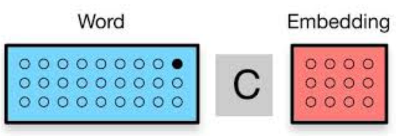
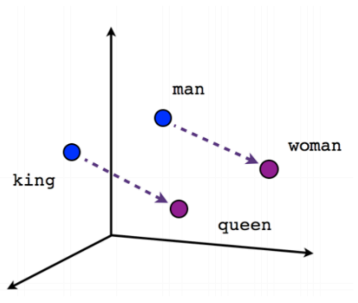
```
# IMPORTANT: RUN THIS CELL IN ORDER TO IMPORT YOUR KAGGLE DATA SOURCES
# TO THE CORRECT LOCATION (/kaggle/input) IN YOUR NOTEBOOK,
# THEN FEEL FREE TO DELETE THIS CELL.
# NOTE: THIS NOTEBOOK ENVIRONMENT DIFFERS FROM KAGGLE'S PYTHON
# ENVIRONMENT SO THERE MAY BE MISSING LIBRARIES USED BY YOUR
# NOTEBOOK.

import os
import sys
from tempfile import NamedTemporaryFile
from urllib.request import urlopen
from urllib.parse import unquote, urlparse
from urllib.error import HTTPError
from zipfile import ZipFile
import tarfile
import shutil

CHUNK_SIZE = 40960
DATA_SOURCE_MAPPING = 'word2vec-google:https%3A%2F%2Fstorage.googleapis.com%2Fkaggle-data-sets%2F7901%2F11147%2Fbundle%2Farchive.zip%3FX-Goog-Algori

KAGGLE_INPUT_PATH='/kaggle/input'
KAGGLE_WORKING_PATH='/kaggle/working'
KAGGLE_SYMLINK='kaggle'

!umount /kaggle/input/ 2> /dev/null
shutil.rmtree('/kaggle/input', ignore_errors=True)
os.makedirs(KAGGLE_INPUT_PATH, 0o777, exist_ok=True)
os.makedirs(KAGGLE_WORKING_PATH, 0o777, exist_ok=True)

try:
  os.symlink(KAGGLE_INPUT_PATH, os.path.join("..", 'input'), target_is_directory=True)
except FileExistsError:
  pass
try:
  os.symlink(KAGGLE_WORKING_PATH, os.path.join("..", 'working'), target_is_directory=True)
except FileExistsError:
  pass

for data_source_mapping in DATA_SOURCE_MAPPING.split(','):
    directory, download_url_encoded = data_source_mapping.split(':')
    download_url = unquote(download_url_encoded)
    filename = urlparse(download_url).path
    destination_path = os.path.join(KAGGLE_INPUT_PATH, directory)
    try:
        with urlopen(download_url) as fileres, NamedTemporaryFile() as tfile:
            total_length = fileres.headers['content-length']
            print(f'Downloading {directory}, {total_length} bytes compressed')
            dl = 0
            data = fileres.read(CHUNK_SIZE)
            while len(data) > 0:
                dl += len(data)
                tfile.write(data)
                done = int(50 * dl / int(total_length))
                sys.stdout.write(f"\r[{'=' * done}{' ' * (50-done)}] {dl} bytes downloaded")
                sys.stdout.flush()
                data = fileres.read(CHUNK_SIZE)
            if filename.endswith('.zip'):
              with ZipFile(tfile) as zfile:
                zfile.extractall(destination_path)
            else:
              with tarfile.open(tfile.name) as tarfile:
                tarfile.extractall(destination_path)
            print(f'\nDownloaded and uncompressed: {directory}')
    except HTTPError as e:
        print(f'Failed to load (likely expired) {download_url} to path {destination_path}')
        continue
    except OSError as e:
        print(f'Failed to load {download_url} to path {destination_path}')
        continue

print('Data source import complete.')
```

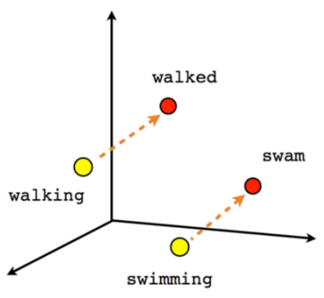
Word embedding with Python

word2vec, doc2vec, GloVe implementation with Python

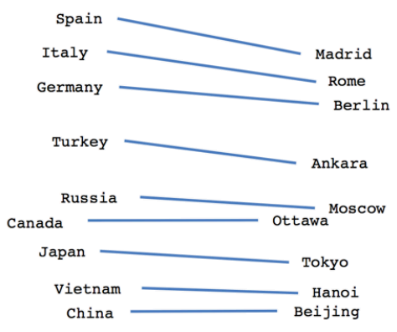




Male-Female



Verb tense



Country-Capital

Table of Contents

1.What are Word Embeddings?

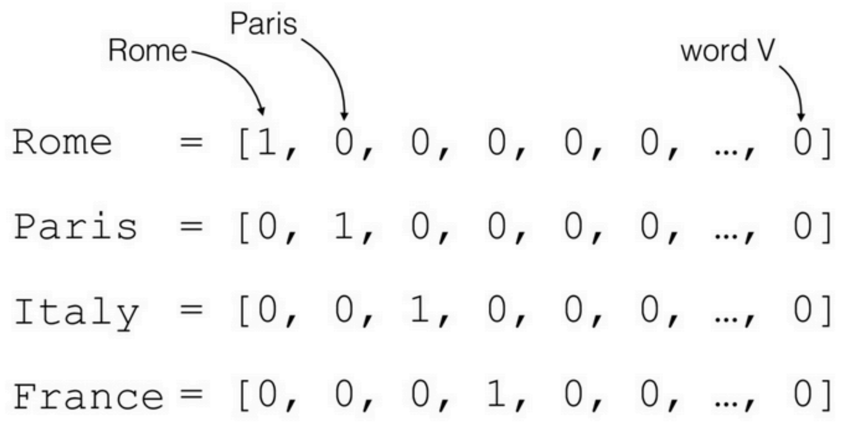
Defination

> **Word embeddings are a type of word representation that allows words with similar meaning to have a similar representation.** ***...By Jason Brownlee.***

Example 1

$$X = \begin{array}{c|cccccccc} & I & like & enjoy & deep & learning & NLP & flying & . \\ \hline I & 0 & 2 & 1 & 0 & 0 & 0 & 0 & 0 \\ like & 2 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ enjoy & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ deep & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ learning & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ NLP & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ flying & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ . & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{array}$$

