Word Embeddings

* Word embeddings are a modern approach for representing text in natural language processing.
* [Word embedding algorithms](https://machinelearningmastery.com/what-are-word-embeddings/) like word2vec and GloVe are key to the state-of-the-art results achieved by neural network models on natural language processing problems like machine translation.
* How to train your own word2vec word embedding model on text data.
* How to visualize a trained word embedding model using Principal Component Analysis.
* How to load pre-trained word2vec and GloVe word embedding models from Google and Stanford.

This tutorial is divided into 6 parts; they are:

1. Word Embeddings
2. Gensim Library
3. Develop Word2Vec Embedding
4. Visualize Word Embedding
5. Load Google’s Word2Vec Embedding
6. Load Stanford’s GloVe Embedding

## Most Popular Word Embedding Techniques

* To build any model in [machine learning](https://dataaspirant.com/category/machine-learning-2/) or deep learning, the final level data has to be in numerical form, because models don’t understand text or image data directly like humans do.
* So how [natural language processing](https://dataaspirant.com/category/natural-language-processing/) (NLP) models learn patterns from text data ?
* We need smart ways to convert the text data into numerical data, which is called **vectorization** or in the NLP world, it is called **word embeddings.**
* Vectorization or word embedding is nothing but the process of converting text data to numerical vectors. Later the numerical vectors are used to build various machine learning models. In a way, we say this as **extracting features** from text to build multiple natural language processing models.
* We have numerous ways to convert the text data to numerical vectors.

### Word embedding techniques

Below are the popular and simple word embedding methods to extract features from text are

* Bag of words
* TF-IDF
* Word2vec
* Glove embedding
* Fastext
* ELMO (Embeddings for Language models)

But in this article, we will learn only the popular word embedding techniques, such as a bag of words, TF-IDF, Word2vec.

Bag of words

****

The bag of words method is simple to understand and easy to implement. This method is mostly used in language modeling and text classification tasks. The concept behind this method is straightforward. In this method, we will represent sentences into vectors with the frequency of words that are occurring in those sentences.

### Bag of words approach

In this approach we perform two operations.

1. Tokenization
2. Vectors Creation

Tokenization

The process of dividing each sentence into **words** or smaller parts. Here each word or symbol is called a **token**. After tokenization we will take unique words from the corpus. Here **corpus** means the tokens we have from all the documents we are considering for the bag of words creation.

Create vectors for each sentence

Here the size of the vector is equal to the number of unique words of the corpus. For each sentence we will fill each position of a vector with corresponding word frequency in a particular sentence.

Let's understand this with an example

1. This pasta is very tasty and affordable.
2. This pasta is not tasty and is affordable.
3. This pasta is very very delicious.

These **3 sentences** are example sentences; our first step is to perform tokenization. Before tokenization we have to convert all sentences to lowercase letters or uppercase letters for normalization, we will convert all the words in the sentences to lowercase.

These **3 sentences** are example sentences; our first step is to perform tokenization. Before tokenization we have to convert all sentences to lowercase letters or uppercase letters for normalization, we will convert all the words in the sentences to lowercase.

##### Output of sentences after converting to lowercase

* this pasta is very tasty and affordable.
* this pasta is not tasty and is affordable.
* this pasta is very very delicious.

Now we will perform **tokenization**.

Dividing sentences into words and creating a list with all unique words and also in  **alphabetical** order.

We will get the below output after the tokenization step.

[*“and”, “affordable.”, “delicious.”, “is”, “not”, “pasta”, “tasty”, “this”, “very”*]

*Now what is our next step?*

*Creating vectors for each sentence with frequency of words. This is called a****sparse matrix****. Below is the sparse matrix of example sentences.*

****

We can see in the above figure, every sentence converting into vectors. We can also find sentence similarities after converting sentences to vectors.

How can we find similarities ? Just calculating distance between any two vectors of  sentences by using any distance measure method for example [Euclidean Distance](https://dataaspirant.com/five-most-popular-similarity-measures-implementation-in-python/)

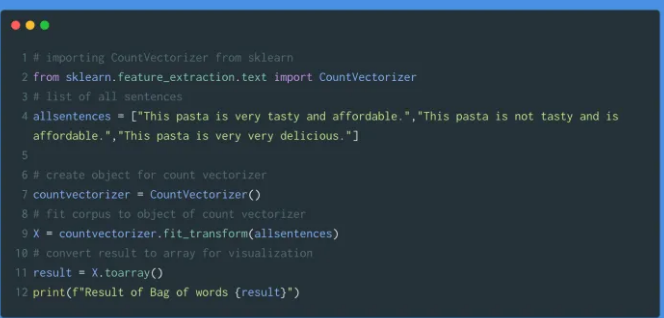
In the above example we are just taking each word as a feature, another name for this is 1-gram representence, we can also take bigram words , tri-Gram words etc .

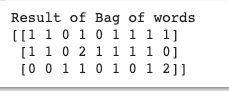
Examples for Bi-Gram word representation  of the first sentence as below.

* this, pasta
* pasta, is
* is, very
* very, tasty
* tasty, and
* and affordable

Like this we can take more tri-gram words and n-gram words etc, here n is the number of words to split. But we can not get any semantic meaning or relation between words from the bag of words technique.

In Bag of word representation we have more zeros in the sparse matrices. The size of the matrix  will be increased based on the total number of words in the corpus. In real world applications corpus will contain thousands of words. So we need more resources to build analytics models with  this type of technique for large datasets. This drawback will be overcome in the next word embedding techniques. Now let’s learn how to implement the bag of words technique in python with Sklearn

****



ONE HOT ENCODING ( BOW / COUNT VECTORIZER )

One hot encoding as the name suggest converts the text data in to 0 and 1 representation. It simply take the count of word in a corpus and convert the data into binary form that is why it is called binary embedding also. The final representation of the text data would be a matrix and it would be the vector representation of the data. Steps we can use to convert the data are listed below.

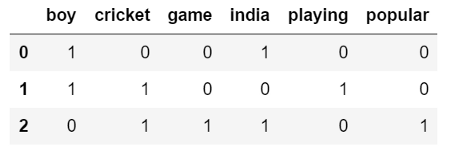
* Tokenize the text into words
* Convert the data to lower
* Preprocess the data with punctuation and stop words removal
* Create the frequency distribution of words using count vectorizer

Consider a corpus X with Y documents. First we will extract N unique words and it will form a matrix of dimension Y X N.

Code Representation

In this code, we will use use CountVectorizer from sklearn library, it tokenizes text documents, builds bag of word and convert one hot encoded data.

from sklearn.feature\_extraction.text import CountVectorizer  
from nltk.tokenize import sent\_tokenize  
from nltk.tokenize import word\_tokenize  
import pandas as pd  
from nltk.corpus import stopwords  
stopwords = set(stopwords.words('english'))  
text = ['He is a boy and he is from India.','The boy is Playing The Cricket.','The Cricket is the most popular game in India.']  
vectorizer = CountVectorizer(text,stop\_words=stopwords)  
sentence\_vectors = vectorizer.fit\_transform(text)  
feature\_names = vectorizer.get\_feature\_names()  
dense = sentence\_vectors.todense()  
denselist = dense.tolist()  
df = pd.DataFrame(denselist, columns=feature\_names)

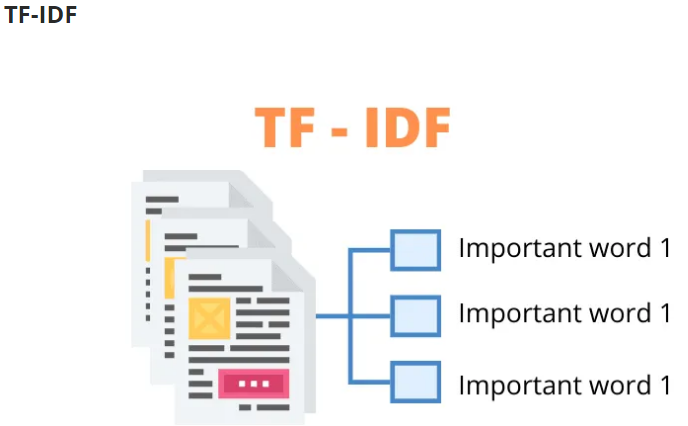


Output of df

We can see that the unique word list length 6 after removing the stopwords. So our matrix output is 3x6. We can also see that in each raw = sentence which words are taken in unique list are presented with one hot encoded values. In first sentence unique words take are boy and india so those are presented in first raw with value 1. We use the Bag of Words model to extract features from the text convert text into matrix of occurrences of words within document.

PROBLEMS WITH COUNT VECTORIZE

Here we can see the matrix generated is sparse matrix. Also, it can not cater sematic information. All the words which are used in the sentences are having 1 value because of this we can not find meaningful words out of the sentences. Also the matrix is having high dimension so it is computationally expensive. To solve this issue we will take a look on another method called TF-IDF

****

Another popular word embedding technique for extracting features from corpus or vocabulary is TF-IDF. This is a statistical method to find how important a word is to a document all over other documents

### TF

The full form of TF is **Term Frequency** (TF). In TF , we are giving some scoring for each word or token based on the frequency of that word. The frequency of a word is dependent on the length of the document. Means in large size of document a word occurs more than a small or medium size of the documents.

