# Notes

## Roadmap

1. Basic
2. Word2vec, AvgWord2Vec
3. RNN, LSTM, GRU, RNN
4. Word Embedding
5. Transformer/ BERT

## Terminology

1. corpus → Paragraph
2. Documents → Sentences
3. Vocabulary → unique words
4. Tokenization →
    a. Paragraph to Sentence
    b. Sentences to Words

## Tokenizer

1. Paragraph to Sentence
2. Sentence to Words

```
import nltk
nltk.download(<Required Packages>)

# 1 sent_tokenize
```

```
# 2 word_tokenize
# 3 wordpunct_tokenize -> Also consider Punctuations
# 4 TreeBankWordTokenizer -> Consider Full Stop as Part of Word
```

## Stemming

https://www.ibm.com/topics/stemming

https://www.geeksforgeeks.org/introduction-to-stemming/

1. PorterStemmer :

   It is based on the idea that the suffixes in the English language are made up of a combination of smaller and simpler suffixes. This stemmer is known for its speed and simplicity.

   ```
   #  It is based on the idea that the suffixes in the English l
   from nltk.stem import PorterStemmer
   ps = PorterStemmer()

   program : program
   programs : program
   programer : program
   programing : program
   programers : program
   eat : eat
   eaten :  eaten
   jump :  jump
   cried :  cri
   laughed :  laugh
   fairly :  fairli
   sporty :  sporti
   goes :  goe
   ```

2. Snowball Stemmer

Performs better than PorterStemmer, Multingual, Porter2Stemmer, Improves Performance When Addes to PorterStemmer

```
rom nltk.stem import SnowballStemmer
ss = SnowballStemmer(language='english')

program : program
programs : program
programer : program
programing : program
programers : program
eating :  eat
jumped :  jump
cried :  cri
laughed :  laugh
fairly :  fair
sporty :  sporti
goes :  goe
```

3. RegexpStemmer class

   The Regexp Stemmer, or Regular Expression Stemmer, is a stemming algorithm that utilizes regular expressions to identify and remove suffixes from words. It allows users to define custom rules for stemming by specifying patterns to match and remove.

```
from nltk.stem import RegexpStemmer
rs = RegexpStemmer('ing$|s$|able$|ed$', min=4)

- 'ing$' Removes From Last
- '$ing' Removes From Beginnng
- 'ing' Removes Complete word

program : program
programs : program
```

```
programer : programer
programing : program
programers : programer
eating :  eat
jumped :  jump
cried :  cri
laughable :  laugh
```

## Lemmatization

https://www.geeksforgeeks.org/python-lemmatization-with-nltk/

- Similar to Stemming
- Convert to Root Word Called Lemma.
- Valid word.

```
from nltk.stem import WordNetLemmatizer
lm = WordNetLemmatizer()

'''
Pos
Noun - n
Verb - v
adjective - a
adverb - r
'''
by adding this as a Parametre it will treat word as that pos

eaten :  eaten
jumped :  jump
cried :  cry
laughed :  laughed
```

```
fairly :  fairly
sporty :  sporty
goes :  go
rocks : rock
corpora : corpus
better : good
```

## Stopwords

Words that does not make much sense to the Computer.

like is

```
from nltk.corpus import stopwords
stopwords.words('english')
```

## POS tagging

Give the Part of speech of for a specific word

```
nltk.pos_tag(<Accept List>)
```

## Named Entity Recognition (NER)

It Give an Entity to the words such as Person, Date, Time Place etc.

```
import matplotlib.pyplot as plt
from nltk.tree import Tree

# Convert to tree object and display with Matplotlib
```

```
chunked = nltk.ne_chunk(tagged)
tree = Tree.fromstring(str(chunked))
tree.pretty_print()  # Text representation
```

## One Hot Encoding

One hot encoding is a technique that we use to represent categorical variables as numerical values in a machine learning model.

