Adem Akdogan
Jul 22, 2021 · 10 min read · Listen

Word Embedding Techniques: Word2Vec and TF-IDF Explained

The words need to be made meaningful for machine learning or deep learning algorithms. Therefore, they must be expressed numerically. Algorithms such as One Hot Encoding, TF-IDF, Word2Vec, FastText enable words to be expressed mathematically as word embedding techniques used to solve such problems.

Photo by Camille Orgel on Unsplash

Word Embeddings

The word embedding techniques are used to represent words mathematically. One Hot Encoding, TF-IDF, Word2Vec, FastText are frequently used Word Embedding methods. One of these techniques (in some cases several) is preferred and used according to the status, size and purpose of processing the data.

* One Hot Encoding

One of the most basic techniques used to represent data numerically is One Hot Encoding technique[1]. In this method, a vector is created in the size of the total number of unique words. The value of vectors is assigned such that the value of each word belonging to its index is 1 and the others are 0. As an example, Figure 1 can be examined.

X	MacOS	Linux	Windows
MacOS	1	0	0
Windows	0	0	1
MacOS	1	0	0
Linux	0	1	0
Windows	0	0	1

Figure 1. Sample of One Hot Encoding

The column named "X" consists of 3 different words in Figure 1. When One Hot Encoding is applied to this column, 3 different columns representing each expression are created (in other words, 3-unit vectors are created for each row). The column corresponding to the word in each row is filled with the values 1, the others 0. Thus, these expressions are digitized. It is generally used in situations where there is not much verbal data diversity and there is no need to represent the semantic and statistical relationships between the data.

* TF-IDF

TF-IDF is a statistical measure used to determine the mathematical significance of words in documents[2]. The vectorization process is similar to One Hot Encoding. Alternatively, the value corresponding to the word is assigned a TF-IDF value instead of 1. The TF-IDF value is obtained by multiplying the TF and IDF values. As an example, let's find the TF-IDF values for 3 documents consisting of 1 sentence.

> *[He is Walter],*
>
> *[He is William],*
>
> *[He isn't Peter or September]*

In the above example, "He" is used in all 3 documents, "is" is in 2 documents, and

In the above example, "He" is used in all 3 documents, "is" is in 2 documents, and "or" is in only one document. According to these, let's find the TF and then the IDF values, respectively.

- *TF (Term Frequency)*

In the simplest terms, term frequency is the ratio of the number of target terms in the document to the total number of terms in the document. If TF values are calculated according to the above example, it will be

[0.33, 0.33, 0.33],

[0.33, 0.33, 0.33],

[0.20, 0.20, 0.20, 0.20, 0.20]

- *IDF (*Inverse Documet Frequency*)*

The IDF value is the logarithm of the ratio of the total number of documents to the number of documents in which the target term occurs. At this stage, it does not matter how many times the term appears in the document. It is sufficient to determine whether it has passed or not. In this example, the base value of the logarithm to be taken is determined as 10. However, there is no problem in using different values.

"He": Log(3/3)= 0,

"is": Log(3/2):0.1761,

"or, Peter, ..": log(3/1) : 0.4771

Thus, both TF and IDF values were obtained. If vectorization is created with these values, firstly a vector consisting of elements equal to the number of unique words in all documents is created for each document (in this example, there are 8 terms). At this stage, there is a problem to be solved. As seen in the term "He", since the IDF value is 0, the TF-IDF value will also be zero. However, words that are not included in the document during the vectorization process (for example, the phrase "Peter" is not included in the 1st sentence) will be assigned a value of 0. In order to avoid confusion here, TF-IDF values are smoothed for vectorization. The most common method is to add 1 to the obtained values. Depending on the purpose, normalization can be applied to these values later. If the vectorization process is created according to the above-mentioned;

[1. , 1.1761 , 1.4771 , 0. , 0. , 0. , 0. , 0.],

[1. , 1.1761 , 0. , 1.4771 , 0. , 0. , 0. , 0.],

[1. , 0. , 0. , 0. , 1.4771 , 1.4771, 1.4771 , 1.4771],

*** Word2Vec**

Word2vec is another of the frequently used word embedding techniques. The entire corpus is scanned, and the vector creation process is performed by determining which words the target word occurs with more often[3]. In this way, the semantic closeness of the words to each other is also revealed. For example, let each letter in the sequences ..x y A z w.. , .. x y B z k.. and ..x l C d m… represent a word. In this case, word_A will be closer to word_B than word_C. When this situation is taken into account in vector formation, the semantic closeness between words is expressed mathematically.

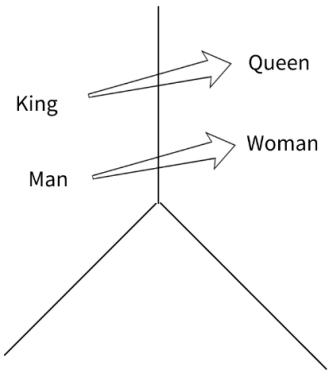

Figure 2. Word Similarity

Figure 2 shows one of the most frequently used images in Word2Vec. The semantic closeness between these words is the mathematical closeness of the vector values to each other. One of the frequently given examples is the equation “king-man + woman = queen”. What happens here is that the vector value obtained as a result of subtracting and adding the vectors from each other is equal to the vector corresponding to the “queen” expression. It can be understood that the words “king” and “queen” are very similar to each other, but vectorial differences arise only because of their gender.