So to overcome this problem we will divide the **frequency of a word** with the length of the document (total number of words) to normalize. By using this technique also, we are creating a sparse matrix with frequency of every word.

Formula to calculate Term Frequency (TF)

**TF =** *no. of times term occurrences in a document / total number of words in a document*

### IDF

The full form of IDF is **Inverse Document Frequency**. Here also we are assigning  a score value  to a word , this scoring value explains how a word is rare across all documents. Rarer words have more IDF score.

Formula to calculate Inverse Document Frequency (IDF) :-

**IDF =** *log base e (total number of documents / number of documents which are having term )*

Formula to calculate complete TF-IDF value is

*TF - IDF  = TF \* IDF*

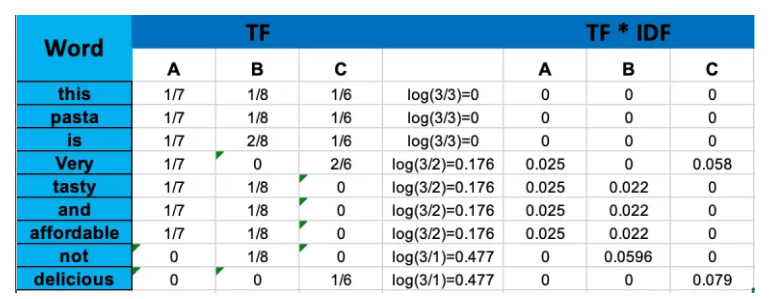
TF-IDF value will be increased based on frequency of the word in a document. Like Bag of Words in this technique also we can not get any semantic meaning for words.

But this technique is mostly used for document classification and also successfully used by search engines like Google, as a ranking factor for content.

#### Example sentences :-

* A: This pasta is very tasty and affordable.
* B: This pasta is not tasty and is affordable.
* C: This pasta is very very delicious.

Let's consider each sentence as a document. Here also our first task is tokenization (dividing sentences into words or tokens) and then taking unique words.

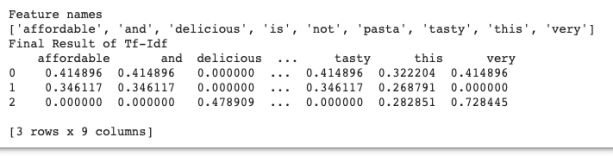
****

From the above table we can observe rarer words have more score than common words.That shows us the significance of the words in our corpus.

### Implementation of TF-IDF by using Sklearn



#### Output



# TF-IDF

TF-IDF is a abbreviation of Term Frequency — Inverse Document Frequency.

Term Frequency, defined as the number of times a term occurs in a document. The formula is shown below it is calculated as the division of number of time the term occur in a document and total number of terms in a document .

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

By this way we can take importance of a particular term in the document. However, when we deal with some stopwords like ‘a’,’an’,’the’ etc. are used frequently and due to this the importance of actual words is not discovered. To deal with these kind of words we use inbuild python libraries to remove stopwords. However, we have removed stop words but still we can not get importance of the word if it is used frequently in sentences. To solve this issue we check the next term Inverse Document Frequency.

Inverse Document Frequency is calculates with below formula. Where we divide total number of document with number of document with particular word and take log of this values. So, if the word is used frequently IDF value would move towards 0 else it will move to positive infinity.

**IDF(t) = log\_e(Total number of documents / Number of documents with term t in it + 1)**(\* +1 is taken in this formula to avoid denominator 0 value in this formula)

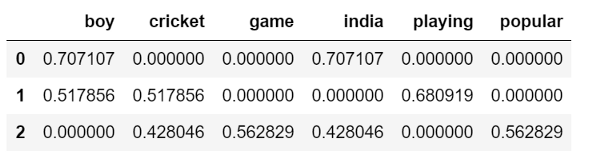
Finally, to get the TF-IDF representation of a word multiplication of TF an IDF is taken. Finally, we can get some values against important words in a sentences. if the word is used in all sentences by default it would be represented with 0 value. So, with this method we can get some sematic information of a word in a sentence.

*TF-IDF(t)=TF(t)\*IDF(t)*

For Example, Consider a document containing 100 words wherein the word *cat* appears 3 times. The term frequency (i.e., tf) for *cat* is then (3 / 100) = 0.03. Now, assume we have 10 million documents and the word *cat* appears in one thousand of these. Then, the inverse document frequency (i.e., idf) is calculated as log(10,000,000 / 1,000) = 4. Thus, the Tf-idf weight is the product of these quantities: 0.03 \* 4 = 0.12.

**Code Representation**

from sklearn.feature\_extraction.text import TfidfVectorizer,TfidfTransformer  
from nltk.tokenize import sent\_tokenize  
from nltk.tokenize import word\_tokenize  
from nltk.corpus import stopwords  
stopwords = set(stopwords.words(‘english’))  
text = [‘He is a boy and he is from India.’,’The boy is Playing The Cricket.’,’The Cricket is the most popular game in India.’]  
vectorizer = TfidfVectorizer(text,stop\_words=stopwords)  
vectors = vectorizer.fit\_transform(text)  
feature\_names = vectorizer.get\_feature\_names()  
dense = vectors.todense()  
denselist = dense.tolist()  
df = pd.DataFrame(denselist, columns=feature\_names)

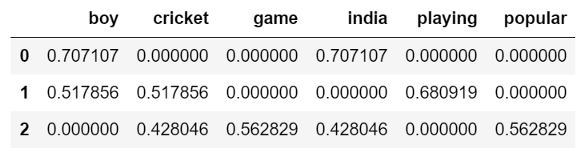


# Output of df

# We can see that the difference between normal countvectorizer and TF-IDF. For example, in second sentence value of playing is given 0.68 which is higher in this sentence that means it has some important information in this sentence. While, in countvectorizer playing is given a value 1 same as other words in this sentence so we would not have semantic information of this sentence.

# Here we imported TfidfTransformer as well it is same as TfidfVectorizer only difference is TfidfTransformer is used wot countvectorizer. Below is the code implementation of it.

from sklearn.feature\_extraction.text import TfidfVectorizer,TfidfTransformer  
from nltk.tokenize import sent\_tokenize  
from nltk.tokenize import word\_tokenize  
from nltk.corpus import stopwords  
stopwords = set(stopwords.words(‘english’))  
text = [‘He is a boy and he is from India.’,’The boy is Playing The Cricket.’,’The Cricket is the most popular game in India.’]  
tfIdfTransformer = TfidfTransformer()  
countVectorizer = CountVectorizer(text,stop\_words=stopwords)  
wordCount = countVectorizer.fit\_transform(text)  
newTfIdf = tfIdfTransformer.fit\_transform(wordCount)  
feature\_names = countVectorizer.get\_feature\_names()  
dense = newTfIdf.todense()  
denselist = dense.tolist()  
df = pd.DataFrame(denselist, columns=feature\_names)



## Word2vec

The Word2Vec model is used for learning vector representations of words called “word embeddings”. Did you observe that we didn’t get any semantic meaning from words of corpus by using previous methods? But for most of the applications of NLP tasks like sentiment classification, sarcasm detection etc require semantic meaning of a word and semantic relationships of a word with other words.

So can we get semantic meaning from words ?

Yeah exactly you got the answer , the answer is by using word2vec technique  we will get what we want.

Word embeddings have a capability of capturing semantic and syntactic relationships between words and also the context of words in a document. Word2vec is the technique to implement word embeddings.

Every word in a sentence is dependent on another word or other words.If you want to find similarities and relations between words ,we have to capture word dependencies.

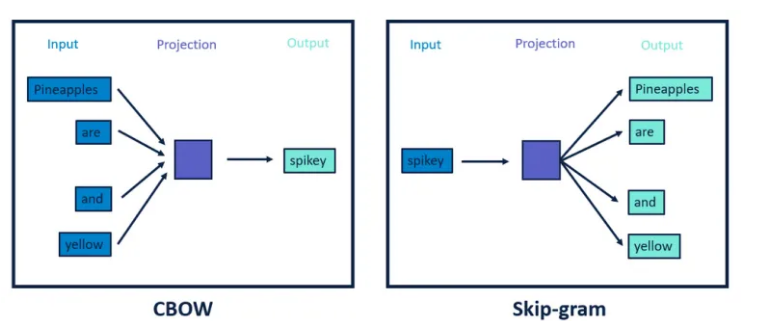
By using **Bag-of-words** and **TF-IDF** techniques we can not capture the meaning or relation of the words from vectors. Word2vec constructs such vectors called **embeddings**.

Word2vec model takes input as a **large size of corpus** and produces output to vector space. This vector space size may be in hundred of dimensionality. Each word vector will be placed on this vector space.