- Find Unique Words - Vocabulary
- if word is present in a vocabulary then 1 else 0.
- shape is no.of.words.in.sentence * no.of.words.in Vocabulary

**Advantage**

1. Simple and Easy to Implement.
2. Sklearn - OneHotEncoder

**Disadvantage**

1. Sparse Matrix - Overfitting
2. ML Algo - Work on Fixed SIze
3. No Semantic Meaning is getting Captured
4. Out of vocabulary

Code

Sklearn OneHotEncoder

```
#one hot encoding using OneHotEncoder of Scikit-Learn

import pandas as pd
from sklearn.preprocessing import OneHotEncoder
data = {'Employee id': [10, 20, 15, 25, 30],
        'Gender': ['M', 'F', 'F', 'M', 'F'],
```

```
        'Remarks': ['Good', 'Nice', 'Good', 'Great', 'Nice'],
        }
df = pd.DataFrame(data)
#Extract categorical columns from the dataframe
#Here we extract the columns with object datatype as they are th
categorical_columns = df.select_dtypes(include=['object']).colum

#Initialize OneHotEncoder
encoder = OneHotEncoder(sparse_output=False)
one_hot_encoded = encoder.fit_transform(df[categorical_columns])

#Create a DataFrame with the one-hot encoded columns
#We use get_feature_names_out() to get the column names for the
one_hot_df = pd.DataFrame(one_hot_encoded, columns=encoder.get_f

# Concatenate the one-hot encoded dataframe with the original da
df_encoded = pd.concat([df, one_hot_df], axis=1)

# Drop the original categorical columns
df_encoded = df_encoded.drop(categorical_columns, axis=1)

# Display the resulting dataframe
print(f"Encoded Employee data : \n{df_encoded}")

Output :
   Employee id  Gender_F  Gender_M  Remarks_Good  Remarks_Great
0           10       0.0       1.0           1.0            0.0
1           20       1.0       0.0           0.0            0.0
2           15       1.0       0.0           1.0            0.0
3           25       0.0       1.0           0.0            1.0
4           30       1.0       0.0           0.0            0.0
```

Pandas get_dummies

```
import numpy as np
import pandas as pd
```

```
data = pd.DataFrame(data)
print(data.head())

data['Gender'].unique()
data['Remarks'].unique()
data['Gender'].value_counts()
data['Remarks'].value_counts()

oneHotEncodedData = pd.get_dummies(data, columns = ['Gender', 'R
oneHotEncodedData

Output :
Employee id Gender_F    Gender_M    Remarks_Good    Remarks_Grea
0   10  False   True    True    False   False
1   20  True    False   False   False   True
2   15  True    False   True    False   False
3   25  False   True    False   True    False
4   30  True    False   False   False   True
```

## Bag of Words

1. Lower all the charecters
2. Remove all the stopwords
3. count frequency of each word and append it in dictionary

- for word if it is in dictionary set count as 1 else 0
- if the word get repeated increment its count.

```
X = []
for data in dataset:
```

```
    vector = []
    for word in freq_words:
        if word in nltk.word_tokenize(data):
            vector.append(1)
        else:
            vector.append(0)
    X.append(vector)
X = np.asarray(X)
```

Advantage

1. Simple and Easy to Implement
2. Fixed Size

Disadvantage

1. Sparse Matrix - Overfitting
2. Ordering gets Changed which changes Meaning
3. Semantic Meaning is Not getting Captured.

# TF - IDF

Term frequency–inverse document frequency (TF-IDF) is

**a technique used in natural language processing (NLP) to measure the importance of words in a document**

. It's a fundamental concept in text representation and information retrieval.

TF-IDF is a numerical statistic that considers the frequency of a word in a document and its rarity across a collection of documents, called a corpus. The higher a term's TF-IDF score, the more important or relevant it is.

$$TF(t,d) = \frac{number\ of\ times\ t\ appears\ in\ d}{total\ number\ of\ terms\ in\ d}$$

$$IDF(t) = log\frac{N}{1+df}$$

$$TF-IDF(t,d) = TF(t,d) * IDF(t)$$

Advantage

1. Intuitive
2. Fixed Size
3. Word Importance is getting captures

Disadvantage

1. Sparsity - Overfitting
2. out of vocabulary

```
# TF - IDF
from sklearn.feature_extraction.text import TfidfVectorizer
d0 = 'good boy'
d1 = 'good girl'
d2 = 'good girl boy'

# merge documents into a single corpus
string = [d0, d1, d2]

tfidf = TfidfVectorizer()
result = tfidf.fit_transform(string)

# get indexing
print('\nWord indexes:')
print(tfidf.vocabulary_)
```

```
# display tf-idf values
print('\ntf-idf value:')
print(result)

# in matrix form
print('\ntf-idf values in matrix form:')
print(result.toarray())

output :
tf-idf value:
  (0, 2)    0.6133555370249717
  (0, 0)    0.7898069290660905
  (1, 2)    0.6133555370249717
  (1, 1)    0.7898069290660905
  (2, 2)    0.48133416873660545
  (2, 0)    0.6198053799406072
  (2, 1)    0.6198053799406072

tf-idf values in matrix form:
[[0.78980693 0.         0.61335554]
 [0.         0.78980693 0.61335554]
 [0.61980538 0.61980538 0.48133417]]
```