In Word2Vec method, unlike One Hot Encoding and TF-IDF methods, unsupervised learning process is performed. Unlabeled data is trained via artificial neural networks to create the Word2Vec model that generates word vectors. Unlike other methods, the vector size is not as much as the number of unique words in the corpus. The size of the vector can be selected according to the corpus size and the type of project. This is particularly beneficial for very large data. For example, if we assume that there are 300 000 unique words in a large corpus, when vector creation is performed with One Hot Encoding, a vector of 300 000 size is created for each word, with the value of only one element of 1 and the others 0. However, by choosing the vector size 300 (it can be more or less depending on the user’s choice) on the Word2Vec side, unnecessary large size vector operations are avoided.

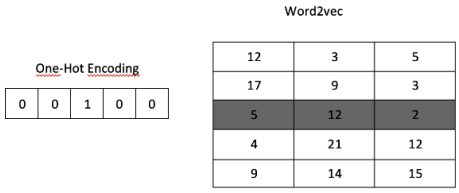

Figure 3. Vectorization of word in Word2Vec model

The vectorization of the “Royal” can be seen in Figure 3. If the word of “Royal” in a five-word sentence is vectorized with One Hot Encoding, the first vector expression (input Vector) is obtained. As can be seen, the size of this vector is as much as the total number of words in the sentence. However, if the vectorization process is done with Word2Vec, this time a vector of three units [5,12,2] is created.

Hotel reviews dataset in kaggle (https://www.kaggle.com/anu0012/hotel-review) was used for applied Word2Vec model training. All of the codes given as examples can be found here.

```python
import pandas as pd
import re
from nltk.corpus import stopwords
from gensim.models import Word2Vec
import multiprocessing


csv_path = ".../hotel-reviews.csv"

df = pd.read_csv(csv_path)
def clean_data(text):
    text = re.sub(r'[^ \nA-Za-z0-9À-Ö0-öø-ÿ/]+', '', text)
    text = re.sub(r'[\\/*\^\]\[+]', '', text)
    return text
def change_lower(text):
    text = text.lower()
    return text

stopwords_list = stopwords.words("english")
def remover(text):
    text_tokens = text.split(" ")
    final_list = [word for word in text_tokens if not word in stopwords_list]
    text = ' '.join(final_list)
    return text

def get_w2vdf(df):
    w2v_df = pd.DataFrame(df["sentences"]).values.tolist()
    for i in range(len(w2v_df)):
        w2v_df[i] = w2v_df[i][0].split(" ")
    return w2v_df

def train_w2v(w2v_df):
    cores = multiprocessing.cpu_count()
    w2v_model = Word2Vec(min_count=4,
                         window=4,
                         size=300,
                         alpha=0.03,
                         min_alpha=0.0007,
                         sg = 1,
                         workers=cores-1)

    w2v_model.build_vocab(w2v_df, progress_per=10000)
    w2v_model.train(w2v_df, total_examples=w2v_model.corpus_count, epochs=100, report_delay=1)
    return w2v_model

df[["sentences"]] = df[["sentences"]].astype(str)
df["sentences"] = df["sentences"].apply(change_lower)
df["sentences"] = df["sentences"].apply(clean_data)
df["sentences"] = df["sentences"].apply(remover)
```

```
50
51  w2v_df = get_w2vdf(df)
52  w2v_model = train_w2v(w2v_df)
```

medium_w2v.py hosted with ❤ by GitHub view raw

Since it is case sensitive, all words are converted to lowercase. Then, special characters and stopwords are cleared. The nltk library was used for ineffective words. If desired, these words can also be determined completely manually. An example sentence before these operations are performed is as follows.

"My husband and I have stayed in this hotel a few times. Though not the fanciest hotel, we love the fact that we can walk the — or so miles to Fenway. It is clean and the staff is very accomodating. My only complaint is that the fan in the bathroom was noisy and went on automatically when you turned the light on and we tried to keep the light off as much as possible. We've stayed in pricier hotels which charged for internet and breakfast and these are both included. Will stay there again."

The new situation that will occur after the preprocessing of the data is as follows.