Example 2



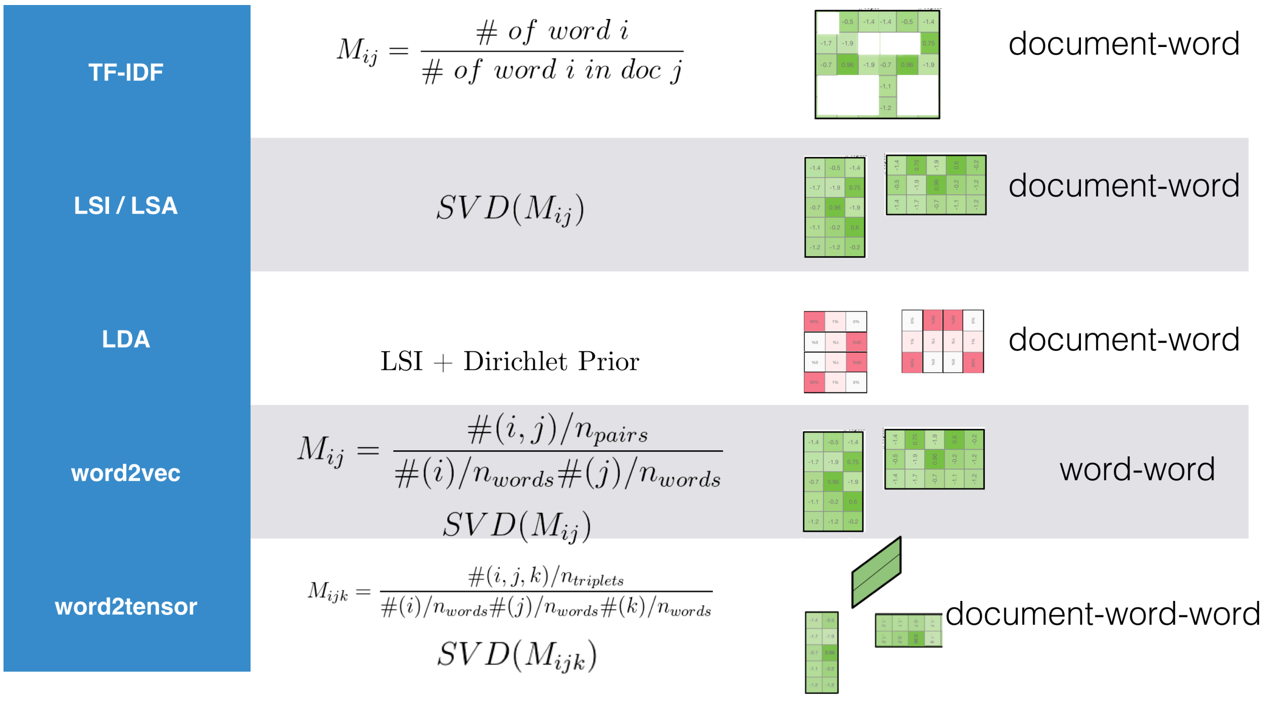
```
Rome   = [1, 0, 0, 0, 0, 0, …, 0]
Paris  = [0, 1, 0, 0, 0, 0, …, 0]
Italy  = [0, 0, 1, 0, 0, 0, …, 0]
France = [0, 0, 0, 1, 0, 0, …, 0]
```

- A very basic definition of a word embedding is a real number, vector representation of a word. Typically, these days, words with similar meaning will have vector representations that are close together in the embedding space (though this hasn't always been the case).
- ***Word embedding is a dense representation of words in the form of numeric vectors. It can be learned using a variety of language models. The word embedding representation is able to reveal many hidden relationships between words. For example, vector("cat") - vector("kitten") is similar to vector("dog") - vector("puppy"). This post introduces several models for learning word embedding and how their loss functions are designed for the purpose.***

2.Different types of Word Embedding

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

1. **Frequency based Embedding**
2. **Prediction based Embedding**



TF-IDF	$M_{ij} = \frac{\#\ of\ word\ i}{\#\ of\ word\ i\ in\ doc\ j}$	document-word
LSI / LSA	$SVD(M_{ij})$	document-word
LDA	LSI + Dirichlet Prior	document-word
word2vec	$M_{ij} = \frac{\#(i,j)/n_{pairs}}{\#(i)/n_{words}\#(j)/n_{words}}$ $SVD(M_{ij})$	word-word
word2tensor	$M_{ijk} = \frac{\#(i,j,k)/n_{triplets}}{\#(i)/n_{words}\#(j)/n_{words}\#(k)/n_{words}}$ $SVD(M_{ijk})$	document-word-word

2.1.Frequency based Embedding

2.1.1.Count Vectors

- Extract the corpus C {d1, d2 ... dD} of the document D and the N unique tokens (words) from the corpus C. N unque form our dictionary and the size of the count vector matrix M by DX N. D (i) is the number of times each row of the matrix contains M tokens in the document.

Let us understand this with a simple example.

- **D1: He is a lazy boy. She is also lazy.**
- **D2: Neeraj is a lazy person.**

The dictionary created can be a word with a **unique tag 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

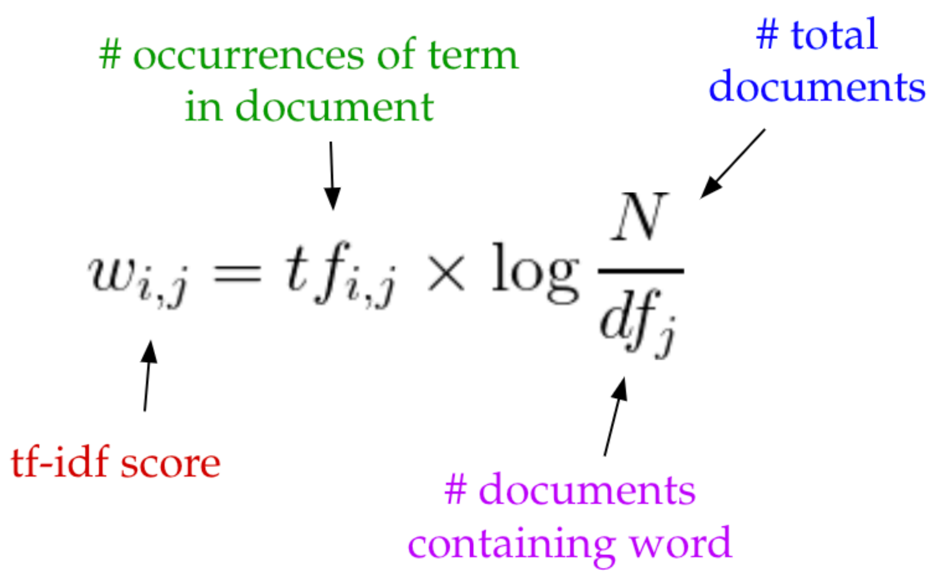
Practical Example

1. Count Vectorization

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
import pandas as pd


text = ['The quick brown fox jumped over the lazy dog']