## Word Embeddings

- used for Representation of words for text.
- Typically in Real valued vector that encodes the meaning of words such that word with similar meaning are close.

1. Count/Frequency -
   a. One Hot Encoding

b. Bag of Words

c. TF - IDF

2. Deep Learning Model:

a. Word2Vec -

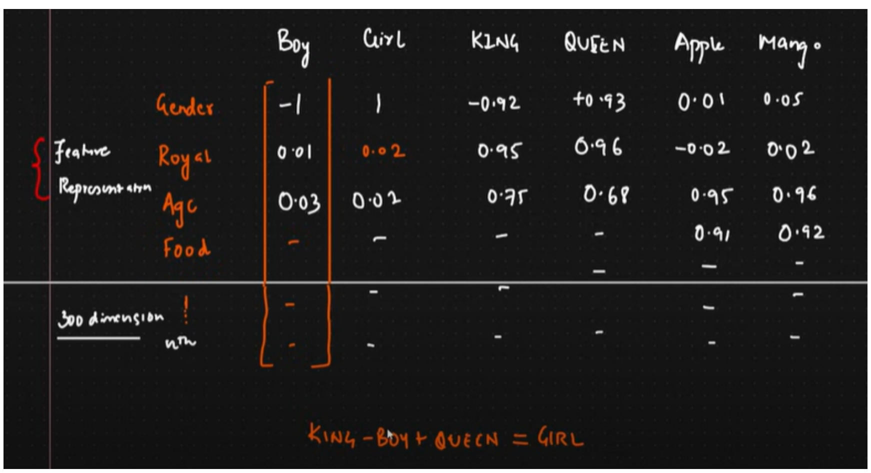
i. Continuous Bag of Words

ii. Skipgram

## Word2Vec

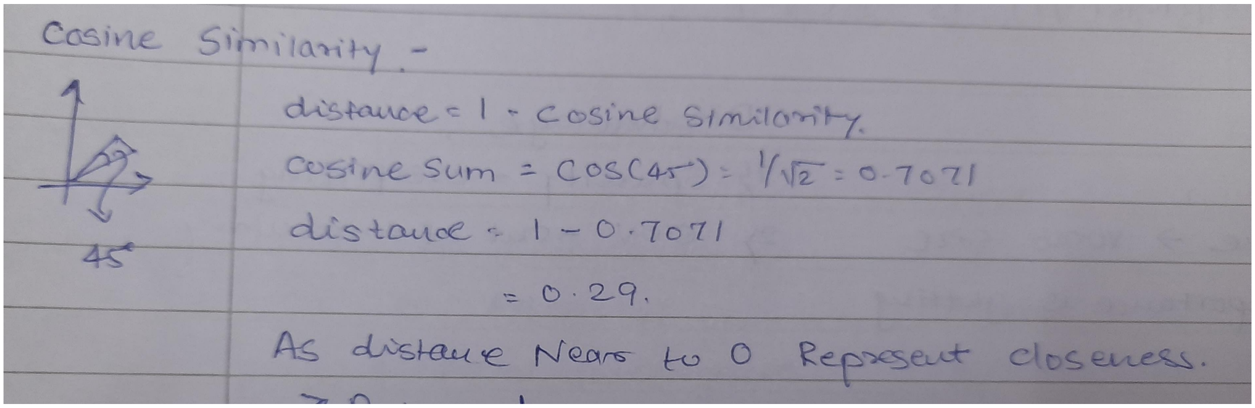
- Neural Network Model to Learn Association from Corpus of Text
- Once Trained it is able to detect  Synonyms and Complete words

Sample Feature Vector




### Cosine Similarity

Cosine similarity is a mathematical metric used to calculate the similarity between two vectors in a multi-dimensional space. It measures the cosine of the angle between the two vectors, resulting in a value between 0 and 1. This value indicates the degree of similarity between the vectors.


