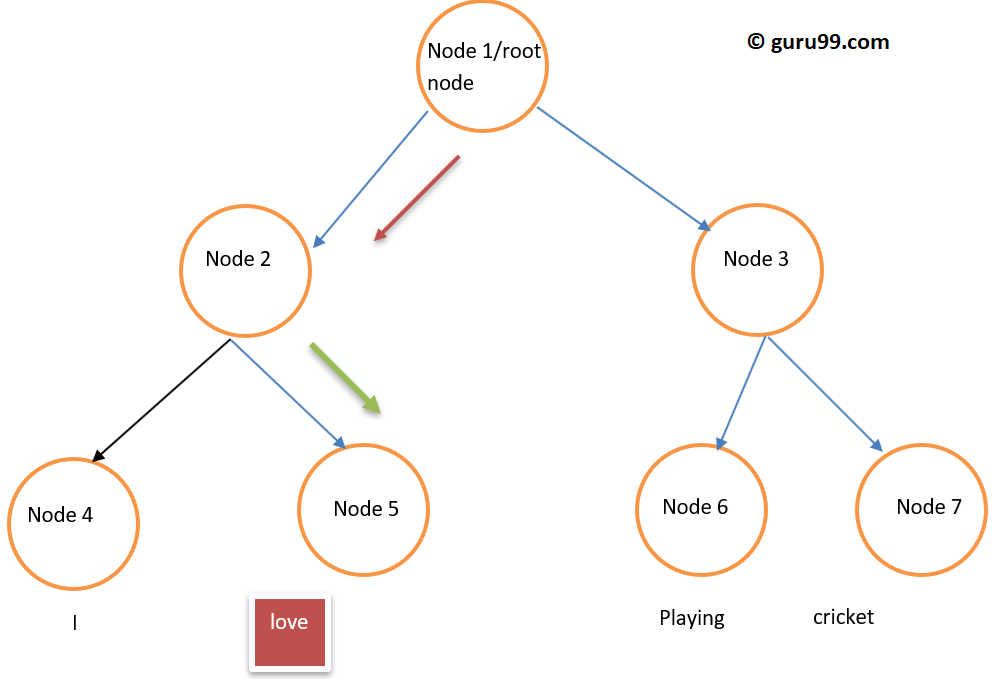
∑ is the summation of all weight and node value which work as the activation function.

### Why Activation function?

If no activation function is used output would be linear but the functionality of linear function is limited. To achieve complex functionality such as object detection, image classification, typing text using voice and many other non-linear outputs is needed which is achieved using activation function.

### How the activation layer is computed in the word embedding (word2vec)

Softmax Layer (normalized exponential function) is the output layer function which activates or fires each node. Another approach used is Hierarchical softmax where the complexity is calculated by O(log2V) wherein the softmax it is O(V) where V is the vocabulary size. The difference between these is the reduction of the complexity in hierarchical softmax layer. To understand its (Hierarchical softmax) functionality, please look at the below example:

[](https://www.guru99.com/images/1/111318_0826_WordEmbeddi9.png)

**Figure Hierarchical softmax tree like structure**

Suppose we want to compute the probability of observing the word **love** given a certain context. The flow from the root to the leaf node will be the first move to node 2 and then to node 5. So if we have had the vocabulary size of 8, only three computations are needed. So it allows decomposing, calculation of the probability of one word (**love**).

### What other options are available other than Hierarchical Softmax?

If speaking in a general sense for word embedding options available are Differentiated Softmax, CNN-Softmax, Importance Sampling, Adaptive Importance sampling, Noise Contrastive Estimations, Negative Sampling, Self-Normalization, and infrequent Normalization.

Speaking specifically about Word2vec we have negative sampling available.

Negative Sampling is a way to sample the training data. It is somewhat like stochastic gradient descent, but with some difference. Negative sampling looks only for negative training examples. It is based on noise contrastive estimation and randomly samples words, not in the context. It is a fast training method and chooses the context randomly. If the predicted word appears in the randomly chosen context both the vectors are close to each other.

### What conclusion can be drawn?

Activators are firing the neurons just like our neurons are fired using the external stimuli. Softmax layer is one of the output layer function which fires the neurons in case of word embedding. In word2vec we have options such as hierarchical softmax and negative sampling. Using activators, one can convert the linear function into the nonlinear function, and a complex machine learning algorithm can be implemented using such.

Pre-trained Word Embeddings

The idea is to create a representation of words that capture their meanings, semantic relationships and the different types of contexts they are used in. That’s what word embeddings are – the numerical representation of a text.

Pre-trained word embeddings are vector representation of words trained on a large dataset. With pre-trained embeddings, you will essentially be using the weights and vocabulary from the end result of the training process done by….someone else

One benefit of using pre-trained embeddings is that you can **hit the ground running** without the need for finding a large text corpora which you will have to [preprocess](http://kavita-ganesan.com/text-preprocessing-tutorial/) and train with the appropriate settings.

