Vatsal Follow

Jul 29, 2021 · 8 min read ★ · Listen

Save

Word2Vec Explained

Explaining the Intuition of Word2Vec & Implementing it in Python

Table of Contents

Introduction

Word2Vec is a recent breakthrough in the world of NLP. Tomas Mikolov a Czech computer scientist and currently a researcher at CIIRC (Czech Institute of Informatics, Robotics and Cybernetics) was one of the leading contributors towards the research and implementation of word2vec. Word embeddings are an integral part of solving many problems in NLP. They depict how humans understand language to a machine. You can imagine them as a vectorized representation of text. Word2Vec, a common method of generating word embeddings, has a variety of applications such as text similarity, recommendation systems, sentiment analysis, etc.

What is a Word Embedding?

Word embeddings is a technique where individual words are transformed into a numerical representation of the word (a vector). Where each word is mapped to one vector, this vector is then learned in a way which resembles a neural network. The vectors try to capture various characteristics of that word with regard to the overall text. These characteristics can include the semantic relationship of the word, definitions, context, etc. With these numerical representations, you can do many things like identify similarity or dissimilarity between words.

Clearly, these are integral as inputs to various aspects of machine learning. A machine cannot process text in its raw form, thus converting the text into an embedding will allow users to feed the embedding to classic machine learning models. The simplest embedding would be a one hot encoding of text data where each vector would be mapped to a category.

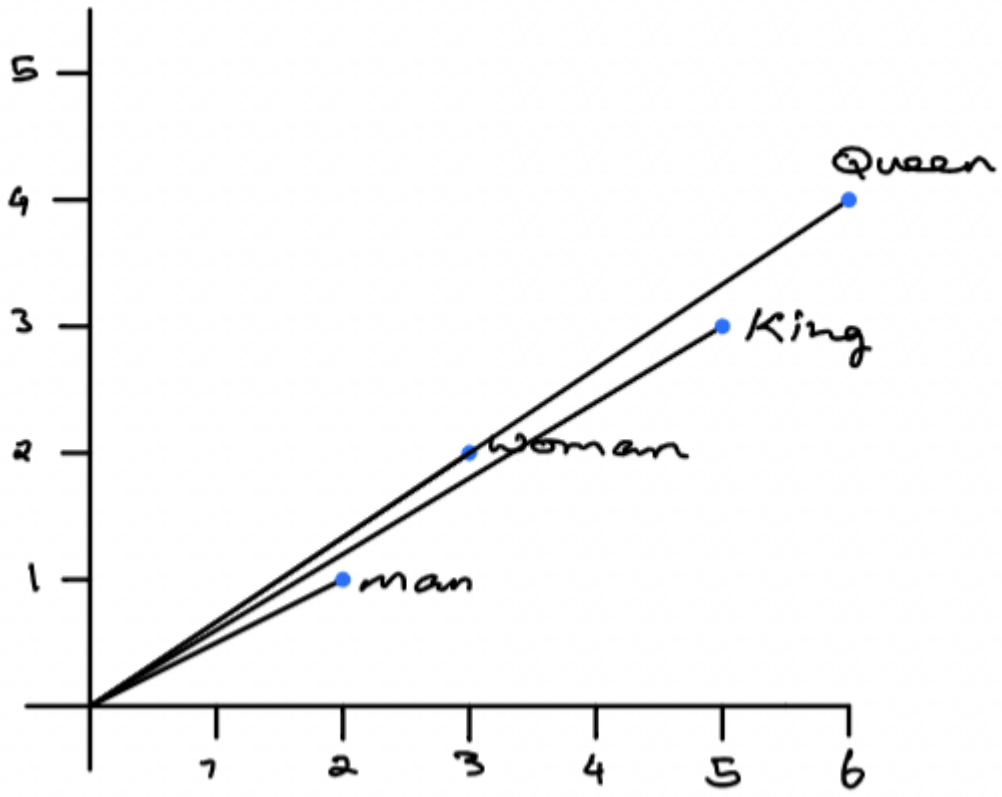
```
For example:

have = [1, 0, 0, 0, 0, 0, ... 0]
a    = [0, 1, 0, 0, 0, 0, ... 0]
good = [0, 0, 1, 0, 0, 0, ... 0]
day  = [0, 0, 0, 1, 0, 0, ... 0] ...
```

However, there are multiple limitations of simple embeddings such as this, as they do not capture characteristics of the word, and they can be quite large depending on the size of the corpus.