Cosine Similarity -

distance = 1 - cosine similarity.

cosine sum = $\cos(45^\circ) = 1/\sqrt{2} = 0.7071$

distance = $1 - 0.7071$

$= 0.29.$

As distance Nears to 0 Represent closeness.

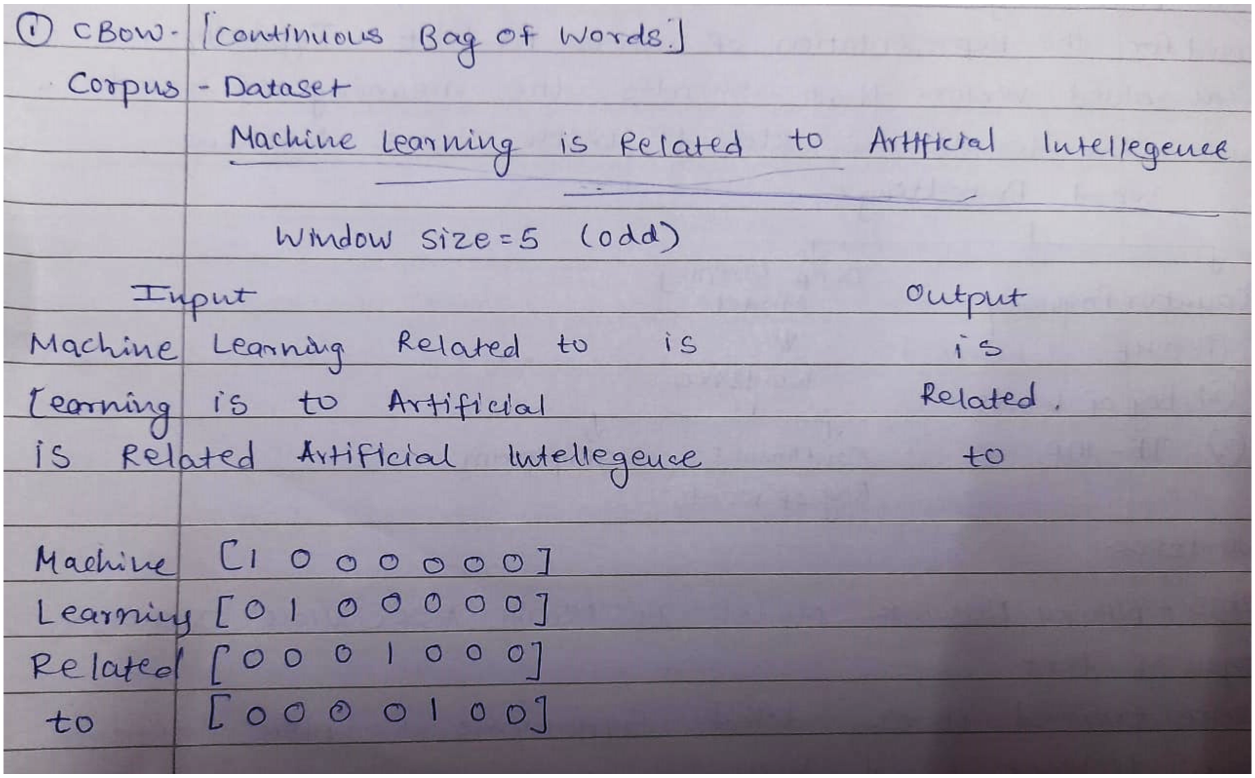


Applications -

• Natural Language Processing (NLP): for finding similar documents, measuring sentence similarity, and detecting plagiarism
• Information Retrieval: for ranking search results based on relevance