Another benefit is the **savings in training time**. Training on a large corpora could demand high computation power and long training times which may not be something that you want to afford for quick experimentation.

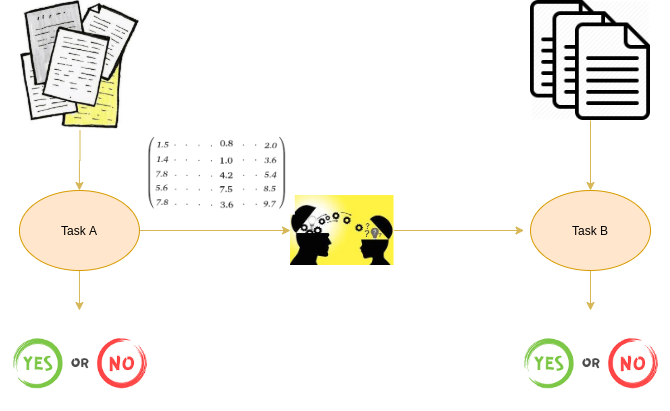
If you want to avoid all of these logistics but still have access to good quality embeddings, you could use **pre-trained word embeddings** trained on a**dataset** that fits the domain you are working in.

For example, if you are working with news articles, it may be perfectly fine to use embeddings trained on a Twitter dataset as there is ongoing discussion about current issues as well as a constant stream of news related Tweets.

**Accessing pre-trained embeddings is extremely easy with**[Gensim](https://github.com/RaRe-Technologies/gensim-data) as it allows you to use pre-trained [GloVe](https://nlp.stanford.edu/pubs/glove.pdf) and [Word2Vec](https://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf) embeddings with minimal effort.

## What are Pre trained Word Embeddings?

Pre -trained Word Embeddings are the embeddings learned in one task that are used for solving another similar task. These embeddings are trained on large datasets, saved, and then used for solving other tasks. That’s why pretrained word embeddings are a form of **Transfer Learning.**



Transfer learning, as the name suggests, is about transferring the learnings of one task to another. Learnings could be either weights or embeddings. In our case here, learnings are the embeddings. Hence, this concept is known as pretrained word embeddings. In the case of weights, the concept is known as a [pretrained model](https://www.analyticsvidhya.com/blog/2019/03/pretrained-models-get-started-nlp/?utm_source=blog&utm_medium=pretrained-word-embeddings-nlp" \t "_blank).

## Why do we need Pretrained Word Embeddings?

Pretrained word embeddings capture the semantic and syntactic meaning of a word as they are trained on large datasets. They are capable of boosting the performance of a [Natural Language Processing (NLP)](https://courses.analyticsvidhya.com/courses/natural-language-processing-nlp?utm_source=blog&utm_medium=pretrained-word-embeddings-nlp) model.

Learning word embeddings from scratch is a challenging problem due to two primary reasons:

* Sparsity of training data
* Large number of trainable parameters

Sparsity of training data

One of the primary reasons for not doing this is the Sparsity of Training Data. Most real-world problems contain a dataset that has a **large volume of rare words**. The embeddings learned from these datasets cannot arrive at the right representation of the word.

In order to achieve this, the dataset must contain a rich vocabulary. Frequently occurring words build just such a rich vocabulary.

**Large number of trainable parameters**

Secondly, the number of Trainable Parameters increases while learning embeddings from scratch. This results in a slower training process. Learning embeddings from scratch might also leave you in an unclear state about the representation of the words. So, the solution to all the above problems is pretrained word embeddings.

## What are the Different Pretrained Word Embeddings?

We divide the embeddings into 2 classes: **Word-level and Character-level embeddings**. [ELMo](https://www.analyticsvidhya.com/blog/2019/03/learn-to-use-elmo-to-extract-features-from-text/?utm_source=blog&utm_medium=pretrained-word-embeddings-nlp" \t "_blank) and [Flair](https://www.analyticsvidhya.com/blog/2019/02/flair-nlp-library-python/?utm_source=blog&utm_medium=pretrained-word-embeddings-nlp) embeddings are examples of Character-level embeddings. In this article, we are going to cover two popular word-level pretrained word embeddings:

* Gooogle’s Word2Vec
* Stanford’s GloVe

### Google’s Word2vec Pretrained Word Embedding

Word2Vec is one of the most popular pretrained word embeddings developed by Google. Word2Vec is trained on the Google News dataset (about 100 billion words). It has several use cases such as [Recommendation Engines](https://www.analyticsvidhya.com/blog/2018/06/comprehensive-guide-recommendation-engine-python/?utm_source=blog&utm_medium=pretrained-word-embeddings-nlp), Knowledge Discovery, and also applied in the different [Text Classification](https://www.analyticsvidhya.com/blog/2018/04/a-comprehensive-guide-to-understand-and-implement-text-classification-in-python/?utm_source=blog&utm_medium=pretrained-word-embeddings-nlp) problems.