In vector space whatever words share context commonly in a corpus that are closer to each other. Word vector having positions of corresponding words in a vector space.

The Word2vec method learns all those types of relationships of words while building a model. For this purpose word2vec uses 2 types of methods. There are

1. Skip-gram
2. CBOW (Continuous Bag of Words)

****

Here one more thing we have to discuss that is window size. Did you remember the Bag-Of-words technique we discussed about 1-gram or uni-gram, bigram, trigram ….n-gram representation of text?

This method also follows the same technique. But here it is called **window size**.

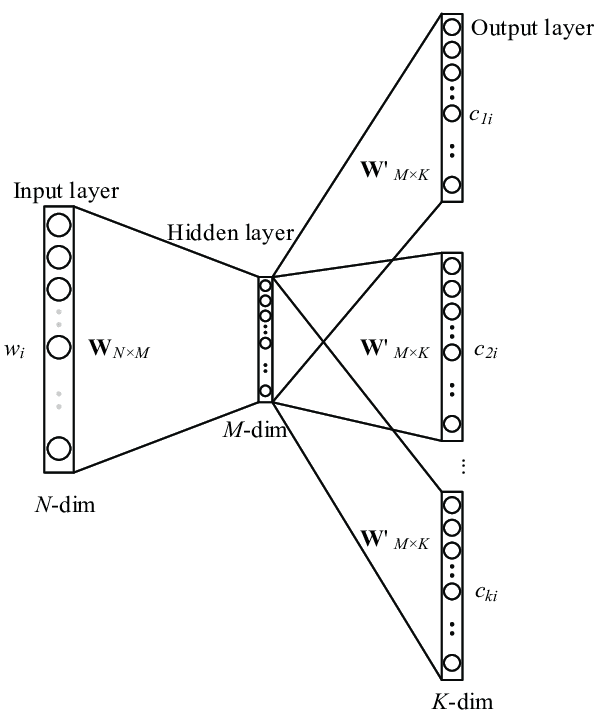
The Word2vec model will capture relationships of words with the help of window size by using skip-gram and CBow methods.

What is the difference between these 2 methods? Do you want to know?

That is a really simple technique. Before going to discuss these techniques, we have to know one more thing , why are we taking windows in this technique?  Just to know the center word and context of the center word.

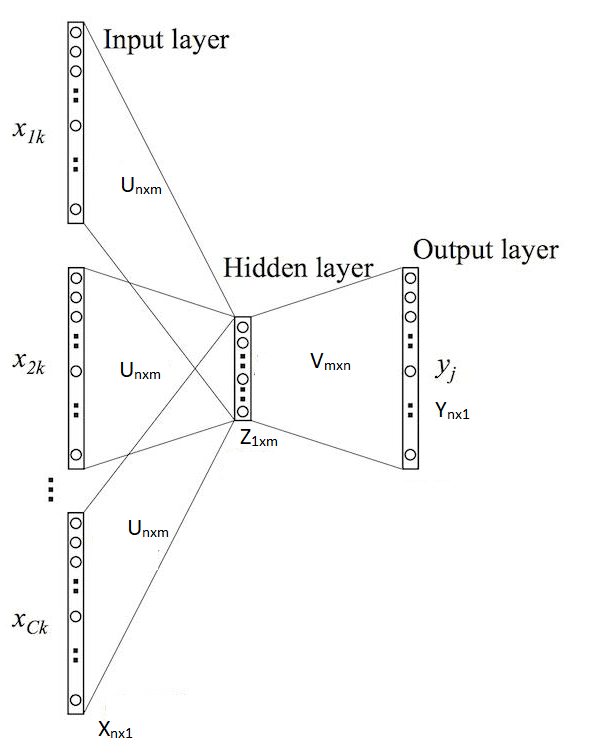
### Skip-Gram

In this method , take the center word from the **window size** words as an input and context words (neighbour words) as outputs. Word2vec models predict the context words of a center word using skip-gram method. Skip-gram works well with a small dataset and identifies rare words really well.



### Continuous Bag-of-words

### CBow is just a reverse method of the skip gram method. Here we are taking context words as input and predicting the center word within the window. Another difference from skip gram method is, It was working faster and better representations for most frequency words.



Difference between Skip gram & CBow

#### Skip gram:

* In this input is centre word and output is context words (neighbour words).
* Works well with small datasets.
* Skip-gram identifies rarer words better.

#### CBow:

* In this context or neighbor words are input and output is the center word.
* Works good with large datasets.
* Better representation for frequent words than rarer.

Context similarity

* The skip-gram model lets you keep information about the context of each word based on their proximity. In the example of “I ate an apple,” “ate” is a context word of “apple.” Context allows for grouping of words based on their syntactic and/or semantic similarity. If we were given additional input of “I ate a banana” and “I ate an orange,” we will soon find out that “ate” is also a context word of “banana” and “orange,” and therefore can infer that “apple,” “banana” and “orange” must share some commonality.
* The vectors of “apple,” “banana,” and “orange,” since they have similar context, are then adjusted to be closer to each other, forming a cluster on some multidimensional geometric space. On this note, linguist J.R. Firth said, “you shall know a word by the company it keeps.”
* Word2vec implementation

1. How to build  word2vec model with these two methods
2. Usage of Word embedding Pre-trained models
   1. Google word2vec
   2. Stanford glove Embeddings

#### Building our word2vec model with custom text

#### Word2vec with gensim

For this i am taking just a sample text file and will build a word2vec model by using the gensim python library.

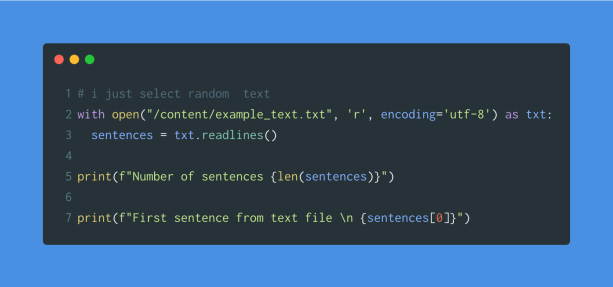
#### Require libraries

1. Gensim (**pip install --upgrade gensim**)
2. NLTK (**pip install nltk**)
3. Regex (**pip install re**)



We will get output like this

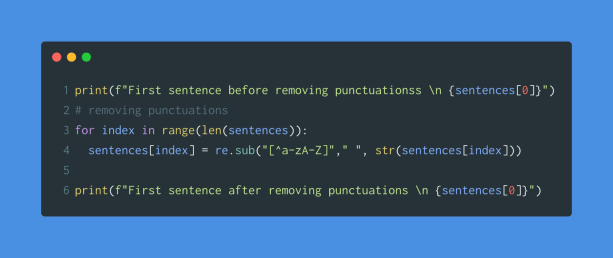
output



output

Now i am removing punctuations from all sentences. Because we can not get that much information from punctuations.But not all applications.

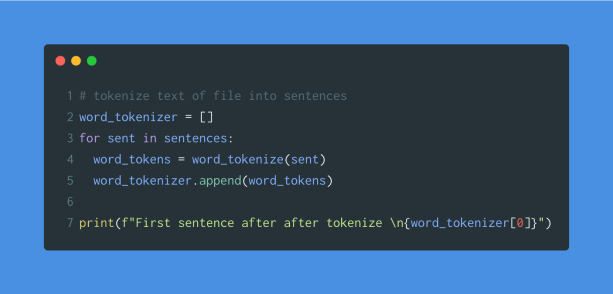
For this sample example we don’t need any punctuations , numbers, all these things so i will remove them with a regex pattern.



removing punctuations from sentences

output

Now we have to apply tokenization to all sentences.



#### Output

output

We can give these tokenized sentences to word2vec as input to the word2vec model.

#### Building word2vec with CBOW method



Output

*Total number of words*

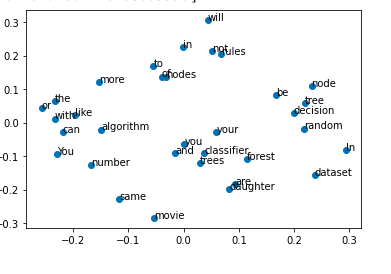
*79*

*array([-0.20608747,  0.05975117], dtype=float32)*

Word2vec model building is done.

So let’s see how it looks like by using **matplotlib** for visualization.





We can see in the above figure , node , tree, random, words are close to each other and also the distance between movie and algorithm. Maybe we can’t observe more words like this because of dataset size , if we use large dataset then we can  observe more clearly.

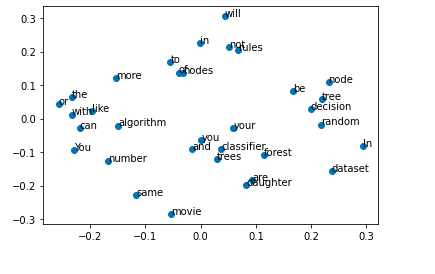
Building word2vec skip-gram method



output

Let’s see the visualization





Same as CBOW visualization graph here also same thing happens,  node , tree, random, words are close to each other and also the distance between movie and algorithm.