Word2Vec Architecture

The effectiveness of Word2Vec comes from its ability to group together vectors of similar words. Given a large enough dataset, Word2Vec can make strong estimates about a word's meaning based on their occurrences in the text. These estimates yield word associations with other words in the corpus. For example, words like "King" and "Queen" would be very similar to one another. When conducting algebraic operations on word embeddings you can find a close approximation of word similarities. For example, the 2 dimensional

```
King    -    Man    +    Woman    =    Queen
[5,3]   -    [2,1]  +    [3, 2]   =    [6,4]
```

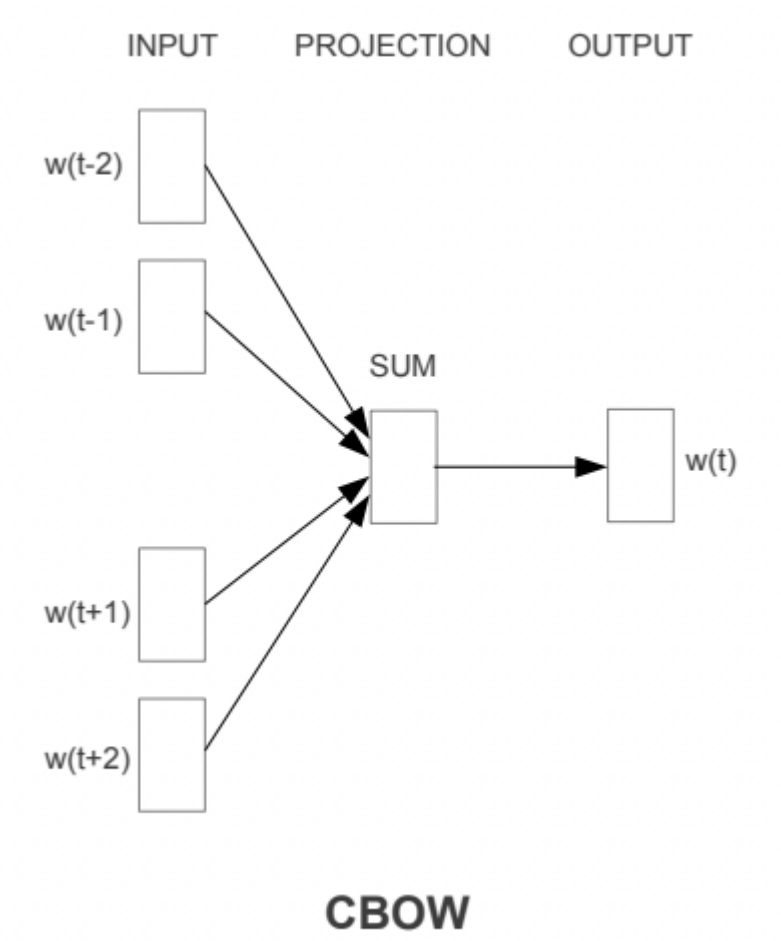


You can see that the words King and Queen are close to each other in position. (Image provided by the author)

There are two main architectures which yield the success of word2vec. The skip-gram and CBOW architectures.