The architecture of Word2Vec is really simple. It’s a feed-forward neural network with just one hidden layer. Hence, it is sometimes referred **to as a Shallow Neural Network architecture.**

**Depending on the way the embeddings are learned, Word2Vec is classified into two approaches:**

* **Continuous Bag-of-Words (CBOW)**
* **Skip-gram model**

Continuous Bag-of-Words (CBOW) model learns the focus word given the neighboring words whereas the Skip-gram model learns the neighboring words given the focus word. Continous Bag Of Words and Skip-gram are inverses of each other

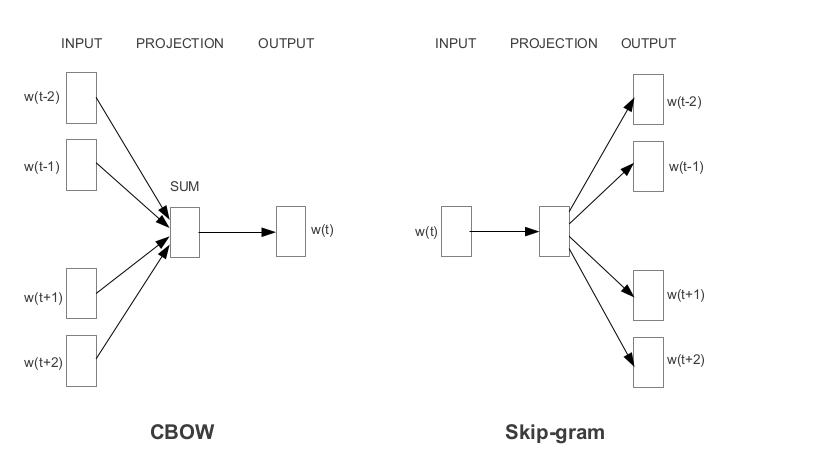
For example, consider the sentence: “I have failed at times but I never stopped trying”.  Let’s say we want to learn the embedding of the word “failed”. So, here the focus word is “failed”.

The first step is to define a context window. A context window refers to the number of words appearing on the left and right of a focus word. The words appearing in the context window are known as neighboring words (or context). Let’s fix the context window to 2 and then input and output pairs for both approaches:

* Continuous Bag-of-Words: Input = [ I, have, at, times ],  Output = failed
* Skip-gram: Input = failed, Output = [I, have, at, times ]

As you can see here, CBOW accepts multiple words as input and produces a single word as output whereas Skip-gram accepts a single word as input and produces multiple words as output.

So, let us define the architecture according to the input and output. But keep in mind that each word is fed into a model as a one-hot vector:



### Stanford’s GloVe Pretrained Word Embedding

The basic idea behind the GloVe word embedding is to derive the relationship between the words from Global Statistics

One of the simplest ways is to look at the co-occurrence matrix. **A co-occurrence matrix tells us how often a particular pair of words occur together. Each value in a co-occurrence matrix is a count of a pair of words occurring together.**

For example, consider a corpus: “I play cricket, I love cricket and I love football”. The co-occurrence matrix for the corpus looks like this:



Now, we can easily compute the probabilities of a pair of words. Just to keep it simple, let’s focus on the word “cricket”:

p(cricket/play)=1

p(cricket/love)=0.5

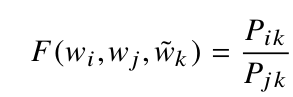
Next, let’s compute the ratio of probabilities:

p(cricket/play) / p(cricket/love) = 2

As the ratio > 1, we can infer that the most relevant word to cricket is “play” as compared to “love”. Similarly, if the ratio is close to 1, then both words are relevant to cricket.

We are able to derive the relationship between the words using simple statistics. This the idea behind the GloVe pretrained word embedding.

GloVe learns to encode the information of the probability ratio in the form of word vectors. The most general form of the model is given by:



## Conclusion

* Word Embedding is a type of word representation that allows words with similar meaning to be understood by machine learning algorithms
* Word Embedding is used to compute similar words, Create a group of related words, Feature for text classification, Document clustering, Natural language processing
* Word2vec is a shallow two-layered neural network model to produce word embedding for better word representation
* Word2vec represents words in vector space representation. Words are represented in the form of vectors and placement is done in such a way that similar meaning words appear together and dissimilar words are located far away
* Word2vec used 2 architectures Continuous Bag of words (CBOW) and skip gram
* CBOW is several times faster than skip gram and provides a better frequency for frequent words whereas skip gram needs a small amount of training data and represents even rare words or phrases.
* NLTK and word2vec can be used together create powerful applications
* The activation function of the neuron defines the output of that neuron given a set of inputs. In word2vec. Softmax Layer (normalized exponential function) is the output layer function which activates or fires each node. Word2vec also has negative sampling available
* Gensim is a topic modeling toolkit which is implemented in python