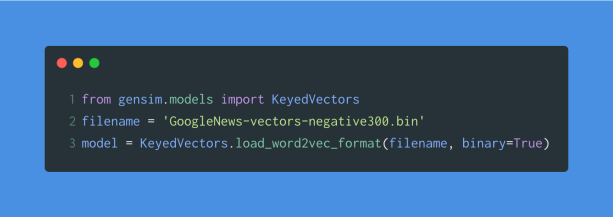
Word embedding model using Pre-trained models

If our  dataset size is small, then we can get too many words, and if we can't provide more sentences, the model will not learn more from our dataset. Otherwise if we want to build a word2vec model with a large corpus then it will require more resources like time,memory etc.

So how can we build a better word embedding model ? don’t worry , we can utilize already trained models. Here we are using 2 most popular pre-trained word embedding models. We  don't explain about these pre-trained models in detail, but tell how to use them.

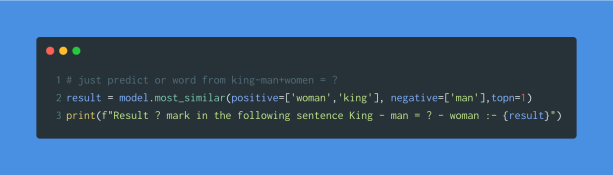
Google word2vec

We can download google word2vec pretrained model from  [link](https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21pQmM/edit?usp=sharing" \t "_blank).This is the compressed file so you have to extract that file before using it in the script.



We will see how word embeddings capture the relation between words with example of

King - man = ? - woman



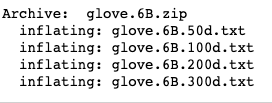
Output

output

### Stanford Glove Embeddings

Full form Glove is Global Vectors for Word Representation.

We can download this pretrained model from [this](http://nlp.stanford.edu/data/glove.6B.zip) link.This file also compressed one we have to extract , after extracting you can see different files. Glove embedding model provides different dimensions  of models like below



For this we have to do some pre-requested task.we have to convert the glove word embedding file to word2vec using **glove2word2vec()** function. From those file , i am taking 100 dimensions file **glove.6B.100d.txt**



output

## Word Embeddings

A word embedding is an approach to provide a dense vector representation of words that capture something about their meaning.

Word embeddings are an improvement over simpler bag-of-word model word encoding schemes like word counts and frequencies that result in large and sparse vectors (mostly 0 values) that describe documents but not the meaning of the words.

Word embeddings work by using an algorithm to train a set of fixed-length dense and continuous-valued vectors based on a large corpus of text. Each word is represented by a point in the embedding space and these points are learned and moved around based on the words that surround the target word.

It is defining a word by the company that it keeps that allows the word embedding to learn something about the meaning of words. The vector space representation of the words provides a projection where words with similar meanings are locally clustered within the space.

The use of word embeddings over other text representations is one of the key methods that has led to breakthrough performance with deep neural networks on problems like machine translation.

In this tutorial, we are going to look at how to use two different word embedding methods called word2vec by researchers at Google and GloVe by researchers at Stanford

## What are Word Embeddings?

* Word Embeddings are the texts converted into numbers and there may be different numerical representations of the same text. But before we dive into the details of Word Embeddings, the following question should be asked – Why do we need Word Embeddings?
* 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.
* Let us now define Word Embeddings formally. A Word Embedding format generally tries to map a word using a dictionary to a vector. Let us break this sentence down into finer details to have a clear view.
* Take a look at this example – **sentence**=” Word Embeddings are Word converted into numbers ”
* A *word*in this **sentence** may be “Embeddings” or “numbers ” etc.
* 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].
* This is just a very simple method to represent a word in the vector form. Let us look at different types of Word Embeddings or Word Vectors and their advantages and disadvantages over the rest.

## Different types of Word Embeddings

The different types of word embeddings can be broadly classified into two categories-

1. Frequency based Embedding
2. Prediction based Embedding

## Frequency based Embedding

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

1. Count Vector
2. TF-IDF Vector
3. Co-Occurrence Vector

#### 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 –

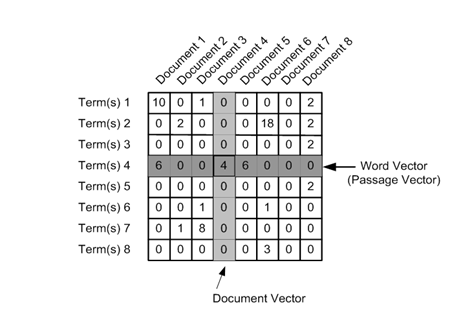
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | He | She | lazy | boy | Neeraj | person |
| D1 | 1 | 1 | 2 | 1 | 0 | 0 |
| D2 | 0 | 0 | 1 | 0 | 1 | 1 |

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.



# 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.

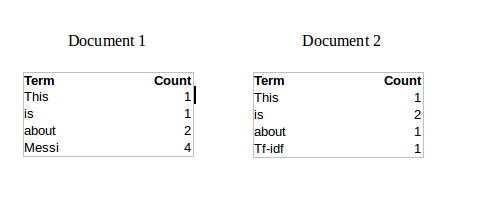
Common words like ‘is’, ‘the’, ‘a’ etc. tend to appear quite frequently in comparison to the words which are important to a document. For example, a document **A** on Lionel Messi is going to contain more occurences of the word “Messi” in comparison to other documents. But common words like “the” etc.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

# Co-Occurrence Matrix with a fixed context window

**The big idea** – Similar words tend to occur together and will have similar context for example – Apple is a fruit. Mango is a fruit.  
Apple and mango tend to have a similar context i.e fruit.

Before I dive into the details of how a co-occurrence matrix is constructed, there are two concepts that need to be clarified – Co-Occurrence and Context Window.

Co-occurrence – For a given corpus, the co-occurrence of a pair of words say w1 and w2 is the number of times they have appeared together in a Context Window.

Context Window – Context window is specified by a number and the direction. So what does a context window of 2 (around) means? Let us see an example below,

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Quick | Brown | Fox | Jump | Over | The | Lazy | Dog |

The green words are a 2 (around) context window for the word ‘Fox’ and for calculating the co-occurrence only these words will be counted. Let us see context window for the word ‘Over’.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Quick | Brown | Fox | Jump | Over | The | Lazy | Dog |

Now, let us take an example corpus to calculate a co-occurrence matrix.

Corpus = He is not lazy. He is intelligent. He is smart.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **He** | **is** | **not** | **lazy** | **intelligent** | **smart** |
| **He** | 0 | 4 | 2 | 1 | 2 | 1 |
| **is** | 4 | 0 | 1 | 2 | 2 | 1 |
| **not** | 2 | 1 | 0 | 1 | 0 | 0 |
| **lazy** | 1 | 2 | 1 | 0 | 0 | 0 |
| **intelligent** | 2 | 2 | 0 | 0 | 0 | 0 |
| **smart** | 1 | 1 | 0 | 0 | 0 | 0 |

Let us understand this co-occurrence matrix by seeing two examples in the table above. Red and the blue box.

Red box- It is the number of times ‘He’ and ‘is’ have appeared in the context window 2 and it can be seen that the count turns out to be 4. The below table will help you visualise the count.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| He | is | not | lazy | He | is | intelligent | He | is | smart |
|  |  |  |  |  |  |  |  |  |  |
| He | is | not | lazy | He | is | intelligent | He | is | smart |
|  |  |  |  |  |  |  |  |  |  |
| He | is | not | lazy | He | is | intelligent | He | is | smart |
|  |  |  |  |  |  |  |  |  |  |
| He | is | not | lazy | He | is | intelligent | He | is | smart |

while the word ‘lazy’ has never appeared with ‘intelligent’ in the context window and therefore has been assigned 0 in the blue box.

**Variations of Co-occurrence Matrix**

Let’s say there are V unique words in the corpus. So Vocabulary size = V. The columns of the Co-occurrence matrix form the context words. The different variations of Co-Occurrence Matrix are-

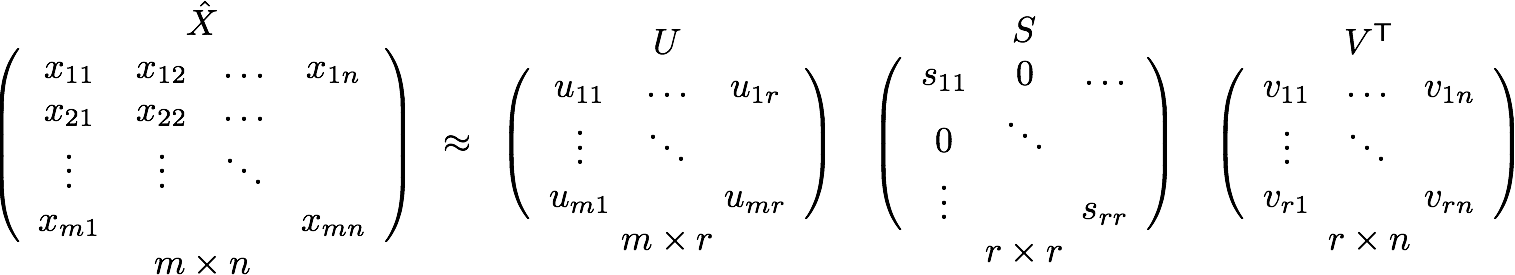
1. A co-occurrence matrix of size V X V. Now, for even a decent corpus V gets very large and difficult to handle. So generally, this architecture is never preferred in practice.
2. A co-occurrence matrix of size V X N where N is a subset of V and can be obtained by removing irrelevant words like stopwords etc. for example. This is still very large and presents computational difficulties.

