**Exp1 : GenAI- Word2vec**

**Pre-Trained Word Embedding in NLP**

Word Embedding is an important term in Natural Language Processing and a significant breakthrough in deep learning that solved many problems. In this article, we’ll be looking into what pre-trained word embeddings in NLP are.

**Table of Content**

* [Word Embeddings](https://www.geeksforgeeks.org/pre-trained-word-embedding-in-nlp/#word-embeddings)
* [Challenges in building word embedding from scratch](https://www.geeksforgeeks.org/pre-trained-word-embedding-in-nlp/#challenges-in-building-word-embedding-from-scratch)
* [Pre Trained Word Embeddings](https://www.geeksforgeeks.org/pre-trained-word-embedding-in-nlp/#pre-trained-word-embeddings)
* [Word2Vec](https://www.geeksforgeeks.org/pre-trained-word-embedding-in-nlp/#word2vec)

**Word Embeddings**

Word embedding is an approach in [Natural language Processing](https://www.geeksforgeeks.org/natural-language-processing-nlp-tutorial/) where raw text gets converted to numbers/vectors. As deep learning models only take numerical input this technique becomes important to process the raw data. It helps in capturing the semantic meaning as well as the context of the words. A real-valued vector with various dimensions represents each word.

There are certain methods of generating word embeddings such as [BOW (Bag of words)](https://www.geeksforgeeks.org/bag-of-words-bow-model-in-nlp/), TF-IDF, Glove, BERT embeddings, etc. The earlier methods only converted the words without extracting the semantic relationship and context. But the recent ones such as BERT embeddings, which is a pre-trained word embedding model capture the full context of the word as well as the semantic relationships of the word within the sentence.

**Challenges in building word embedding from scratch**

Training word embeddings from scratch is possible but it is quite challenging due to **large trainable parameters** and **sparsity of training data**. These models need to be trained on a large number of datasets with rich vocabulary and as there are large number of parameters, it makes the**training slower**. So, it’s quite challenging to train a word embedding model on an individual level.

**Pre Trained Word Embeddings**

There’s a solution to the above problem, i.e., using pre-trained word embeddings. Pre-trained word embeddings are trained on large datasets and capture the syntactic as well as semantic meaning of the words. This technique is known as **transfer learning** in which you take a model which is trained on large datasets and use that model on your own similar tasks.

There are two broad classifications of pre trained word embeddings – **word-level and character-level**. We’ll be looking into two types of word-level embeddings i.e. [Word2Vec](https://www.geeksforgeeks.org/python-word-embedding-using-word2vec/) and [GloVe](https://www.geeksforgeeks.org/pre-trained-word-embedding-using-glove-in-nlp-models/) and how they can be used to generate embeddings.

**Word2Vec**

Word2Vec is one of the most popular pre trained word embeddings developed by Google. It is trained on Good news dataset which is an extensive dataset. As the name suggests, it represents each word with a collection of integers known as a vector. The vectors are calculated such that they show the semantic relation between words.

A popular example of how semantic relation is made is the king queen example:

King - Man + Woman ~ Queen

Word2vec is a feed-forward neural network which consists of two main models – **Continuous Bag-of-Words** (CBOW) and [**Skip-gram**](https://www.geeksforgeeks.org/implement-your-own-word2vecskip-gram-model-in-python/) model. The continuous bag of words model learns the target word from the adjacent words whereas in the skip-gram model, the model learns the adjacent words from the target word. They are completely opposite of each other.

Firstly, the size of context window is defined. Context window is a sliding window which runs through the whole text one word at a time. It basically refers to the number of words appearing on the right and left side of the focus word. eg. if size of the context window is set to 2, then it will include 2 words on the right as well as left of the focus word.

Focus word is our target word for which we want to create the embedding / vector representation. Generally, focus word is the middle word but in the example below we’re taking last word as our target word. The neighbouring words are the words that appear in the context window. These words help in capturing the context of the whole sentence. Let’s understand this with the help of an example.