vectorizer = CountVectorizer()
vectorizer.fit(text)
print(vectorizer.vocabulary_)
#encode the document
vector = vectorizer.transform(text)
print(vector.shape)
print(type(vector))
print(vector.toarray())


vector = vectorizer.transform(text)
print(vector.shape)
print(type(vector))
print(vector.toarray())


vector.toarray()
df = pd.DataFrame(vector.todense())
df.describe()
```

2.1.2.TF-IDF

Formula

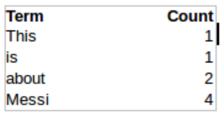


# occurrences of term in document
# total documents
$w_{i,j} = tf_{i,j} \times \log \frac{N}{df_j}$
tf-idf score
# documents containing word

1. **TF Score (Term Frequency)** : Considers documents as bag of words, agnostic to order of words. A document with 10 occurrences of the term is more relevant than a document with term frequency 1. But it is not 10 times more relevant, relevance is not proportional to frequency
2. **IDF Score (Inverse Document Frequency)** We also want to use the frequency of the term in the collection for weighting and ranking. Rare terms are more informative than frequent terms. We want low positive weights for frequent terms and high weights for rare terms.

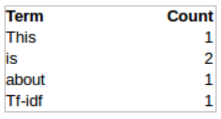
Mathematical Example

Document 1



Term	Count
This	1
is	1
about	2
Messi	4

Document 2



Term	Count
This	1
is	2
about	1
Tf-idf	1

Term Frequency

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

- **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.**

Inverse Document Frequency

IDF(***Inverse Document Frequency***) = **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.**
- **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

```
text_2 = ['The quick brown fox jumped over the lazy dog','The dog','The fox']
```

```
#create the transform
vectorizer_2 = TfidfVectorizer()
#tokenize and build vocab
vectorizer_2.fit(text_2)
# summarize
print(vectorizer_2.vocabulary_)
print(vectorizer_2.idf_)
#encode document
vector_2 = vectorizer_2.transform(text_2)
#summarize encode vector
print(vector_2.shape)
print(vector_2.toarray())
```

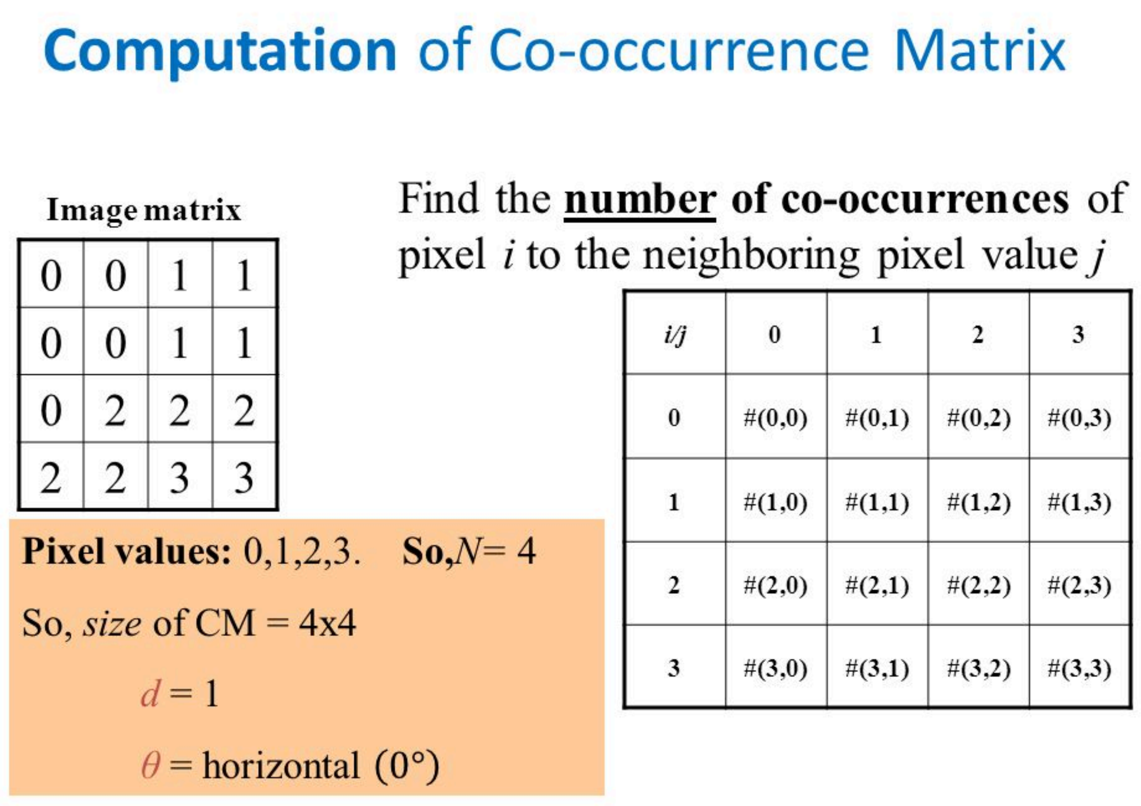
```
vector_2.shape
```