But, remember this co-occurrence matrix is not the word vector representation that is generally used. Instead, this Co-occurrence matrix is decomposed using techniques like PCA, SVD etc. into factors and combination of these factors forms the word vector representation.

Let me illustrate this more clearly. For example, you perform PCA on the above matrix of size VXV. You will obtain V principal components. You can choose k components out of these V components. So, the new matrix will be of the form V X k.

And, a single word, instead of being represented in V dimensions will be represented in k dimensions while still capturing almost the same semantic meaning. k is generally of the order of hundreds.

So, what PCA does at the back is decompose Co-Occurrence matrix into three matrices, U,S and V where U and V are both orthogonal matrices. What is of importance is that dot product of U and S gives the word vector representation and V gives the word context representation.



**Advantages of Co-occurrence Matrix**

1. It preserves the semantic relationship between words. i.e man and woman tend to be closer than man and apple.
2. It uses SVD at its core, which produces more accurate word vector representations than existing methods.
3. It uses factorization which is a well-defined problem and can be efficiently solved.
4. It has to be computed once and can be used anytime once computed. In this sense, it is faster in comparison to others.

**Disadvantages of Co-Occurrence Matrix**

1. It requires huge memory to store the co-occurrence matrix.  
   But, this problem can be circumvented by factorizing the matrix out of the system for example in Hadoop clusters etc. and can be saved.

## Prediction based Vector

**Pre-requisite**: This section assumes that you have a working knowledge of how a neural network works and the mechanisms by which weights in an NN are updated. If you are new to Neural Network, I would suggest you go through [this awesome article](https://www.analyticsvidhya.com/blog/2017/05/neural-network-from-scratch-in-python-and-r/) by Sunil to gain a very good understanding of how NN works.

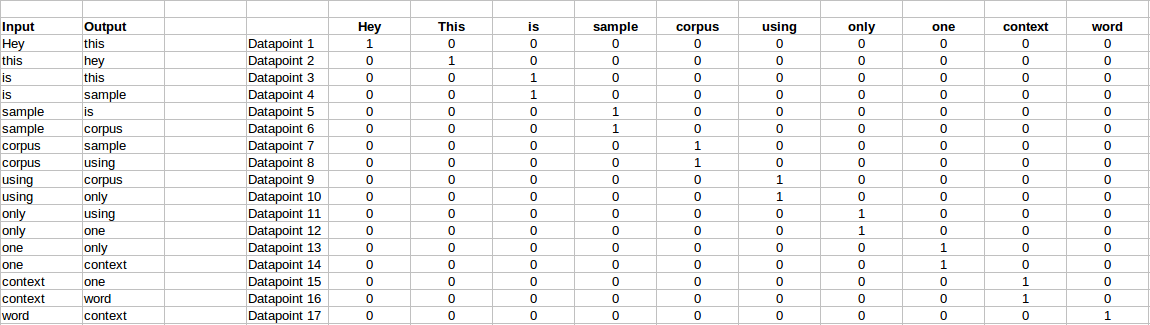
So far, we have seen deterministic methods to determine word vectors. But these methods proved to be limited in their word representations until Mitolov etc. el introduced word2vec to the NLP community. These methods were prediction based in the sense that they provided probabilities to the words and proved to be state of the art for tasks like word analogies and word similarities. They were also able to achieve tasks like King -man +woman = Queen, which was considered a result almost magical. So let us look at the word2vec model used as of today to generate word vectors.

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. Let us discuss both these methods separately and gain intuition into their working.

## CBOW (Continuous Bag of words)

The way CBOW work is that it tends to predict the probability of a word given a context. A context may be a single word or a group of words. But for simplicity, I will take a single context word and try to predict a single target word.

Suppose, we have a corpus C = “Hey, this is sample corpus using only one context word.” and we have defined a context window of 1. This corpus may be converted into a training set for a CBOW model as follow. The input is shown below. The matrix on the right in the below image contains the one-hot encoded from of the input on the left.

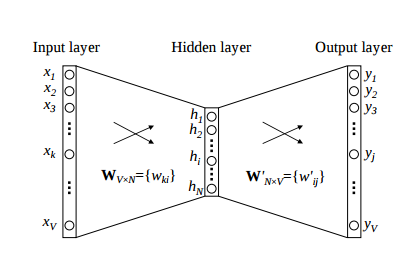


The target for a single datapoint say Datapoint 4 is shown as below

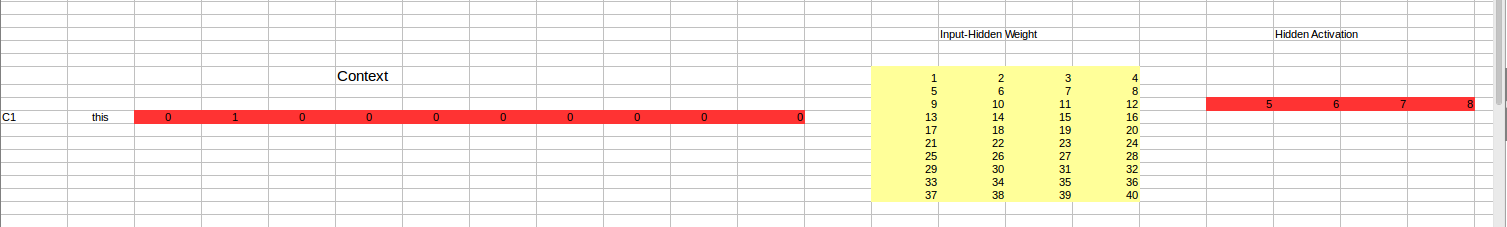
|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Hey | this | is | sample | corpus | using | only | one | context | word |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

This matrix shown in the above image is sent into a shallow neural network with three layers: an input layer, a hidden layer and an output layer. The output layer is a softmax layer which is used to sum the probabilities obtained in the output layer to 1. Now let us see how the forward propagation will work to calculate the hidden layer activation.

Let us first see a diagrammatic representation of the CBOW model.



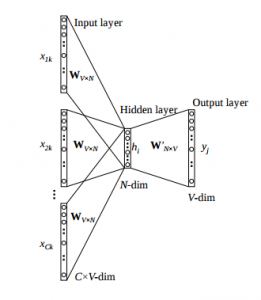
The matrix representation of the above image for a single data point is below.



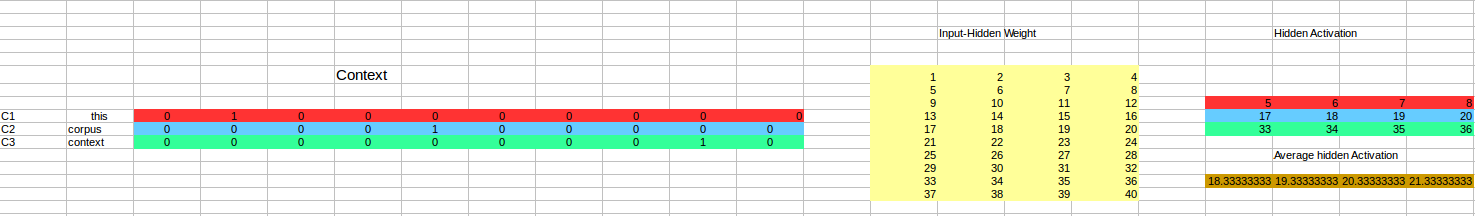
The flow is as follows:

1. The input layer and the target, both are one- hot encoded of size [1 X V]. Here V=10 in the above example.
2. There are two sets of weights. one is between the input and the hidden layer and second between hidden and output layer.  
   Input-Hidden layer matrix size =[V X N] , hidden-Output layer matrix  size =[N X V] : Where N is the number of dimensions we choose to represent our word in. It is arbitary and a hyper-parameter for a Neural Network. Also, N is the number of neurons in the hidden layer. Here, N=4.
3. There is a no activation function between any layers.( More specifically, I am referring to linear activation)
4. The input is multiplied by the input-hidden weights and called hidden activation. It is simply the corresponding row in the input-hidden matrix copied.
5. The hidden input gets multiplied by hidden- output weights and output is calculated.
6. Error between output and target is calculated and propagated back to re-adjust the weights.
7. The weight  between the hidden layer and the output layer is taken as the word vector representation of the word.