Suppose we have a sentence – “He poured himself a cup of coffee”. The target word here is “himself”.

**Continuous Bag-Of-Words** –

input = [“He”, “poured”, “a”, “cup”]

output = [“himself”]

**Skip-gram model** –

input = [“himself”]

output = [“He”, “poured”, “a”, “cup”]

This can be used to generate high-quality word embeddings. You can learn more about these word representations from [https://arxiv.org/pdf/1301.3781.pdf]

**Code**

To generate word embeddings using pre trained word word2vec embeddings, first download the model bin file from [here](https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21pQmM/edit?resourcekey=0-wjGZdNAUop6WykTtMip30g). Then import all the necessary libraries needed such as gensim (will be used for initialising the pre trained model from the bin file.

* Python

|  |
| --- |
| #import gensim library  from gensim.models import Word2Vec  from gensim.models import KeyedVectors  #replace with the path where you have downloaded your model.  pretrained\_model\_path = 'GoogleNews-vectors-negative300.bin.gz'  #initialise the pre trained model using load\_word2vec\_format from gensim module.  word\_vectors = KeyedVectors.load\_word2vec\_format(pretrained\_model\_path, binary=True)    # Calculate cosine similarity between word pairs  word1 = "early"  word2 = "seats"  #calculate the similarity  similarity1 = word\_vectors.similarity(word1, word2)  #print final value  print(similarity1)    word3 = "king"  word4 = "man"  #calculate the similarity  similarity2 = word\_vectors.similarity(word3, word4)  #print final value  print(similarity2) |

**Output:**

0.035838068  
0.2294267

The above code initialises word2vec model using gensim library. It calculates the cosine similarity between words. As you can see the second value is comparatively larger than the first one (these values ranges from -1 to 1), so this means that the words “king” and “man” have more similarity.

We can also find words which are most similar to the given word as parameter

* Python3

|  |
| --- |
| # finding most similar word embeddings with King  king = word\_vectors.most\_similar('King')  print(f'Top 10 most similar words to "King" are : {king}') |

**Output:**

Top 10 most similar words to "King" are : [('Jackson', 0.5326348543167114),   
('Prince', 0.5306329727172852),   
('Tupou\_V.', 0.5292826294898987),   
('KIng', 0.5227501392364502),   
('e\_mail\_robert.king\_@', 0.5173623561859131),   
('king', 0.5158917903900146),   
('Queen', 0.5157250165939331),   
('Geoffrey\_Rush\_Exit', 0.49920955300331116),   
('prosecutor\_Dan\_Satterberg', 0.49850785732269287),   
('NECN\_Alison', 0.49128594994544983)]