```
vector_2.toarray()
pd.DataFrame(vector_2.todense())
```

```
df.info
```

```
# trying to get cosine similarity
cosine_similarity(vector,vector_2)
```

2.1.3.Co-Occurrence Matrix



Computation of Co-occurrence Matrix

Image matrix

0	0	1	1
0	0	1	1
0	2	2	2
2	2	3	3

Pixel values: 0,1,2,3. **So,** $N = 4$

So, *size* of CM = 4x4

$d = 1$

θ = horizontal (0°)

Find the **number of co-occurrences** of pixel i to the neighboring pixel value j

i/j	0	1	2	3
0	#(0,0)	#(0,1)	#(0,2)	#(0,3)
1	#(1,0)	#(1,1)	#(1,2)	#(1,3)
2	#(2,0)	#(2,1)	#(2,2)	#(2,3)
3	#(3,0)	#(3,1)	#(3,2)	#(3,3)

The big idea: similar words tend to happen together and will have a similar context, for example, Apple is a fruit. Mango is a fruit.Apples and mangos tend to have a similar background, namely fruits. Before delving deeper into the details of **constructing a co-occurrence matrix,** two concepts need to be clarified: **co-occurrence and context limitations.**

- **Co-occurrence:** for a given corpus, ***the symbiosis of a pair of words w1 and w2 is the number of times they appear together within the context boundary.***
- **Context limits:** Context limits are specified by ***numbers and addresses. So, what does the context limit 2 (around) mean? Let's see an example.***

The **green word is the context boundary 2 (surrounding) of the word "Fox"** and only these words are calculated to calculate the co-occurrence. Let's look at the context limit of the word "Over".

Let's take an example to calculate a **co-occurrence matrix.**

- **Corpus = He is not lazy. He is intelligent. He is intelligent.**

Let us understand this co-occurrence matrix by using the two examples from the previous table. **Red and blue boxes.**

Red box: The number of occurrences of "**He**" and "**East**" within context 2, you can see that this number is 4.

- The word "**Lazy**" has never been "**intelligent**" in the ***context of the boundary, so it has been assigned a value of 0 in the blue box.***

Change of co-occurrence matrix.

- Suppose there are *V unique words in the corpus. Therefore, the size of the vocabulary = V.* The columns of the ***concurrency matrix form a context word. The varied changes in the co-occurrence matrix are:*
  1. ***V X V size co-occurrence matrix Now, a regular V body becomes very large, which will be difficult to handle.*** In general, this framework is not the first application in practice.
  2. A ***co-occurrence matrix of size V x N, where N is a subset of V and can be obtained***, for example, by removing irrelevant words, such as invalid words, which are still very large and present. Computational difficulties

However, keep in mind that this co-occurrence matrix is not generally used for the vector representation of words, but is **divided into factors that use techniques such as PCA, SVD, etc. These factors form a representation of the word vector.**

For example, perform a PCA in a full-size VXV array. You will get the main components of V. You can select k components of these V. V X components.

In addition, **a word will be represented in k-dimensional form instead of v-dimensional while rigorously capturing identical semantic information. K is generally of the order of several hundred.**

Next, ***PCA will do this to divide the co-occurrence matrix into three matrices U, S, and V, where U and V are orthogonal matrices. What is important is that the scalar product of U and S gives a representation of the word vector, and V gives a representation of the word context.***

$$\underset{m \times n}{\overset{\hat{X}}{\begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & \\ \vdots & \vdots & \ddots & \\ x_{m1} & & & x_{mn} \end{pmatrix}}} \approx \underset{m \times r}{\overset{U}{\begin{pmatrix} u_{11} & \cdots & u_{1r} \\ \vdots & \ddots & \\ u_{m1} & & u_{mr} \end{pmatrix}}} \underset{r \times r}{\overset{S}{\begin{pmatrix} s_{11} & 0 & \cdots \\ 0 & \ddots & \\ \vdots & & s_{rr} \end{pmatrix}}} \underset{r \times n}{\overset{V^{\top}}{\begin{pmatrix} v_{11} & \cdots & v_{1n} \\ \vdots & \ddots & \\ v_{r1} & & v_{rn} \end{pmatrix}}}$$

Advantages of Co-occurrence Matrix:

- It preserves the semantic relationship between words. i.e man and woman tend to be closer than man and apple.
- It uses SVD at its core, which produces more accurate word vector representations than existing methods.
- It uses factorization which is a well-defined problem and can be efficiently solved.
- 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

- 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.

Practical Example

```
# libraries we'll need
# https://www.kaggle.com/rtatman/co-occurrence-matrix-plot-in-python
import pandas as pd # dataframes
from io import StringIO # string to data frame
import seaborn as sns # plotting


# read in our data & convert to a data frame
data_tsv = StringIO("""city    province    position
0   Massena     NY  jr
1   Maysville   KY  pm
2   Massena     NY  m
3   Athens      OH  jr
4   Hamilton    OH  sr
5   Englewood   OH  jr
6   Saluda      SC  sr
7   Batesburg   SC  pm
8   Paragould   AR  m""")

my_data_frame = pd.read_csv(data_tsv, delimiter=r"\s+")