CBOW (Continuous Bag of Words)

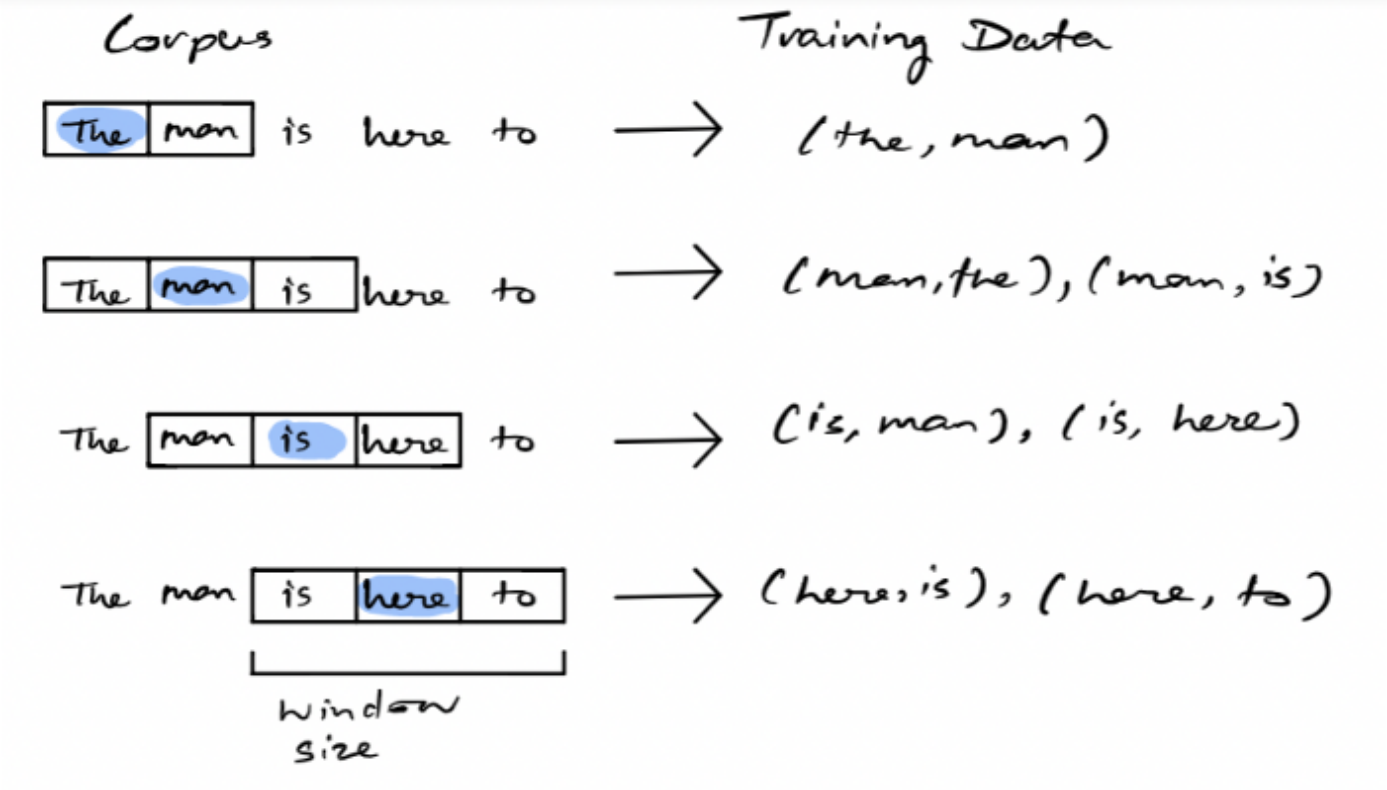
This architecture is very similar to a feed forward neural network. This model architecture essentially tries to predict a target word from a list of context words. The intuition behind this model is quite simple: given a phrase `"Have a great day"` , we will choose our target word to be “a” and our context words to be [“have”, “great”, “day”]. What this model will do is take the distributed representations of the context words to try and predict the target



CBOW Architecture. Image taken from Efficient Estimation of Word Representation in Vector Space