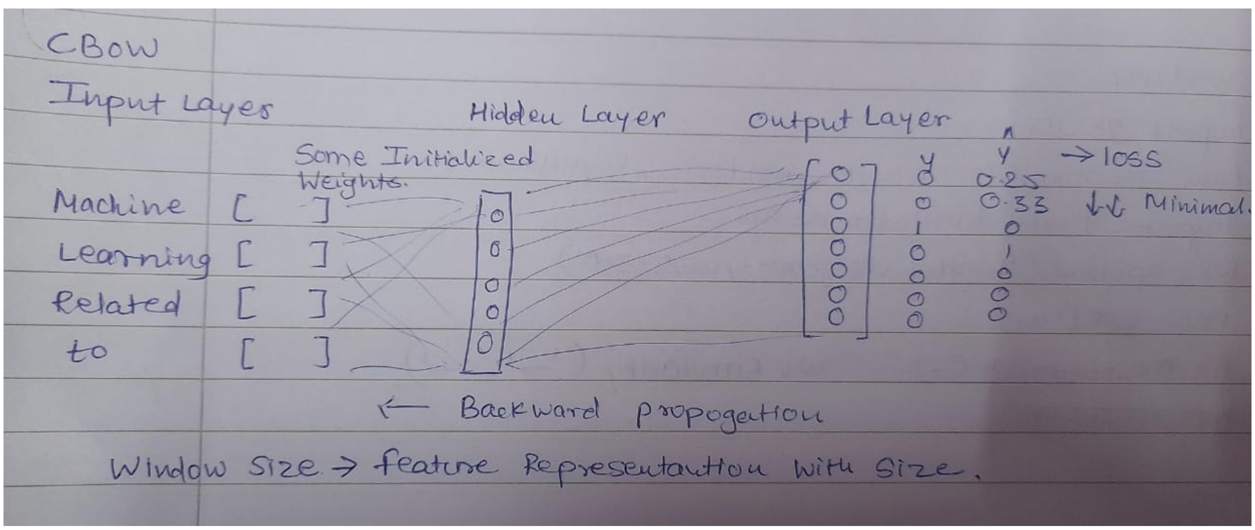
1. Continuous Bag of Words

CBOW predicts a target word based on the context words. It tries to guess the current word by looking at the surrounding words. This model is useful when the context is more important than the specific words


① CBOW - [Continuous Bag of Words.]
Corpus - Dataset
Machine Learning is Related to Artificial Intellegence
Window size = 5 (odd)

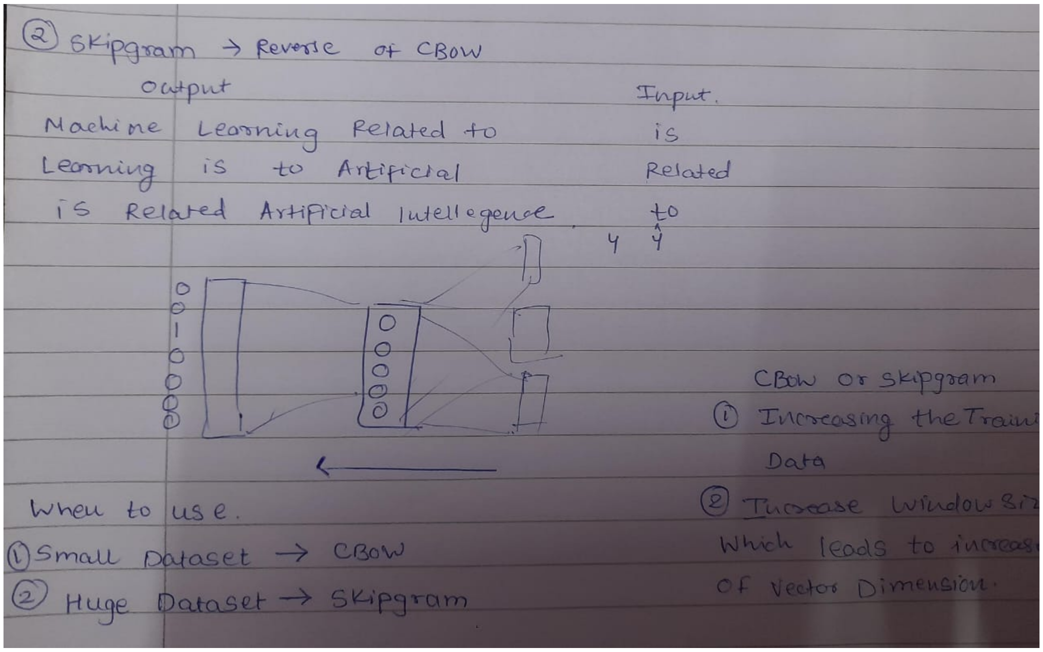
| Input | | | | | Output |
|---|---|---|---|---|---|
| Machine | Learning | Related | to | is | is |
| Learning | is | to | Artificial | | Related |
| is | Related | Artificial | Intellegence | | to |

Machine [1 0 0 0 0 0 0]
Learning [0 1 0 0 0 0 0]
Related [0 0 0 1 0 0 0]
to [0 0 0 0 1 0 0]




2. Skipgram

Skipgram predicts the context words given a target word. It's the opposite of CBOW. This model is useful when the specific words are more important than the context.