my_data_frame


# conver to co-occurance matrix
co_mat = pd.crosstab(my_data_frame.province, my_data_frame.position)
co_mat


# plot heat map of co-occuance matrix
sns.heatmap(co_mat)
```

2.2.Prediction based Embedding

- **Word2vec** is not a single algorithm, but a **combination of two technologies - CBOW (continuous word bag) and Skip-gram model.** Both of these are shallow neural networks, which also map words to a target variable. Both techniques learn the weights represented by word vectors.

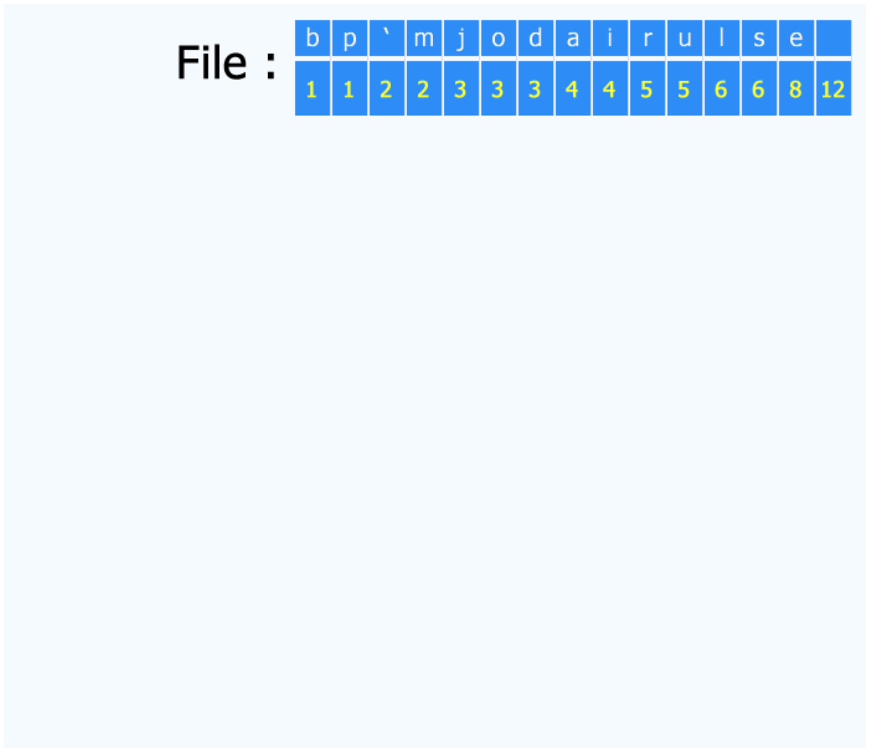
2.2.1.CBOW

(Continuous Bag of words)

- **CBOW model and the skip-gram model are based on the Huffman tree. *

Huffman Tree

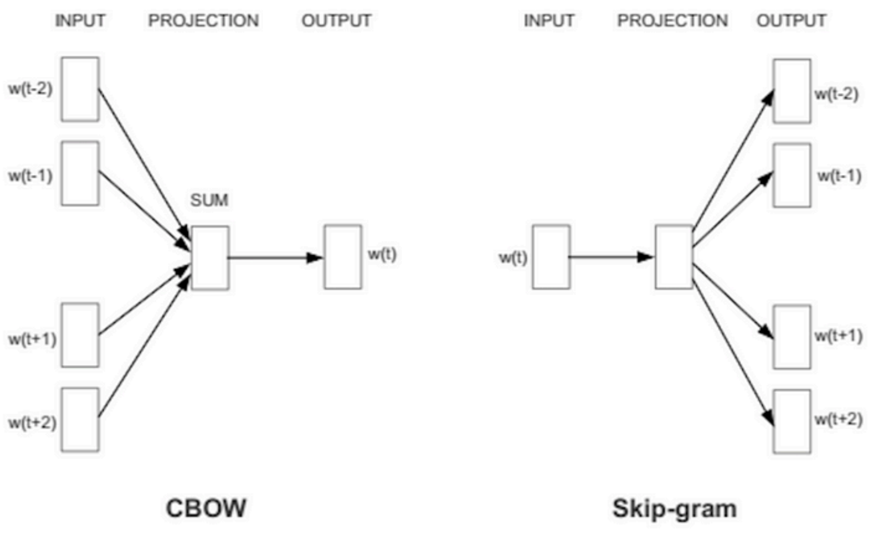
- **Huffman Tree** is a ***lossless data encoding algorithm.*** *The process behind its scheme includes sorting numerical values from a set in order of their frequency. The* *least frequent numbers are gradually eliminated via the Huffman tree**, which **adds the two lowest frequencies from the sorted list in every new "branch."** The sum is then positioned above the **two eliminated lower frequency values, and replaces them in the new sorted list**. Each time a **new branch is created**, it moves the **general direction of the tree** either to **the right (for higher values) or the left (for lower values)** When the sorted list is exhausted and the tree is complete, ***the final value is zero if the tree ended on a left number, or it is one if it ended on the right.*** This is a method of reducing complex code into simpler sequences and is common in video encoding.



File :

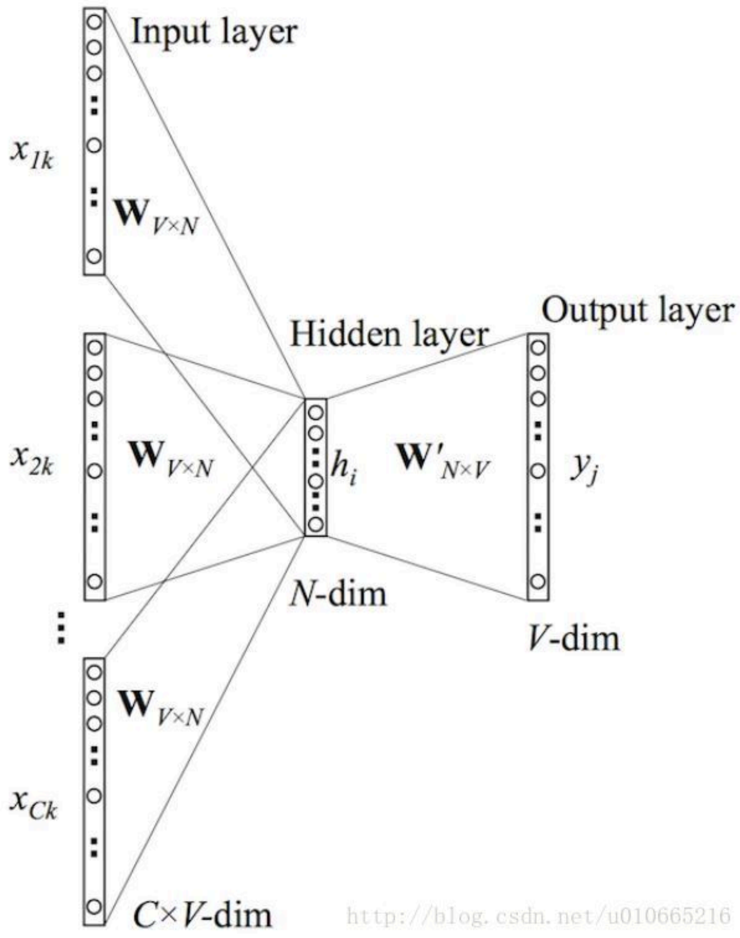
b	p	`	m	j	o	d	a	i	r	u	l	s	e	
1	1	2	2	3	3	3	4	4	5	5	6	6	8	12

CBOW Continues...



Forward propagation

- Next we look at CBOW neural network, the neural network model CBOW neural network model skip-gram is a mirror image of



- the figure above, the **input and output** of the input-output model **skip-gram model** is opposite to so,Here the input layer is the **input context encoded by one-hot** $x1, \ldots, xC$ **composition**, where the window size is C and the vocabulary size is V. The hidden layer is an N-dimensional vector. The final output layer is the output word that is also encoded by one-hot.y. Input vector encoded by one-hot through one $V \times N$ Dimension weight matrix W Connect to the hidden layer; hide the layer through one $N \times V$ Weight matrix $W\prime$ Connect to the output layer.

> The first step is to calculate the hidden layer.Output. as follows

$$h = \frac{1}{C} W \cdot (\sum_{i=1}^{C} x_i) \tag{1}$$

- This output is the **weighted average of the input vectors.** The hidden layer here is significantly different from the hidden layer of the ***skip-gram.***
- The second part is to calculate the ***input at each node of the output layer. as follows:***

$$u_j = v_{wj}^{\prime T} \cdot h \tag{2}$$

- among them $v_{wj}^{\prime T}$ Output matrix $W^{\prime}$
- Finally we calculate the **output of the output layer, the y_j outputas follows:**

$$y_{c,j} = p(w_{y,j}|w_1, \ldots, w_c) = \frac{exp(u_j)}{\sum_{j^{\prime}=1}^{V} exp(u^{\prime} j)} \tag{3}$$

Learn weights by BP (backpropagation) algorithm and stochastic gradient descent

- Learning weight matrix W versus$W^{\prime}$ In the process, we can assign a **random value to these weights to initialize.** The samples are then **trained in order, and the error between the output and the true value is observed one by one, and the gradient of these errors is calculated.** And correct the **weight matrix** in the gradient direction. This method is called **random gradient descent.** But this derived method is called the back propagation error algorithm.

The first is to define the loss function. This loss function is the ***conditional probability of the output word given the input context. It is usually a logarithm, as shown below:**

$$E = -logp(w_O|w_I) \tag{4}$$

$$= -v_{wo}^{T} \cdot h - log \sum_{j^{\prime}=1}^{V} exp(v_{w_{j^{\prime}}}^{T} \cdot h) \tag{5}$$

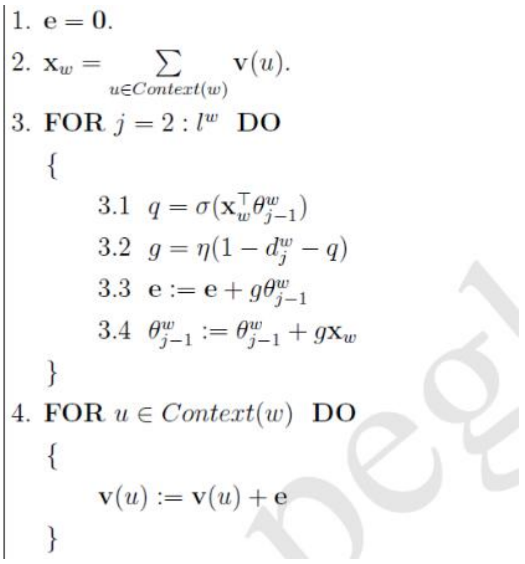
- The next step is to **derive the above probability.** The ***specific derivation process can look at the BP algorithm.*** We get the output weight matrix.W'Update rules:

$$w^{\prime (new)} = w_{ij}^{\prime (old)} - \eta \cdot (y_j - t_j) \cdot h_i \tag{6}$$

- **Equal weight** W' The update rules are as follows:

$$w^{(new)} = w_{ij}^{(old)} - \eta \cdot \frac{1}{C} \cdot EH \tag{7}$$

Psuedo Code