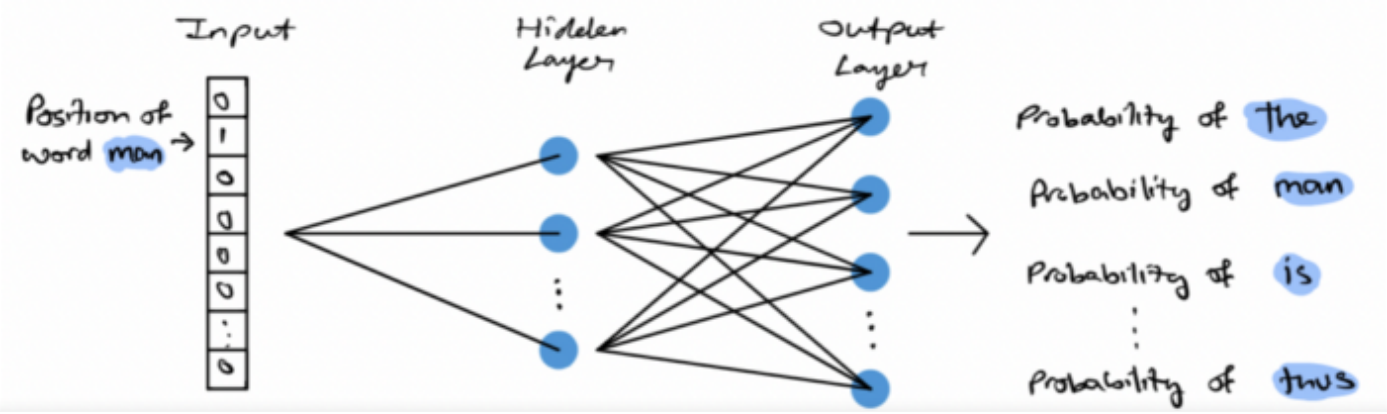
Continuous Skip-Gram Model

The skip-gram model is a simple neural network with one hidden layer trained in order to predict the probability of a given word being present when an input word is present. Intuitively, you can imagine the skip-gram model being the opposite of the CBOW model. In this architecture, it takes the current word as an input and tries to accurately predict the words before and after this current word. This model essentially tries to learn and predict the context words around the specified input word. Based on experiments assessing the accuracy of this model it was found that the prediction quality improves given a large range of word vectors, however it also increases the computational complexity. The process can be described visually as seen below.



Example of generating training data for skip-gram model. Window size is 3. Image provided by author

As seen above, given some corpus of text, a target word is selected over some rolling window. The training data consists of pairwise combinations of that target word and all other words in the window. This is the resulting training data for the neural network. Once the model is trained, we can essentially yield a probability of a word being a context word for a given target. The following image below represents the architecture of the neural network for the skip-gram model.



A corpus can be represented as a vector of size N, where each element in N corresponds to a word in the corpus. During the training process, we have a pair of target and context words, the input array will have 0 in all elements except for the target word. The target word will be equal to 1. The hidden layer will learn the embedding representation of each word, yielding a d-dimensional embedding space. The output layer is a dense layer with a softmax activation function. The output layer will essentially yield a vector of the same size as the input, each element in the vector will consist of a probability. This probability indicates the similarity between the target word and the associated word in the corpus.

For a more detailed overview of both these models, I highly recommend reading the original paper which outlined these results here.

Implementation

I'll be showing how to use word2vec to generate word embeddings and use those embeddings for finding similar words and visualization of embeddings through PCA.

Data

For the purposes of this tutorial we'll be working with the Shakespeare dataset. You can find the file I used for this tutorial here, it includes all the lines Shakespeare has written for his plays.

Requirements

```
nltk==3.6.1
node2vec==0.4.3
pandas==1.2.4
matplotlib==3.3.4
gensim==4.0.1
scikit-learn=0.24.1
```

Note: Since we're working with NLTK you might need to download the following corpus for the rest of the tutorial to work. This can easily be done by the following commands :


Import Data

```
import pandas as pd
import nltk
import string
import matplotlib.pyplot as plt

from nltk.corpus import stopwords
from nltk import word_tokenize
from gensim.models import Word2Vec as w2v
from sklearn.decomposition import PCA

# constants
PATH = 'data/shakespeare.txt'
sw = stopwords.words('english')
plt.style.use('ggplot')
# nltk.download('punkt')
# nltk.download('stopwords')

# import data
lines = []
with open(PATH, 'r') as f:
    for l in f:
        lines.append(l)
```

w2v_import.py hosted with ❤ by GitHub view raw

Note: Change the `PATH` variable to the path of the data you're working with.