The main difference between them is the direction of prediction. CBOW predicts the target word from the context, while Skipgram predicts the context from the target word.

CBOW → Small Dataset

Skipgram → Huge Dataset

1. Pretrained Word2Vec

Pre-trained vectors trained on a part of the 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 'Distributed Representations of Words and Phrases and their Compositionality'

```
import gensim
from gensim.models import Wor2Vec, Keyedvector
```

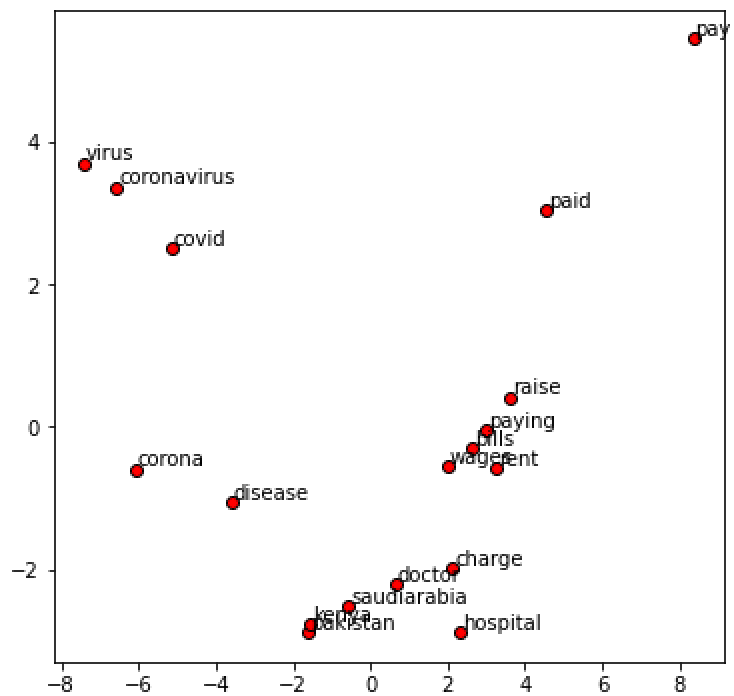
```
import gensim.downloader as api
wv = api.load('word2vec-google-news-300')

vec_king = wv['king'] # Provide vector word Embedding for word

wv.most_similar('google') # Finds the Most similar words

wv.similarity('modi', 'trump') # Check the Similarity between 2

vec = wv['king'] - wv['man'] + wv['woman']
vec
wv.most_similar([vec])
```

https://medium.com/@manansuri/a-dummys-guide-to-word2vec-456444f3c673

2. Train Word2Vec for our dataset

```
from gensim.models import Word2Vec

sentences = [line.split() for line in texts]

# Training Model
w2v =Word2Vec(sentences, size=100, window=5, workers=4, iter=10,
words = list(w2v.wv.vocab)

# Visualization
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE

def display_pca_scatterplot(model, words=None, sample=0):
    if words == None:
        if sample > 0:
            words = np.random.choice(list(model.vocab.keys()), s
        else:
```

```
        words = [ word for word in model.vocab ]

    word_vectors = np.array([model[w] for w in words])

    twodim = PCA().fit_transform(word_vectors)[:,:2]

    plt.figure(figsize=(6,6))
    plt.scatter(twodim[:,0], twodim[:,1], edgecolors='k', c='r')
    for word, (x,y) in zip(words, twodim):
        plt.text(x+0.05, y+0.05, word)

display_pca_scatterplot(w2v,['coronavirus', 'covid', 'virus', 'c
                             'pay', 'paying', 'paid', 'wages',

# Exporting Model
w2v.save("word2vec.model")

# For Further Use
model = Word2Vec.load("word2vec.model")
model.train([["hello", "world"]], total_examples=1, epochs=1)
```




Visualization of Words Realation on 2D Graph