```
1. e = 0.
2. $\mathbf{x}_w = \sum_{u \in Context(w)} \mathbf{v}(u)$.
3. FOR $j = 2 : l^w$ DO
   {
       3.1  $q = \sigma(\mathbf{x}_w^\top \theta_{j-1}^w)$
       3.2  $g = \eta(1 - d_j^w - q)$
       3.3  $\mathbf{e} := \mathbf{e} + g\theta_{j-1}^w$
       3.4  $\theta_{j-1}^w := \theta_{j-1}^w + g\mathbf{x}_w$
   }
4. FOR $u \in Context(w)$ DO
   {
       $\mathbf{v}(u) := \mathbf{v}(u) + \mathbf{e}$
   }
```

2.2.2.Skip-Gram

- In many **natural language processing** tasks, many **word expressions** are determined by their **tf-idf** scores. Even though these **scores tell us the relative importance of a word** in a text, they do not tell us the **semantics of the word. Word2vec is a type of neural network model** that, in the case of an **unlabeled corpus, produces a vector that expresses semantics for words in the corpus.** These vectors are usually useful:

> - Calculating the semantic similarity of two words by word vector
> - Semantic analysis of some supervised NLP tasks such as text categorization

- Before go into detail about the **skip-gram model,** let's first understand the format of the **training data.** The input to the **skip-gram model** is a word w_I output is w_I Context $w_{O,1}, \ldots, w_{O,C}$ the context window size is CC. For example, here is a sentence "I drive my car to the store." If we use "car" as the training input data, the word group {"I", "drive", "my", "to", "the", "store"} is the output. For all these words, we will do one-hot coding. The skip-gram model diagram is as follows:

Forward propagation

- Next, we look at the skip-gram neural network model. The neural network model of **skip-gram** is improved from the **feedforward neural network** model. It is said that the model is **more effective through some techniques based on the feedforward neural network model.** Let's take a look at the image of a **wave-gram neural network model:**

- In the above figure, the input **vector** x **One-hot encoding representing a word,** corresponding ***output vector*** $y1y1, \ldots, yCyC$. ***Weight matrix between input layer and hidden layer*** W ***The iiRow represents the i in the vocabulary*** i ***The weight of the words.*** The next point is : this weight matrix $W's$ the **goal we need to learn (same as** W‘**), because this weight matrix contains weight information for all words in the vocabulary.** In the above model, each output **word vector also has a** $N \times V$ **Dimension output vector** W‘. **The final model also has NNThe hidden layer of the node, we can find the hidden layer node hihiThe input is the weighted sum of the input layer inputs. So because of the input vector** x **one-hot encoding, then only non-zero elements in the vector can produce input to the hidden layer.** So for the input vector x **Where** $x_{k'} = 0, k \neq k'$ **And** $xk‘ = 0, k \neq k‘xk‘ = 0, k \neq k‘$. So the output of the hidden layer is only with the weight matrix k Row related, mathematically proved as follows:

$$h = x^T W = W_{k,.} := v_{wI} \tag{1}$$

- Note that since the input is ***one-hot encoded, there is no need to use the activation function here.*** Similarly, **the model output node** $C \times V$. The **input** is also **calculated from the weighted sum of the corresponding input nodes:***

$$u_{c,j} = v_{wj}^{'T} h \tag{2}$$

- In fact, we also saw from the above figure that ***each word in the output layer is shared weight, so we have*** $u_{c,j} = u_j$ Finally, we ***generate the C through the softmax function.CThe polynomial distribution of words.***

$$p(w_{c,j} = w_{O,c} | w_I) = y_{c,j} = \frac{exp(u_{c,j})}{\sum_{j'=1}^{V} exp(uj')} \tag{3}$$

- To put it bluntly, this value is the ***probability of the jth node of the*** C ***th output word.***

Learn weights by BP (backpropagation) algorithm and stochastic gradient descent

- the input vector of the ***skip-gram model** and *the probabilistic expression of the output*, as well as the goals we learned. Next, we explain in detail the *process of learning weights. The first step is to define the loss function. This loss function is the conditional probability of the output word group. It is usually a logarithm, as shown below:*

$$E = -logp(w_{O,1}, w_{O,2}, \dots, w_{O,C}|w_I) \tag{4}$$

$$= -log \prod_{c=1}^{C} \frac{exp(u_{c,j})}{\sum_{j'=1}^{V} exp(u'_{j})} \tag{5}$$

- The next step is to ***derive the above probability.*** The **specific derivation process** can look at the **BP algorithm**. We get the output **weight matrix.W‘Update rules:**

$$w'^{(new)} = w'^{(old)}_{ij} - \eta \cdot \sum_{c=1}^{C} (y_{c,j} - t_{c,j}) \cdot h_i \tag{6}$$

- From the above update rules, we can find that each update requires **summing the entire vocabulary, so for a large corpus, this computational complexity is very high.** So in practical applications, Google's Mikolov et al. proposed that layered softmax and negative sampling can make the computational complexity much lower.