Preprocess Data

```
# remove new lines
lines = [line.rstrip('\n') for line in lines]

# make all characters lower
lines = [line.lower() for line in lines]
```

```
lines = [word_tokenize(line) for line in lines]

def remove_stopwords(lines, sw = sw):
    '''
    The purpose of this function is to remove stopwords from a given array of
    lines.

    params:
        lines (Array / List) : The list of lines you want to remove the stopwords from
        sw (Set) : The set of stopwords you want to remove

    example:
        lines = remove_stopwords(lines = lines, sw = sw)
    '''

    res = []
    for line in lines:
        original = line
        line = [w for w in line if w not in sw]
        if len(line) < 1:
            line = original
        res.append(line)
    return res

filtered_lines = remove_stopwords(lines = lines, sw = sw)
```

w2v_preprocess.py hosted with ❤ by GitHub view raw

Stopword Filtering Note

- Be aware that the stopwords removed from these lines are of modern vocabulary. The application & data has a high importance to the type of preprocessing tactics necessary for cleaning of words.
- In our scenario, words like “you” or “yourself” would be present in the stopwords and eliminated from the lines, however since this is Shakespeare text data, these types of words would not be used. Instead “thou” or “thyself” might be useful to remove. Stay keen to these types of miniature changes because they make a drastic difference in the

Embed

```
w = w2v(
    filtered_lines,
    min_count=3,
    sg = 1,
    window=7
)

print(w.wv.most_similar('thou'))

emb_df = (
    pd.DataFrame(
        [w.wv.get_vector(str(n)) for n in w.wv.key_to_index],
        index = w.wv.key_to_index
    )
)
print(emb_df.shape)
emb_df.head()
```

w2v_embedd.py hosted with ❤ by GitHub view raw

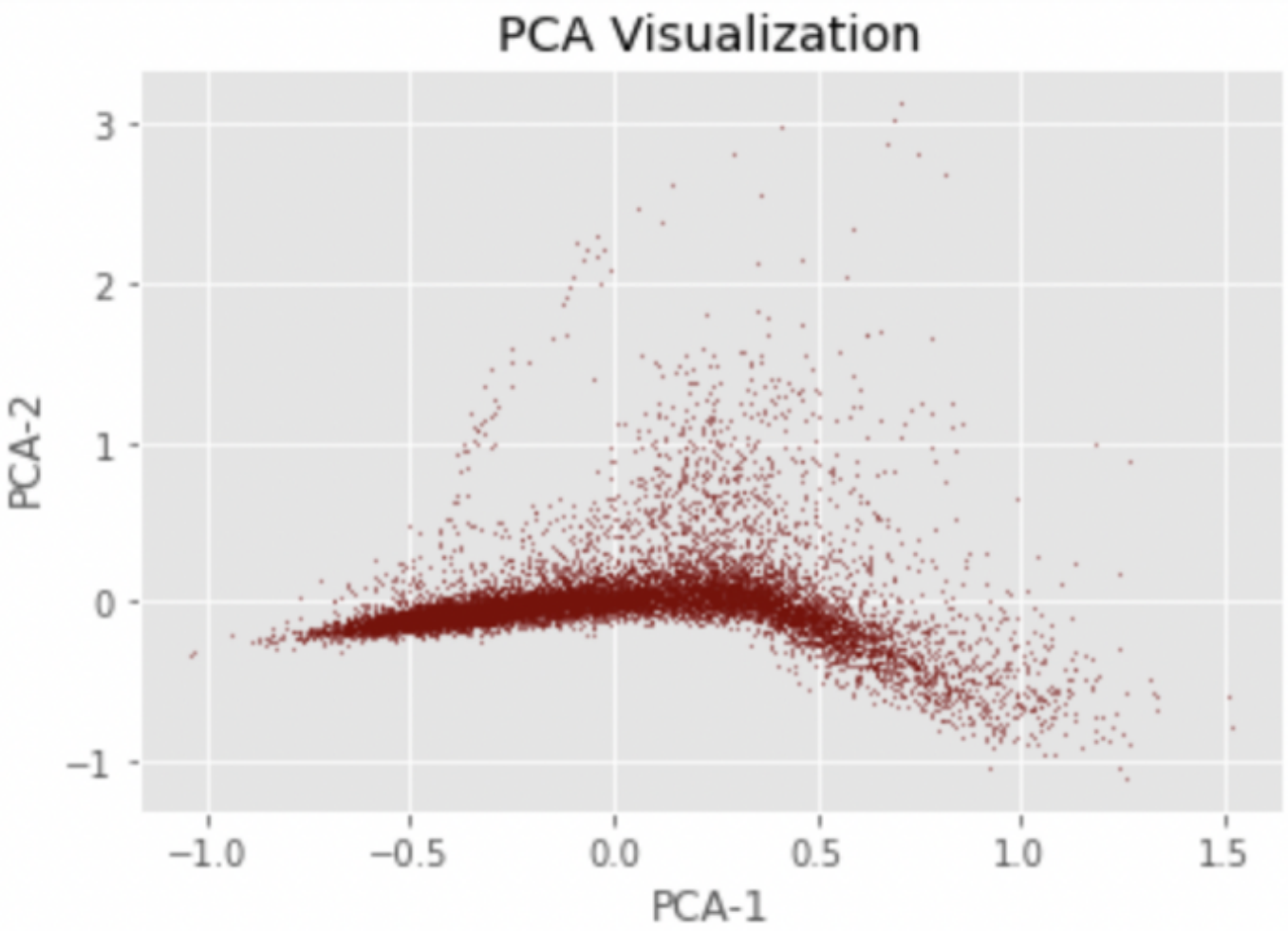
```
w.wv.most_similar('thou')
```

```
[('thyself', 0.7645957469940186),
 ('art', 0.707123875617981),
 ('dost', 0.6897317171096802),
 ('wouldst', 0.6642480492591858),
 ('wilt', 0.6552817225456238),
 ('villain', 0.651666522026062),
 ('didst', 0.6509525775909424),
 ('shalt', 0.6448981165885925),
```