We saw the above steps for a single context word. Now, what about if we have multiple context words? The image below describes the architecture for multiple context words.



Below is a matrix representation of the above architecture for an easy understanding.



The image above takes 3 context words and predicts the probability of a target word. The input can be assumed as taking three one-hot encoded vectors in the input layer as shown above in red, blue and green.

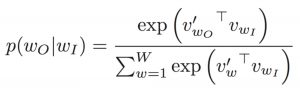
So, the input layer will have 3 [1 X V] Vectors in the input as shown above and 1 [1 X V] in the output layer. Rest of the architecture is same as for a 1-context CBOW.

The steps remain the same, only the calculation of hidden activation changes. Instead of just copying the corresponding rows of the input-hidden weight matrix to the hidden layer, an average is taken over all the corresponding rows of the matrix. We can understand this with the above figure. The average vector calculated becomes the hidden activation. So, if we have three context words for a single target word, we will have three initial hidden activations which are then averaged element-wise to obtain the final activation.

In both a single context word and multiple context word, I have shown the images till the calculation of the hidden activations since this is the part where CBOW differs from a simple MLP network. The steps after the calculation of hidden layer are same as that of the MLP as mentioned in this article – [Understanding and Coding Neural Networks from scratch](https://www.analyticsvidhya.com/blog/2017/05/neural-network-from-scratch-in-python-and-r/).

The differences between MLP and CBOW are  mentioned below for clarification:

1. The objective function in MLP is a MSE(mean square error) whereas in CBOW it is negative log likelihood of a word given a set of context i.e -log(p(wo/wi)), where p(wo/wi) is given as



wo : output word  
wi: context words

2. The gradient of error with respect to hidden-output weights and input-hidden weights are different since MLP has  sigmoid activations(generally) but CBOW has linear activations. The method however to calculate the gradient is same as an MLP.

**Advantages of CBOW:**

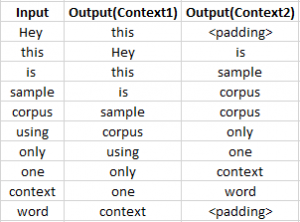
1. Being probabilistic is nature, it is supposed to perform superior to deterministic methods(generally).
2. It is low on memory. It does not need to have huge RAM requirements like that of co-occurrence matrix where it needs to store three huge matrices.

**Disadvantages of CBOW:**

1. CBOW takes the average of the context of a word (as seen above in calculation of hidden activation). For example, Apple can be both a fruit and a company but CBOW takes an average of both the contexts and places it in between a cluster for fruits and companies.
2. Training a CBOW from scratch can take forever if not properly optimized.

## Skip – Gram model

Skip – gram follows the same topology as of CBOW. It just flips CBOW’s architecture on its head. The aim of skip-gram is to predict the context given a word. Let us take the same corpus that we built our CBOW model on. C=”Hey, this is sample corpus using only one context word.” Let us construct the training data.

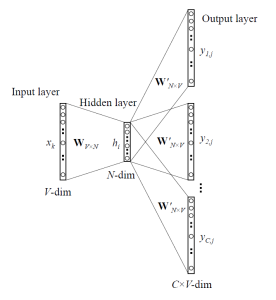


The input vector for skip-gram is going to be similar to a 1-context CBOW model. Also, the calculations up to hidden layer activations are going to be the same. The difference will be in the target variable. Since we have defined a context window of 1 on both the sides, there will be “**two” one hot encoded target variables** and “**two” corresponding outputs** as can be seen by the blue section in the image.

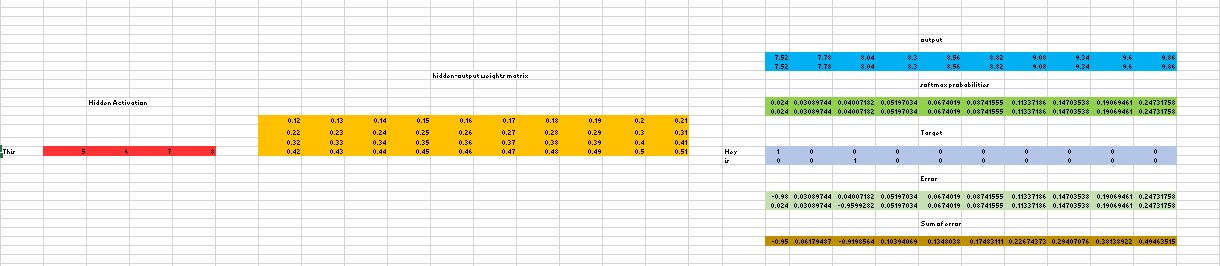
Two separate errors are calculated with respect to the two target variables and the two error vectors obtained are added element-wise to obtain a final error vector which is propagated back to update the weights.

The weights between the input and the hidden layer are taken as the word vector representation after training. The loss function or the objective is of the same type as of the CBOW model.

The skip-gram architecture is shown below.



For a better understanding, matrix style structure with calculation has been shown below.



Let us break down the above image.

Input layer  size – [1 X V], Input hidden weight matrix size – [V X N], Number of neurons in hidden layer – N, Hidden-Output weight matrix size – [N X V], Output layer size – C [1 X V]

In the above example, C is the number of context words=2, V= 10, N=4

1. The row in red is the hidden activation corresponding to the input one-hot encoded vector. It is basically the corresponding row of input-hidden matrix copied.
2. The yellow matrix is the weight between the hidden layer and the output layer.
3. The blue matrix is obtained by the matrix multiplication of hidden activation and the hidden output weights. There will be two rows calculated for two target(context) words.
4. Each row of the blue matrix is converted into its softmax probabilities individually as shown in the green box.
5. The grey matrix contains the one hot encoded vectors of the two context words(target).
6. Error is calculated by substracting the first row of the grey matrix(target) from the first row of the green matrix(output) element-wise. This is repeated for the next row. Therefore, for **n**target context words, we will have **n**error vectors.
7. Element-wise sum is taken over all the error vectors to obtain a final error vector.
8. This error vector is propagated back to update the weights.

### Advantages of Skip-Gram Model

1. Skip-gram model can capture two semantics for a single word. i.e it will have two vector representations of Apple. One for the company and other for the fruit.
2. Skip-gram with negative sub-sampling outperforms every other method generally.

# How does it work -Word2Vec

* The word2vec tool takes a text corpus as input and produces the word vectors as output. It first constructs a vocabulary from the training text data and then learns vector representation of words. The resulting word vector file can be used as features in many natural language processing and machine learning applications.
* A simple way to investigate the learned representations is to find the closest words for a user-specified word. The distance tool serves that purpose. For example, if you enter 'france', distance will display the most similar words and their distances to 'france', which should look like:

```

Word Cosine distance

spain 0.678515

belgium 0.665923

netherlands 0.652428

italy 0.633130

switzerland 0.622323

luxembourg 0.610033

portugal 0.577154

russia 0.571507

germany 0.563291

catalonia 0.534176

There are two main learning algorithms in word2vec : continuous bag-of-words and continuous skip-gram. The switch -cbow allows the user to pick one of these learning algorithms. Both algorithms learn the representation of a word that is useful for prediction of other words in the sentence

## Interesting properties of the word vectors

It was recently shown that the word vectors capture many linguistic regularities, for example vector operations vector('Paris') - vector('France') + vector('Italy') results in a vector that is very close to vector('Rome'), and vector('king') - vector('man') + vector('woman') is close to vector('queen') [3, 1].

To observe strong regularities in the word vector space, it is needed to train the models on large data set, with sufficient vector dimensionality as shown in [1]. Using the word2vec tool, it is possible to train models on huge data sets (up to hundreds of billions of words).

# From words to phrases and beyond

In certain applications, it is useful to have vector representation of larger pieces of text. For example, it is desirable to have only one vector for representing 'san francisco'. This can be achieved by pre-processing the training data set to form the phrases using the word2phrase tool,

## Word Cosine distance

los\_angeles 0.666175

golden\_gate 0.571522

oakland 0.557521

california 0.554623

san\_diego 0.534939

pasadena 0.519115

seattle 0.512098

taiko 0.507570

houston 0.499762

chicago\_illinois 0.491598

```

The linearity of the vector operations seems to weakly hold also for the addition of several vectors, so it is possible to add several word or phrase vectors to form representation of short sentences

## How to measure quality of the word vectors

Several factors influence the quality of the word vectors: \* amount and quality of the training data \* size of the vectors \* training algorithm

The quality of the vectors is crucial for any application. However, exploration of different hyper-parameter settings for complex tasks might be too time demanding. Thus, we designed simple test sets that can be used to quickly evaluate the word vector quality.

# Word clustering

The word vectors can be also used for deriving word classes from huge data sets. This is achieved by performing K-means clustering on top of the word vectors.