```
from nltk.tokenize import sent_tokenize, word_tokenize
import warnings

warnings.filterwarnings(action = 'ignore')

import gensim
from gensim.models import Word2Vec


#  Reads ‘alice.txt’ file
sample = open("../input/text-data/alice.txt", "r")
s = sample.read()


# Replaces escape character with space
f = s.replace("\n", " ")


data = []

# iterate through each sentence in the file
for i in sent_tokenize(f):
    temp = []

    # tokenize the sentence into words
    for j in word_tokenize(i):
        temp.append(j.lower())

    data.append(temp)
```

CBOW Model

```
# Create CBOW model
model1 = gensim.models.Word2Vec(data, min_count = 1, size = 100, window = 5)


# Print results
print("Cosine similarity between 'alice' " + "and 'wonderland' - CBOW : ", model1.similarity('alice', 'wonderland'))
print("Cosine similarity between 'alice' " +"and 'machines' - CBOW : ", model1.similarity('alice', 'machines'))
```

SKIP-GRAM Model

```
# Create Skip Gram model
model2 = gensim.models.Word2Vec(data, min_count = 1, size = 100, window = 5, sg = 1)


# Print results
print("Cosine similarity between 'alice' " + "and 'wonderland' - Skip Gram : ", model2.similarity('alice', 'wonderland'))
print("Cosine similarity between 'alice' " + "and 'machines' - Skip Gram : ", model2.similarity('alice', 'machines'))
```

3.Using pre-trained word vectors

1.Glove

- **Official Page : https://nlp.stanford.edu/projects/glove/ (Reading More About Glove)**
- **Original Paper: https://nlp.stanford.edu/pubs/glove.pdf**

Installation of Glove-python

```
# https://medium.com/data-science-group-iitr/word-embedding-2d05d270b285
!pip install glove_python
```

```
import re
import numpy as np

from glove import Corpus, Glove
from nltk.corpus import gutenberg
from multiprocessing import Pool
from scipy import spatial
```

Import training dataset

- Import Shakespeare's Hamlet corpus from nltk library

```
sentences = list(gutenberg.sents('shakespeare-hamlet.txt'))
```

```
print(sentences[0])    # title, author, and year
print(sentences[1])
print(sentences[10])
```

Preprocess data

- Use re module to preprocess data
- Convert all letters into lowercase
- Remove punctuations, numbers, etc.

```
for i in range(len(sentences)):
    sentences[i] = [word.lower() for word in sentences[i] if re.match('^[a-zA-Z]+', word)]
```

```
print(sentences[0])    # title, author, and year
print(sentences[1])
print(sentences[10])
```

Create Corpus instance

- Sentences should be fitted into the Corpus instance
- Recall that GloVe takes advantage of both count-based matrix factorization and local context-based window methods

```
corpus = Corpus()
```

```
corpus.fit(sentences, window = 3)    # window parameter denotes the distance of context
```

```
glove = Glove(no_components = 100, learning_rate = 0.05)
```

Train model

- GloVe model is trained with corpus matrix (global statistics of words)
- Key parameter description
  - **matrix**: co-occurence matrix of the corpus
  - **epochs**: number of epochs (i.e., training iterations)
  - **no_threads**: number of training threads
  - **verbose**: whether to print out the progress messages

```
glove.fit(matrix = corpus.matrix, epochs = 40, no_threads = Pool()._processes, verbose = False)
glove.add_dictionary(corpus.dictionary)    #  supply a word-id dictionary to allow similarity queries
```

Save and load model

- word2vec model can be saved and loaded locally
- Doing so can reduce time to train model again

```
glove.save('glove_model')
glove.load('glove_model')
```

Similarity calculation

- Similarity between embedded words (i.e., vectors) can be computed using metrics such as cosine similarity
- For other metrics and comparisons between them, refer to: https://github.com/taki0112/Vector_Similarity

```
glove.most_similar('king', number = 10)
```

```
# define a function that converts word into embedded vector
def vector_converter(word):
    idx = glove.dictionary[word]
    return glove.word_vectors[idx]
```

```
# define a function that computes cosine similarity between two words
def cosine_similarity(v1, v2):
    return 1 - spatial.distance.cosine(v1, v2)
```

```
v1 = vector_converter('king')
v2 = vector_converter('queen')
```

```
cosine_similarity(v1, v2)
```

2.Sentence modeling

Reference Paper : https://arxiv.org/abs/1704.05358

- One of the methods to represent sentences as vectors (Mu et al 2017)
- Computing vector representations of each embedded word, and weight average them using PCA
- If there are n words in a sentence, select N words with high explained variance (n>N)
- Most of "energy" (around 80%) can be containted using only 4 words (N=4) in the original paper (Mu et al 2017)

```
import re
import numpy as np

from gensim.models import Word2Vec
from nltk.corpus import gutenberg
from multiprocessing import Pool
from scipy import spatial
from sklearn.decomposition import PCA
```

```
sentences = list(gutenberg.sents('shakespeare-hamlet.txt'))   # import the corpus and convert into a list
```

```
print('Type of corpus: ', type(sentences))
print('Length of corpus: ', len(sentences))
```

```
print(sentences[0])    # title, author, and year
print(sentences[1])
print(sentences[10])
```

```
for i in range(len(sentences)):
    sentences[i] = [word.lower() for word in sentences[i] if re.match('^[a-zA-Z]+', word)]
```

```
print(sentences[0])    # title, author, and year
print(sentences[1])
print(sentences[10])
```

```
# set threshold to consider only sentences longer than certain integer
threshold = 5

for i in range(len(sentences)):
    if len(sentences[i]) < 5:
        sentences[i] = None
```

```
sentences = [sentence for sentence in sentences if sentence is not None]
print('Length of corpus: ', len(sentences))
```

```
model = Word2Vec(sentences = sentences, size = 100, sg = 1, window = 3, min_count = 1, iter = 10, workers = Pool()._processes)
model.init_sims(replace = True)
```

```
# converting each word into its vector representation
for i in range(len(sentences)):
    sentences[i] = [model[word] for word in sentences[i]]
```

```
print(sentences[0])    # vector representation of first sentence
```

```
# define function to compute weighted vector representation of sentence
# parameter 'n' means number of words to be accounted when computing weighted average
def sent_PCA(sentence, n = 2):
    pca = PCA(n_components = n)
    pca.fit(np.array(sentence).transpose())
    variance = np.array(pca.explained_variance_ratio_)
    words = []
    for _ in range(n):
        idx = np.argmax(variance)
        words.append(np.amax(variance) * sentence[idx])
        variance[idx] = 0
    return np.sum(words, axis = 0)
```

```
sent_vectorized = []

# computing vector representation of each sentence
for sentence in sentences:
    sent_vectorized.append(sent_PCA(sentence))
```

```
# vector representation of first sentence
list(sent_PCA(sentences[0])) == list(sent_vectorized[0])
```

```
# define a function that computes cosine similarity between two words
def cosine_similarity(v1, v2):
    return 1 - spatial.distance.cosine(v1, v2)
```

```
# similarity between 11th and 101th sentence in the corpus
print(cosine_similarity(sent_vectorized[10], sent_vectorized[100]))
```

3.Doc2Vec

- Python implementation and application of doc2vec with Gensim
- Original paper: Le, Q., & Mikolov, T. (2014). Distributed representations of sentences and documents. In Proceedings of the 31st International Conference on Machine Learning (ICML-14) (pp. 1188-1196).

