Word2vec word embedding model using Gensim

Mrinal Chaudhari, Colin Gladden

October 27, 2021

1 Summary

1.1 What is word embeddings?

Word embeddings is one of the important techniques in natural language processing for mapping words in vectors of real words. Word embeddings are useful to capture meaning of words in the document , similarity between words, relationship between other words. Word embeddings are often used for recommender systems and text classifications.

1.2 What is Word2vec?

Word2vec is the most popular technique to learn word embeddings using a two-layer neural network. Its input is a text corpus and its output is a set of vectors. Word embedding via word2vec can make natural language computer-readable, then further implementation of mathematical operations on words can be used to detect their similarities. A well-trained set of word vectors will place similar words close to each other in that space. For instance, the words cat, dog, and tiger might cluster in one corner, while yellow, red and blue cluster together in another. There are mainly two training algorithms for word2vec, one is continuous bag of words(CBOW) and another is skip-gram. The difference between both algorithms is that CBOW is context to predict target word while skip-gram is uses a word to predict a target context. The skip-gram method outperformed compared to CBOW because it captured two semantics for single words. Skip-gram is one of the unsupervised learning techniques used to find the most related words for a given word[1].

1.3 Architecture of word2vec

The architecture of the skip-gram model is similar to the continuous bag of word model, however instead of predicting the current word, it tries to maximize the classification of words based on another word in the same corpus. We take each current as an input to a log linear classifier with continuous projection and predict the words within a certain range before and after the current word. It has been seen that increasing the range for words would increase the quality of resulting word vectors and we can get more similarity scores between the two given words. Also, the computational complexity may increase if we increase the range of a given word. While more distant words are less related to the current word than those close to it, we assigned less weight to distant words.

The training complexity of this architecture is proportional to,
Q=C*(D+D*log2(V))
Where,
C is the maximum distance of the words,
The vocabulary of unique words present in our dataset is V

[1]https://towardsdatascience.com/a-beginners-guide-to-word-embedding-with-gensim-word2vec-model-5970fa56cc92

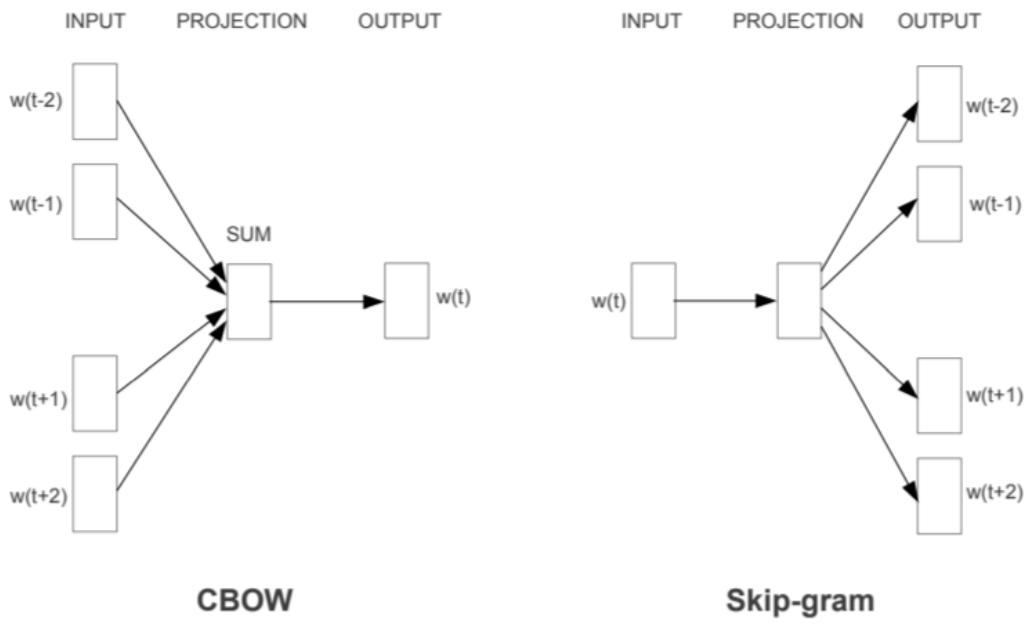

Figure 1: Architecture

As we can see in "Fig. 1" the skip-gram architecture diagram[2], w(t) is the target word and input word. There is one hidden layer which performs the dot product of weight matrix and input word. The dot product of the hidden layer is passed to the output layer. The output layer computes the dot product between the output vector of the hidden layer and weight matrix of the output layer. The softmax activation function has been applied to compute the probability of the words in the context of w(t).

N= number of neurons present in the hidden layer.
The output vector of the hidden layer is H[N]

Working Steps:

- The words are converted into a vector.
- The word w(t) is passed to the hidden layer from |V| neuron
- The Hidden layer performs the dot product between weight vector W[|v|, N] and the input vector w(t).
- Output layer will apply dot product between H[1, N] and W'[N, |v|] and will give the vector U. Here hidden layer H[1, N] directly passes to the output function as activation function which is not used in the hidden layer.
- Softmax function was used to find the probability of each vector and each iteration gives output vector U.
- The word with highest probability is the result

This step will be executed for each word w(t) present in the corpus and vocabulary. And each word w(t) will be passed k times.