# Performance

The training speed can be significantly improved by using parallel training on multiple-CPU machine

The hyper-parameter choice is crucial for performance (both speed and accuracy), however varies for different applications. The main choices to make are:

* architecture: skip-gram (slower, better for infrequent words) vs CBOW (fast)
* the training algorithm: hierarchical softmax (better for infrequent words) vs negative sampling (better for frequent words, better with low dimensional vectors)
* sub-sampling of frequent words: can improve both accuracy and speed for large data sets (useful values are in range 1e-3 to 1e-5)
* dimensionality of the word vectors: usually more is better, but not always
* context (window) size: for skip-gram usually around 10, for CBOW around 5

## Pre-trained word and phrase vectors

We are publishing pre-trained vectors trained on part of Google News dataset (about 100 billion words). The model contains 300-dimensional vectors for 3 million words and phrases. The phrases were obtained using a simple data-driven approach described in [2]. The archive is available here: [GoogleNews-vectors-negative300.bin.gz](https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21pQmM/edit?usp=sharing).

An example output of ./distance GoogleNews-vectors-negative300.bin:

# Pre-trained entity vectors with Freebase naming

We are also offering more than 1.4M pre-trained entity vectors with naming from [Freebase](http://www.freebase.com/). This is especially helpful for projects related to knowledge mining.

* Entity vectors trained on 100B words from various news articles: [freebase-vectors-skipgram1000.bin.gz](https://docs.google.com/file/d/0B7XkCwpI5KDYaDBDQm1tZGNDRHc/edit?usp=sharing)
* Entity vectors trained on 100B words from various news articles, using the deprecated /en/ naming (more easily readable); the vectors are sorted by frequency: [freebase-vectors-skipgram1000-en.bin.gz](https://docs.google.com/file/d/0B7XkCwpI5KDYeFdmcVltWkhtbmM/edit?usp=sharing)

## Develop Word2Vec Embedding

* [Word2vec](https://en.wikipedia.org/wiki/Word2vec) is one algorithm for learning a word embedding from a text corpus.
* There are two main training algorithms that can be used to learn the embedding from text; they are continuous bag of words (CBOW) and skip grams.
* We will not get into the algorithms other than to say that they generally look at a window of words for each target word to provide context and in turn meaning for words. The approach was developed by [Tomas Mikolov](https://en.wikipedia.org/wiki/Word2vec), formerly at Google and currently at Facebook.
* Word2Vec models require a lot of text, e.g. the entire Wikipedia corpus. Nevertheless, we will demonstrate the principles using a small in-memory example of text.
* Gensim provides the [Word2Vec](https://radimrehurek.com/gensim/models/word2vec.html) class for working with a Word2Vec model.

## Learning a word embedding from text involves loading and organizing the text into sentences and providing them to the constructor of a new *Word2Vec()* instance

|  |  |
| --- | --- |
| 1  2 | sentences = ...  model = Word2Vec(sentences) |

* Specifically, each sentence must be tokenized, meaning divided into words and prepared (e.g. perhaps pre-filtered and perhaps converted to a preferred case).
* The sentences could be text loaded into memory, or an iterator that progressively loads text, required for very large text corpora.
* There are many parameters on this constructor; a few noteworthy arguments you may wish to configure are:
* size: (default 100) The number of dimensions of the embedding, e.g. the length of the dense vector to represent each token (word).
* window: (default 5) The maximum distance between a target word and words around the target word.
* min\_count: (default 5) The minimum count of words to consider when training the model; words with an occurrence less than this count will be ignored.
* workers: (default 3) The number of threads to use while training.
* sg: (default 0 or CBOW) The training algorithm, either CBOW (0) or skip gram (1).
* The defaults are often good enough when just getting started. If you have a lot of cores, as most modern computers do, I strongly encourage you to increase workers to match the number of cores (e.g. 8).
* After the model is trained, it is accessible via the “*wv*” attribute. This is the actual word vector model in which queries can be made.
* For example, you can print the learned vocabulary of tokens (words) as follows:
* words = list(model.wv.vocab)
* print(words)

## You can review the embedded vector for a specific token as follows:

## print(model['word'])

* Finally, a trained model can then be saved to file by calling the *save\_word2vec\_format()* function on the word vector model.
* By default, the model is saved in a binary format to save space
* model.wv.save\_word2vec\_format('model.bin')
* When getting started, you can save the learned model in ASCII format and review the contents.
* You can do this by setting *binary=False* when calling the *save\_word2vec\_format()* function, for example:

## model.wv.save\_word2vec\_format('model.txt', binary=False)

* The saved model can then be loaded again by calling the *Word2Vec.load()* function. For example:   
  model = Word2Vec.load('model.bin')

## Visualize Word Embedding

* After you learn word embedding for your text data, it can be nice to explore it with visualization.
* You can use classical projection methods to reduce the high-dimensional word vectors to two-dimensional plots and plot them on a graph.
* The visualizations can provide a qualitative diagnostic for your learned model.
* We can retrieve all of the vectors from a trained model as follows:

## X = model[model.wv.vocab]

## We can then train a projection method on the vectors, such as those methods offered in scikit-learn, then use matplotlib to plot the projection as a scatter plot. Plot it using PCA

# 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. These word embeddings come in handy during [hackathons](http://datahack.analyticsvidhya.com/?utm_source=blog&utm_medium=pretrained-word-embeddings-nlp) and of course, in real-world problems as well.

But why should we not learn our own embeddings? Well, 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. Let us discuss different pretrained word embeddings in the coming section.

What are the Different Pretrained Word Embeddings?

I would broadly 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

Let’s understand the working of Word2Vec and 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. That’s why:

Continous Bag Of Words and Skip-gram are inverses of each other

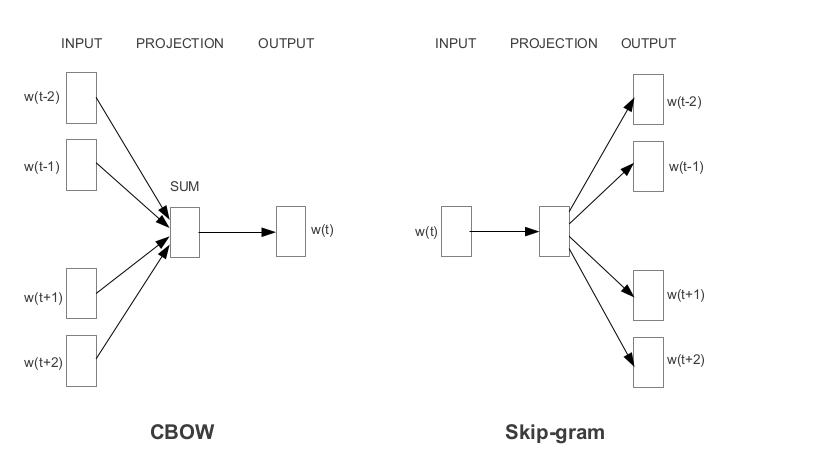
For example, consider the sentence: “I have failed at times but I never stopped trying”.  Let’s say we want to learn the embedding of the word “failed”. So, here the focus word is “failed”.

**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:

[](https://cdn.analyticsvidhya.com/wp-content/uploads/2020/03/Screenshot-from-2020-03-12-13-05-42.png)

### 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

But how can statistics represent meaning? Let me explain.

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

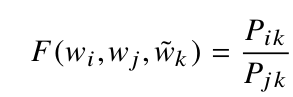
Next, let’s compute the ratio of probabilities:

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:



# Pre-trained Word Embeddings

Pre-trained models are the simplest way to start working with word embeddings. A pre-trained model is a set of word embeddings that have been created elsewhere that you simply load onto your computer and into memory.

The advantage of these models is that they can leverage massive datasets that you may not have access to, built using billions of different words, with a vast corpus of language that captures word meanings in a statistically robust manner. Example training data sets include the [entire corpus of wikipedia text](https://dumps.wikimedia.org/), [the common crawl dataset](http://commoncrawl.org/), or the [Google News Dataset](https://code.google.com/archive/p/word2vec/). Using a pre-trained model removes the need for you to spend time obtaining, cleaning, and processing (intensively) such large datasets

Pre-trained models are also available in languages other than English, opening up multi-lingual opportunities for your applications.

The disadvantage of pre-trained word embeddings is that the words contained within may not capture the peculiarities of language in your specific application domain. For example, Wikipedia may not have great word exposure to particular aspects of legal doctrine or religious text, so if your application is specific to a domain like this, your results may not be optimal due to the generality of the downloaded model’s word embeddings.