PCA on Embeddings

```
pca = PCA(n_components=2, random_state=7)
pca_mdl = pca.fit_transform(emb_df)

emb_df_PCA = (
    pd.DataFrame(
        pca_mdl,
        columns=['x','y'],
        index = emb_df.index
    )
)

plt.clf()
fig = plt.figure(figsize=(6,4))

plt.scatter(
    x = emb_df_PCA['x'],
    y = emb_df_PCA['y'],
    s = 0.4,
    color = 'maroon',
    alpha = 0.5
)

plt.xlabel('PCA-1')
plt.ylabel('PCA-2')
plt.title('PCA Visualization')
plt.plot()
```

w2v_pca.py hosted with ❤ by GitHub view raw

Words similar to each other would be placed closer together to one another. Image provided by author

Tensorflow has made a very beautiful, intuitive and user-friendly representation of the word2vec model. I highly recommend you to explore it as it allows you to interact with the results of word2vec. The link is below.

Embedding projector - visualization of high-dimensional data

Visualize high dimensional data.

projector.tensorflow.org

produces an embedding vector associated with each word in the corpus. These embeddings are structured such that words with similar characteristics are in close proximity to one another. CBOW (continuous bag of words) and the skip-gram model are the two main architectures associated with word2vec. Given an input word, skip-gram will try to predict the words in context to the input whereas the CBOW model will take a variety of words and try to predict the missing one.

I've also written about node2vec which uses word2vec to generate node embeddings given a network. You can read about it here.

Node2Vec Explained

Explaining & Implementing the Node2Vec Paper in Python

towardsdatascience.com

Resources

- https://arxiv.org/pdf/1301.3781.pdf
- https://www.kdnuggets.com/2019/02/word-embeddings-nlp-applications.html
- https://wiki.pathmind.com/word2vec
- https://projector.tensorflow.org/

If you enjoyed reading this article, please consider following me for upcoming articles explaining other data science materials and those materials (like word2vec) to solve relevant problems in different areas of data science. Here are some other articles I've written which I think you might enjoy.