[2]https://towardsdatascience.com/skip-gram-nlp-context-words-prediction-algorithm-5bbf34f84e0c

1.4 Dataset

I am using a large set of cohesive texts from the Gutenberg website[3]. This dataset contains around 26k number of sentences and 674k number of words. To get good results for the word2vec model such types of texts are needed

We are also doing the pre-processing on the text using nltk.corpus Stop words. This does some basic preprocessing such as removal of stopwords, punctuation's, and conversion of lowercase. After preprocessing, we have converted each sentence into tokens and to check if the words are present in the text, we printed the first 1000 words.

1.5 Training the word2vec model:

We instantiated the word2vec model using the gensim library and passed the corpus (data). At first, we need to generate a format of 'list of lists' for training the make model word embedding. Corpus(dataset) contains the set of tokens from the Gutenberg text file. Word2Vec uses all these tokens to internally create a vocabulary i.e set of unique words. After building the vocabulary, we need to call a train to start training the Word2Vec model. Behind the scenes we are actually training a simple neural network with a single hidden layer. Ultimate goal is to learn the weight of the hidden layer and not to use a neural network after training. These weights are the word vector that we wanted to implement.

```
model= gensim.models.Word2Vec(corpus, window=5, size=150, min_count=5, workers=10)
model.train(corpus,total_examples=len(corpus),epochs=10)
print("done")
model.save("myw2v.model")
model=Word2Vec.load("myw2v.model")
model.wv['shoes']
```

Size: The number of dimension of the embeddings and the default is 100
Window: The maximum distance between a target word and words around the target word. The default window is 5.
Min_count: The minimum count of words to consider when training the model. The default min_count is 5
workers: The number of partitions during training and the default workers is 3.
sg: The training algorithm, either CBOW(0) or skip gram(1). The default training algorithm is CBOW.

Let's look at the some of outputs:
1: For inspecting vector
we are getting below output "Fig. 2":
Model.wv['shoes']
2.finding similarity between-anologies between words
This example shows "Fig. 3" the simple case of looking up similarity between two words. All we need to do is to call similarity functions and provide two words and this will return the similarity score between two pairs.

```
model.wv.similarity('shoes', 'star')
```

In this example the similarity between shoes and star is 0.76 which is significantly less than the similarity between terrible and terrible.

The above snippets "Fig. 4" computes the cosine similarity between two specified words using word vectors of each. From the scores it makes sense that "beautiful" is highly similar to "attentive" but "old" is not similar to "young". The similarity score between two similar words is 1 as the range of the cosine similarity

[3]https://archive.org/details/gutenberg

```
In [32]: model.wv['shoes']

Out[32]: array([ 0.4231349 ,  0.24150918,  0.38285658,  0.06408017, -0.17142354,
                -0.04129152,  0.3853281 , -0.14677821, -0.26670447, -0.22693107,
                -0.50594753, -0.33232173, -0.16423033,  0.00350111, -0.01708927,
                -0.10291281, -0.09570538, -0.34596434,  0.16891117, -0.1446808 ,
                -0.03939368,  0.45344812,  0.73229504, -0.35361183,  0.12062233,
                -0.31858477, -0.05579877,  0.00765155, -0.07230373,  0.06586425,
                 0.12120774,  0.40344542, -0.24305697,  0.15529591, -0.230878  ,
                -0.09555674, -0.6350937 , -0.9522375 , -0.0336275 , -0.4497442 ,
                -0.01075968, -0.04038865, -0.02028355, -0.5081734 ,  0.16163445,
                 0.10244984, -0.5159051 , -0.22231674,  0.03271761,  0.241761  ,
                -0.0466414 , -0.14471164, -0.46766496,  0.40970576,  0.07306921,
                -0.20057037,  0.5483451 ,  0.16856162, -0.08717429,  0.21793388,
                 0.0503979 , -0.22151564, -0.3858898 , -0.3786885 ,  0.00628941,
                -0.42726997, -0.35701966, -0.5073404 , -0.25980356,  0.23788051,
                -0.3876806 , -0.20322098,  0.49150562, -0.32926744,  0.08243912,
                 0.0285237 ,  0.44471934, -0.63565534,  0.05004558,  0.25124463,
                 0.27193248, -0.22239411, -0.47342902,  0.16916156, -0.50688064,
                -0.3128145 , -0.15879601,  0.4222654 ,  0.25136566,  0.58132714,
                 0.01839506,  0.41276222, -0.24576394,  0.25471917,  0.336561  ,
                -0.10257746, -0.00727359, -0.23527986,  0.456522  ,  0.22479267],
               dtype=float32)
```

Figure 2: Vector

```
In [33]: model.wv.similarity('shoes', 'star')

Out[33]: 0.76202315
```

```
In [34]: model.wv.similarity('terrible', 'terrible')

Out[34]: 1.0
```

Figure 3: Similarity pair

score will always be between [0.0-1.0].