```
import re
import numpy as np

from gensim.models import Doc2Vec
from gensim.models.doc2vec import TaggedDocument
from nltk.corpus import gutenberg
from multiprocessing import Pool
from scipy import spatial
```

Import training dataset

- Import Shakespeare's Hamlet corpus from nltk library

```
sentences = list(gutenberg.sents('shakespeare-hamlet.txt'))   # import the corpus and convert into a list
```

```
print('Type of corpus: ', type(sentences))
print('Length of corpus: ', len(sentences))
```

```
print(sentences[0])    # title, author, and year
print(sentences[1])
print(sentences[10])
```

Preprocess data

- Use re module to preprocess data
- Convert all letters into lowercase
- Remove punctuations, numbers, etc.
- For the doc2vec model, input data should be in format of iterable **TaggedDocuments**"
  - Each TaggedDocument instance comprises **words** and **tags**
  - Hence, each document (i.e., a sentence or paragraph) should have a unique tag which is **identifiable**

```
for i in range(len(sentences)):
    sentences[i] = [word.lower() for word in sentences[i] if re.match('^[a-zA-Z]+', word)]
```

```
print(sentences[0])    # title, author, and year
print(sentences[1])
print(sentences[10])

for i in range(len(sentences)):
    sentences[i] = TaggedDocument(words = sentences[i], tags = ['sent{}'.format(i)])    # converting each sentence into a TaggedDocument
```

```
sentences[0]
```

Create and train model

- Create a doc2vec model and train it with Hamlet corpus
- Key parameter description (https://radimrehurek.com/gensim/models/doc2vec.html)
- **documents**: training data (has to be iterable TaggedDocument instances)
    - **size**: dimension of embedding space
    - **dm**: DBOW if 0, distributed-memory if 1
    - **window**: number of words accounted for each context (if the window size is 3, 3 word in the left neighorhood and 3 word in the right neighborhood are considered)
    - **min_count**: minimum count of words to be included in the vocabulary
    - **iter**: number of training iterations
    - **workers**: number of worker threads to train

```
model = Doc2Vec(documents = sentences, dm = 1, size = 100, window = 3, min_count = 1, iter = 10, workers = Pool()._processes)
model.init_sims(replace = True)
```

Save and load model

- doc2vec model can be saved and loaded locally
- Doing so can reduce time to train model again

```
model.save('doc2vec_model')
model = Doc2Vec.load('doc2vec_model')
```

Similarity calculation

- Similarity between embedded words (i.e., vectors) can be computed using metrics such as cosine similarity
- For other metrics and comparisons between them, refer to: https://github.com/taki0112/Vector_Similarity

```
v1 = model.infer_vector('sent2')    # in doc2vec, infer_vector() function is used to infer the vector embedding of a document
v2 = model.infer_vector('sent3')
```

```
model.most_similar([v1])
# define a function that computes cosine similarity between two words
def cosine_similarity(v1, v2):
    return 1 - spatial.distance.cosine(v1, v2)
```

```
cosine_similarity(v1, v2)
```

4.Word2Vec

- Python implementation and application of word2vec with Gensim
- Original paper: Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781.

```
import re
import numpy as np

from gensim.models import Word2Vec
from nltk.corpus import gutenberg
from multiprocessing import Pool
from scipy import spatial
```

Import training dataset

- Import Shakespeare's Hamlet corpus from nltk library

```
sentences = list(gutenberg.sents('shakespeare-hamlet.txt'))   # import the corpus and convert into a list
```

```
print('Type of corpus: ', type(sentences))
print('Length of corpus: ', len(sentences))
```

```
print(sentences[0])    # title, author, and year
print(sentences[1])
print(sentences[10])
```

Preprocess data

- Use re module to preprocess data
- Convert all letters into lowercase
- Remove punctuations, numbers, etc.

```
for i in range(len(sentences)):
    sentences[i] = [word.lower() for word in sentences[i] if re.match('^[a-zA-Z]+', word)]
```

```
print(sentences[0])    # title, author, and year
print(sentences[1])
print(sentences[10])
```

Create and train model

- Create a word2vec model and train it with Hamlet corpus
- Key parameter description (https://radimrehurek.com/gensim/models/word2vec.html)
  - sentences: training data (has to be a list with tokenized sentences)
  - size: dimension of embedding space
  - sg: CBOW if 0, skip-gram if 1
  - window: number of words accounted for each context (if the window size is 3, 3 word in the left neighorhood and 3 word in the right neighborhood are considered)
  - min_count: minimum count of words to be included in the vocabulary
  - iter: number of training iterations
  - workers: number of worker threads to train

```
model = Word2Vec(sentences = sentences, size = 100, sg = 1, window = 3, min_count = 1, iter = 10, workers = Pool()._processes)
model.init_sims(replace = True)
```

Save and load model

- word2vec model can be saved and loaded locally
- Doing so can reduce time to train model again

```
model.save('word2vec_model')
model = Word2Vec.load('word2vec_model')
```

Similarity calculation

- Similarity between embedded words (i.e., vectors) can be computed using metrics such as cosine similarity
- For other metrics and comparisons between them, refer to: https://github.com/taki0112/Vector_Similarity

```
model.most_similar('hamlet')
```

```
v1 = model['king']
v2 = model['queen']
```

```
# define a function that computes cosine similarity between two words
def cosine_similarity(v1, v2):
    return 1 - spatial.distance.cosine(v1, v2)
```

```
cosine_similarity(v1, v2)
```

4.Training your own Word Vectors

- Word2Vec requires that a format of list of list for **training where every document is contained in a list and every list contains list of tokens** of that documents. I won't be covering the ***pre-preprocessing part here. So let's take an example list of list to train our word2vec**

Pretrained Google News Vector Model

```
from gensim.models import Word2Vec, KeyedVectors

#loading the downloaded model
model = KeyedVectors.load_word2vec_format('../input/word2vec-google/GoogleNews-vectors-negative300.bin', binary=True)


#the model is loaded. It can be used to perform all of the tasks mentioned above.

# getting word vectors of a word
dog = model['dog']

#performing king queen magic
print(model.most_similar(positive=['woman', 'king'], negative=['man']))

#picking odd one out
print(model.doesnt_match("breakfast cereal dinner lunch".split()))

#printing similarity index
print(model.similarity('woman', 'man'))
```

Own Word2Vector Model

```
sentence=[['Neeraj','Boy'],['Sarwan','is'],['good','boy']]
```