Pre-trained models in Spacy

Using pre-trained models in [Spacy](https://spacy.io/usage/spacy-101) is incredible convenient, given that they come built in. Simply [download the core English model using](https://spacy.io/usage/models):

# run this from a normal command line

python -m spacy download en\_core\_web\_md

## Spacy has a number of different [models](https://spacy.io/usage/models) of different sizes available for use, with models in 7 different languages (include English, Polish, German, Spanish, Portuguese, French, Italian, and Dutch), and of different sizes to suit your requirements. The code snippet above installs the larger-than-standard  [en\_core\_web\_md](https://spacy.io/models/en) library, which includes 20k unique vectors with 300 dimensions.

|  |  |
| --- | --- |
|  | **import** spacy  # Load the spacy model that you have installed  nlp = spacy.load('en\_core\_web\_md')  # process a sentence using the model  doc = nlp("This is some text that I am processing with Spacy")  # It's that simple - all of the vectors and words are assigned after this point  # Get the vector for 'text':  doc[3].vector  # Get the mean vector for the entire sentence (useful for sentence classification etc.)  doc.vector The vectors can be accessed directly using the .vector attribute of each processed token (word). The mean vector for the entire sentence is also calculated simply using .vector, providing a very convenient input for machine learning models based on sentences.Pre-trained models in Gensim [Gensim](https://spacy.io/usage/spacy-101) doesn’t come with the same in built models as Spacy, so to load a pre-trained model into Gensim, you first need to find and download one. A popular pre-trained option is the Google News dataset model, containing 300-dimensional embeddings for 3 millions words and phrases. Download the binary file ‘GoogleNews-vectors-negative300.bin’ (1.3 GB compressed) from <https://code.google.com/archive/p/word2vec/>. **from** gensim.models **import** KeyedVectors  # Load vectors directly from the file  model = KeyedVectors.load\_word2vec\_format('data/GoogleGoogleNews-vectors-negative300.bin', binary=**True**)  # Access vectors for specific words with a keyed lookup:  vector = model['easy']  # see the shape of the vector (300,)  vector.shape  # Processing sentences is not as simple as with Spacy:  vectors = [model[x] **for** x **in** "This is some text I am processing with Spacy".split(' ')] Gensim includes functions to explore the vectors loaded, examine word similarity, and to find synonyms in of words using ‘similar’ vectors: |

## Load Google’s Word2Vec Embedding

* Training your own word vectors may be the best approach for a given NLP problem.
* But it can take a long time, a fast computer with a lot of RAM and disk space, and perhaps some expertise in finessing the input data and training algorithm.
* An alternative is to simply use an existing pre-trained word embedding.
* Along with the paper and code for word2vec, Google also published a pre-trained word2vec model on the [Word2Vec Google Code Project](https://code.google.com/archive/p/word2vec/).
* A pre-trained model is nothing more than a file containing tokens and their associated word vectors. The pre-trained Google word2vec model was trained on Google news data (about 100 billion words); it contains 3 million words and phrases and was fit using 300-dimensional word vectors.
* It is a 1.53 Gigabytes file. You can download it from here:
* [GoogleNews-vectors-negative300.bin.gz](https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21pQmM/edit?usp=sharing)

Unzipped, the binary file (GoogleNews-vectors-negative300.bin) is 3.4 Gigabytes.

* The Gensim library provides tools to load this file. Specifically, you can call the *KeyedVectors.load\_word2vec\_format()* function to load this model into memory, for example:
* from gensim.models import KeyedVectors
* filename = 'GoogleNews-vectors-negative300.bin'
* model = KeyedVectors.load\_word2vec\_format(filename, binary=True)
* On my modern workstation, it takes about 43 seconds to load.
* Another interesting thing that you can do is do a little linear algebra arithmetic with words.
* For example, a popular example described in lectures and introduction papers is:

|  |
| --- |
| queen = (king - man) + woman |

* That is the word queen is the closest word given the subtraction of the notion of man from king and adding the word woman. The “man-ness” in king is replaced with “woman-ness” to give us queen. A very cool concept.
* Gensim provides an interface for performing these types of operations in the most\_similar() function on the trained or loaded model.
* result = model.most\_similar(positive=['woman', 'king'], negative=['man'], topn=1)
* print(result)

## Load Stanford’s GloVe Embedding

* Stanford researchers also have their own word embedding algorithm like word2vec called [Global Vectors for Word Representation](https://nlp.stanford.edu/projects/glove/), or GloVe for short.
* I won’t get into the details of the differences between word2vec and GloVe here, but generally, NLP practitioners seem to prefer GloVe at the moment based on results.
* Like word2vec, the GloVe researchers also provide pre-trained word vectors, in this case, a great selection to choose from.
* You can download the GloVe pre-trained word vectors and load them easily with gensim.
* The first step is to convert the GloVe file format to the word2vec file format. The only difference is the addition of a small header line. This can be done by calling the *glove2word2vec()* function. For example:

from gensim.scripts.glove2word2vec import glove2word2vec

glove\_input\_file = 'glove.txt'

word2vec\_output\_file = 'word2vec.txt'

glove2word2vec(glove\_input\_file, word2vec\_output\_file)

Once converted, the file can be loaded just like word2vec file above.

Let’s make this concrete with an example.

You can download the smallest GloVe pre-trained model from the [GloVe website](https://nlp.stanford.edu/projects/glove/). It an 822 Megabyte zip file with 4 different models (50, 100, 200 and 300-dimensional vectors) trained on Wikipedia data with 6 billion tokens and a 400,000 word vocabulary.

The direct download link is here:

* [glove.6B.zip](http://nlp.stanford.edu/data/glove.6B.zip)

Working with the 100-dimensional version of the model, we can convert the file to word2vec format as follows:

|  |  |
| --- | --- |
|  | from gensim.scripts.glove2word2vec import glove2word2vec  glove\_input\_file = 'glove.6B.100d.txt'  word2vec\_output\_file = 'glove.6B.100d.txt.word2vec'  glove2word2vec(glove\_input\_file, word2vec\_output\_file) |

## You now have a copy of the GloVe model in word2vec format with the filename glove.6B.100d.txt.word2vec.

## Create Custom Word Embeddings

Training your own word embeddings need not be daunting, and, for specific problem domains, will lead to enhanced performance over pre-trained models. The Gensim library provides a [simple API](https://radimrehurek.com/gensim/models/word2vec.html) to the [Google word2ve](https://code.google.com/archive/p/word2vec/)c algorithm which is a go-to algorithm for beginners.

To train your own model, the main challenge is getting access to a training data set. Computation is not massively onerous – you’ll manage to process a large model on a powerful laptop in hours rather than days.

**Phrase Detection using Gensim Phraser**

* Commonly occurring multiword expressions (bigrams / trigrams) in text carry different meaning to the words occurring singularly. For example, the words ‘new’ and ‘York’ expressed singularly are inherently different to the utterance ‘New York’. Detecting frequently co-occuring words and combining them can enhance word vector accuracy.
* A ‘[Phraser](https://radimrehurek.com/gensim/models/phrases.html)‘ from [Gensim](https://radimrehurek.com/gensim/) can detect frequently occurring bigrams easily, and apply a transform to data to create pairs, i.e. ‘New York’ -> ‘New\_York’. Pre-processing text input to account for such bigrams can improve the accuracy and usefulness of the resulting word vectors. Ultimately, instead of training vectors for ‘new’ and ‘york’ separately, a new vector for ‘New\_York’ is created.

The [gensim.models.phrases](https://radimrehurek.com/gensim/models/phrases.html) module provides everything required in a simple form:

# Phrase Detection

# Give some common terms that can be ignored in phrase detection

# For example, 'state\_of\_affairs' will be detected because 'of' is provided here:

common\_terms = ["of", "with", "without", "and", "or", "the", "a"]

# Create the relevant phrases from the list of sentences:

phrases = Phrases(all\_sentences, common\_terms=common\_terms)

# The Phraser object is used from now on to transform sentences

bigram = Phraser(phrases)

# Applying the Phraser to transform our sentences is simply

all\_sentences = list(bigram[all\_sentences])

[](https://shanelynnwebsite-mid9n9g1q9y8tt.netdna-ssl.com/wp-content/uploads/2018/03/Phraser-application-example-gensim.png)

## In addition to Word2Vec, Gensim also includes algorithms for [fasttext](https://radimrehurek.com/gensim/models/fasttext.html" \l "module-gensim.models.fasttext), [VarEmbed](https://github.com/rguthrie3/MorphologicalPriorsForWordEmbeddings), and [WordRank](https://radimrehurek.com/gensim/models/wrappers/wordrank.html) ([original](https://bitbucket.org/shihaoji/wordrank/)) also.

## Visualize Word Embedding

**Applications**

The word embeddings are widely used in the following tasks:

* Machine Translation
* Sentiment Analysis
* Document Classification/ Clustering
* Topic Modelling
* Automatic Speech Recognition
* Document Similarities
* Natural Language Generation
* Natural Language Understanding

<https://www.analyticsvidhya.com/blog/2020/08/top-4-sentence-embedding-techniques-using-python/>

<https://machinelearningmastery.com/develop-word-embeddings-python-gensim/>

<https://code.google.com/archive/p/word2vec/>

<https://nlp.stanford.edu/projects/glove/>

<https://www.shanelynn.ie/word-embeddings-in-python-with-spacy-and-gensim/>

https://towardsdatascience.com/creating-word-embeddings-coding-the-word2vec-algorithm-in-python-using-deep-learning-b337d0ba17a8

https://dataaspirant.com/word-embedding-techniques-nlp/