3. To find the most similar words:

w1=”beautiful”
model.wv.most_similar (positive=w1)

This example shows “Fig. 5” the simple case of looking up words similar to the word beautiful. All we need to do is to call most_similar functions and word beautiful as the positive word and this will return the 10 similar words to beautiful.

Similarly, the other examples “Fig. 6” return the most similar pair of given word.

To find out the most similar word to happy but not related to sad “Fig. 7”.

```
# similarity between two different words
model.wv.similarity(w1="beautiful",w2="attentive")
```

```
0.7461889
```

```
# similarity between two identical words
model.wv.similarity(w1="dirty",w2="dirty")
```

```
1.0
```

```
# similarity between two unrelated words
model.wv.similarity(w1="old",w2="young")
```

```
0.43908346
```

Figure 4: Similarity pair

4: Checking for Potential bias: In the example "Fig. 8", there is a potential bias towards women in the textbook. There are more women nurses than doctors. Also the book talks more about Italian women than European women. This could be because the number of times the word Italian is used for women's is greater than the European women.

1.6 Division of work

problem solving: Mrinal and colin
coding: Mrinal
Writing: Mrinal and Colin

```
n [44]: w1="beautiful"
        model.wv.most_similar (positive=w1)

Out[44]: [('luminous', 0.7729548811912537),
          ('thoughtful', 0.7637312412261963),
          ('cast', 0.7577607035636902),
          ('pallor', 0.7529593110084534),
          ('attentive', 0.7461888790130615),
          ('radiant', 0.7417609095573425),
          ('slim', 0.7407193183898926),
          ('shining', 0.7400855422019958),
          ('remarkably', 0.7395445704460144),
          ('emaciated', 0.7385475039482117)]

n [36]: # look up top 6 words similar to 'angry'
        w1 = ["angry"]
        model.wv.most_similar (positive=w1,topn=6)

Out[36]: [('tearful', 0.7470288872718811),
          ('abruptly', 0.7208036780357361),
          ('deliberate', 0.7061899304389954),
          ('hoarse', 0.7042058706283569),
          ('breathless', 0.7019003033638),
          ('imploring', 0.700491726398468)]

n [37]: # look up top 6 words similar to 'smile'
        w1 = ["smile"]
        model.wv.most_similar (positive=w1,topn=6)

Out[37]: [('sarcastic', 0.6681339144706726),
          ('ironic', 0.663745105266571),
          ('smiles', 0.6603155732154846),
          ('childlike', 0.6573683619499207),
          ('subtle', 0.6511207818984985),
          ('amiable', 0.639979362487793)]
```

Figure 5: Most Similarity pair

```
In [46]:  # look up top 6 words similar to 'france'
          w1 = ["france"]
          model.wv.most_similar (positive=w1,topn=6)
```

```
Out[46]: [('napoleonic', 0.910703718662262),
          ('italy', 0.8761806488037109),
          ('crimes', 0.874067485332489),
          ('secure', 0.8735527396202087),
          ('rulers', 0.864109992980957),
          ('rhine', 0.854250967502594)]
```

```
In [48]:  # look up top 6 words similar to 'polite'
          w1 = ["polite"]
          model.wv.most_similar (positive=w1,topn=6)
```

```
Out[48]: [('characteristic', 0.885038435459137),
          ('tact', 0.8442219495773315),
          ('views', 0.8408305644989014),
          ('m', 0.8375087976455688),
          ('favors', 0.8345000743865967),
          ('amiably', 0.8327236175537109)]
```

Figure 6: Most Similarity pair

```
# get everything related to stuff on the happy
w1 = ["pleasant",'happy','smile']
w2 = ['sad']
model.wv.most_similar (positive=w1,negative=w2,topn=10)
```

```
[('absentminded', 0.8084676861763),
 ('smiles', 0.8060325384140015),
 ('calm', 0.7908859252929688),
 ('careless', 0.7889071106910706),
 ('bewildered', 0.7869927287101746),
 ('childlike', 0.7859752774238586),
 ('pathetic', 0.7848834991455078),
 ('stupid', 0.7845149040222168),
 ('irony', 0.7779282331466675),
 ('tender', 0.7772961854934692)]
```

Figure 7: Most Similarity pair

```
In [63]: # similarity between two unrelated words
         model.wv.similarity(w1="nurse",w2="women")

Out[63]: 0.42871758

In [66]: # similarity between two unrelated words
         model.wv.similarity(w1="doctor",w2="women")

Out[66]: 0.33425188

In [67]: # similarity between two unrelated words
         model.wv.similarity(w1="doctor",w2="man")

Out[67]: 0.20945829

In [68]: # similarity between two unrelated words
         model.wv.similarity(w1="nurse",w2="women")

Out[68]: 0.42871758

In [69]: # similarity between two unrelated words
         model.wv.similarity(w1="nurse",w2="man")

Out[69]: 0.2413639

In [60]: # similarity between two unrelated words
         model.wv.similarity(w1="europe",w2="women")

Out[60]: 0.17368746

In [59]: # similarity between two unrelated words
         model.wv.similarity(w1="italian",w2="women")

Out[59]: 0.5244944
```

Figure 8: Most Similarity pair