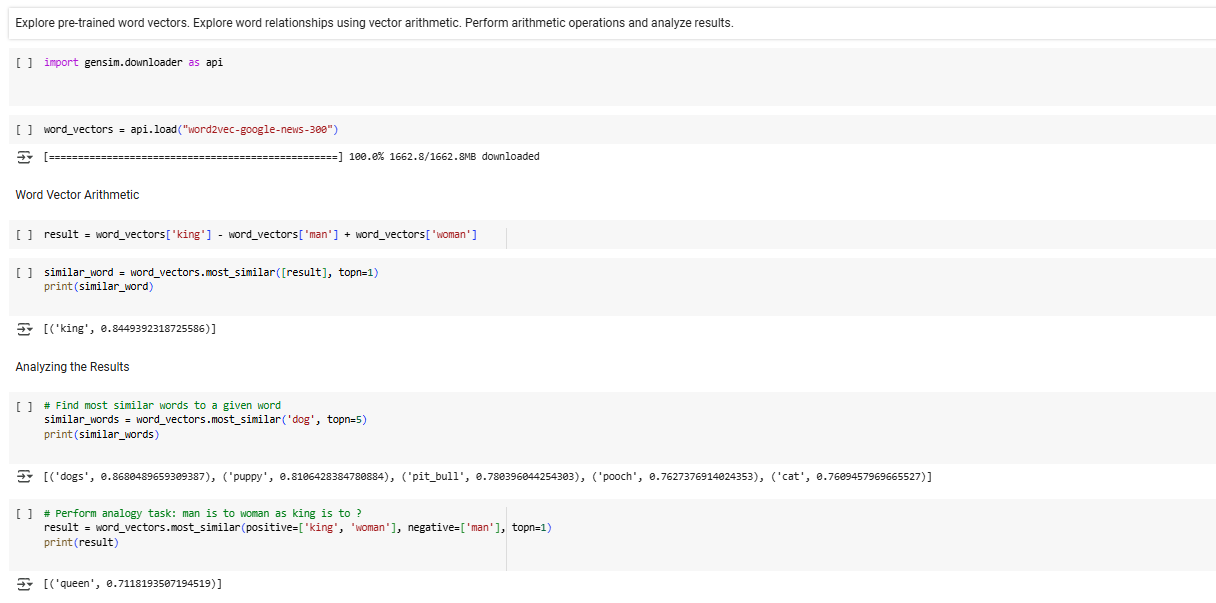
**Program :**

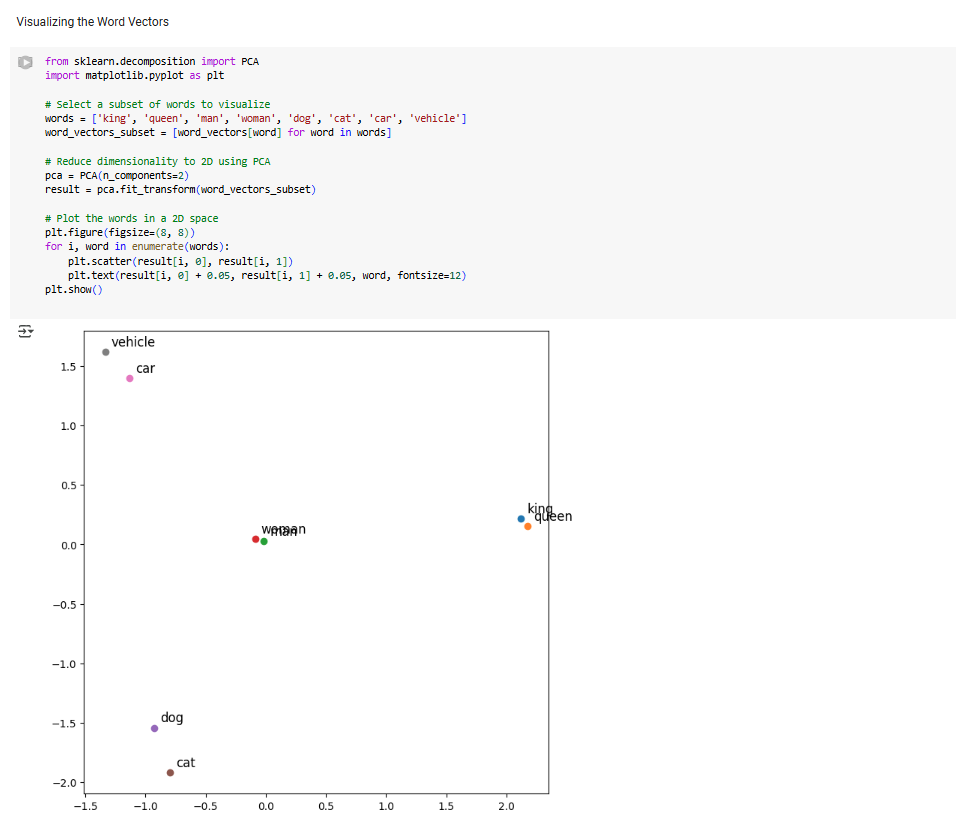
1. Explore pre-trained word vectors. Explore word relationships using vector arithmetic. Perform arithmetic operations and analyse results.

Aim:

Explore pre-trained word vectors. Explore word relationships using vector arithmetic. Perform arithmetic operations and analyse results

Program:





Result:

The word2vec arithmetic were executed successfully.