"husband stayed hotel times though fanciest hotel love fact walk miles fenway clean staff accomodating complaint fan bathroom noisy went automatically turned light tried keep light much possible weve stayed pricier hotels charged internet breakfast included stay"

After these processes are completed, Word2Vec training is performed. Parameters used during training:

min_count : The minimum number of occurrences of the target word in the corpus. Especially for very large copruses, keeping this limit high increases the success even more. However, it would be more accurate to keep it small for small datasets.

window : It is the number of neighboring words that will directly affect the vector calculations of the target expression. For example, "He is a very good person." For window $=1$, the words "a" and "good" are effective in the formation of the "very" word vector. When window $= 2$, the words "is","a","good" and "person" are effective in creating the "very" word vector.

size : It is the size of the vector to be created for each element.

alpha : Initial learning rate

min_alpha : It is the minimum value at which the learning rate will decrease linearly during training.

sg : Specifies how the training algorithm will work. If value of sg is 1, skip-gram algorithm is used, otherwise the CBOW algorithm is used.

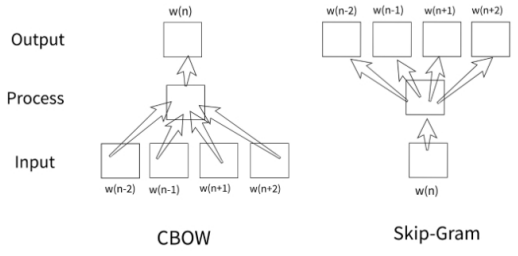

Figure 4. Skip-gram vs CBOW

The difference between CBOW (Continuous Bag of Words) vs Skip-gram algorithms can be seen in Figure 4. In the trainings in which the CBOW algorithm is used, the words adjacent to the target word are given as input and the target word itself is obtained as output. In the skip-gram algorithm, while the target word itself is given as input, neighboring words are obtained as output.

workers : Traning can be performed in parallel. The number of cores to be used for this can be determined with the workers parameter.

If you want to see the vector of the word "great" by using the model obtained as a result of the training;

```
w2v_model["great"]
>>>array([ 3.03658217e-01, -1.56424701e-01, -8.23674500e-01,
.
.
.

-1.36196673e-01,  8.55127215e-01, -7.31807232e-01,  1.36362463e-01],
      dtype=float32)
```

```
print(w2v_model["great"].shape)
>>>(300,)
```

The 10 words closest to “great”, “terrible”, “boston”, “valet” are as follows.

```
w2v_model.wv.most_similar(positive=["great"])
>>>[('excellent', 0.8094755411148071),
 ('fantastic', 0.7735758423805237),
 ('perfect', 0.7473931312561035),
 ('wonderful', 0.7063912153244019),
 ('good', 0.7039040327072144),
 ('amazing', 0.6384587287902832),
 ('loved', 0.6266685724258423),
 ('nice', 0.6253951787948608),
 ('awesome', 0.6186609268188477),
 ('enjoyed', 0.5889394283294678)]
---------------------------------
w2v_model.wv.most_similar(positive=["terrible"])
>>>[('bad', 0.5275813341140747),
 ('poor', 0.5044319629669189),
 ('horrible', 0.4722219705581665),
 ('awful', 0.42389577627182007),
 ('worst', 0.40153956413269043),
 ('dirty', 0.3467090427875519),
 ('disgusting', 0.32588857412338257),
 ('horrendous', 0.3157917261123657),
 ('lousy', 0.30114778876304626),
 ('uncomfortable', 0.3005620837211609)]
---------------------------------
w2v_model.wv.most_similar(positive=["boston"])
>>>[('chicago', 0.5191804170608521),
 ('seattle', 0.5126588940620422),
 ('dc', 0.4830571711063385),
 ('bostons', 0.4459514617919922),
 ('copley', 0.4455355107784271),
 ('city', 0.44090309739112854),
 ('newbury', 0.4349810481071472),
 ('fenway', 0.4237935543060303),
 ('philadelphia', 0.40892332792282104),
 ('denver', 0.3984081149101257)]
---------------------------------
w2v_model.wv.most_similar(positive=["valet"])
>>>[('parking', 0.7374086380004883),
 ('car', 0.6263512969017029),
 ('garage', 0.6224508285522461),
 ('retrieving', 0.5173929929733276),
 ('self', 0.5013973712921143),
 ('inandout', 0.4847780168056488),
 ('selfpark', 0.47603434324264526),
 ('fee', 0.47458043694496155),
 ('per', 0.4741314947605133),
 ('parked', 0.4707031846046448)]
```

*** FastText**

The working logic of FastText algorithm is similar to Word2Vec, but the biggest difference is that it also uses N-grams of words during training [4]. While this increases the size and processing time of the model, it also gives the model the ability to predict different variations of words. For example, let’s say that the word “Windows” is in the training dataset and we want to get the vector of the word “Wndows” after the training is finished. If the Word2Vec model is used for these operations, it will give an error that the word “Wndows” does not exist in the dictionary and will not return any vectors. However, if the FastText model is used for this process, both the vector will return and the word of “Windows” will be among the closest words. As explained above, not only the word itself but also N-gram variations are included in training (Example 3-gram expressions for the word “Windows” -> Win, ind, ndo, dow, ows). Although the FastText model is used in many different areas today, it is frequently preferred especially when word embedding techniques are needed in OCR works. Especially compared to other techniques that do not tolerate the slightest OCR error, FastText provides a great advantage in obtaining vectors of even words that are not directly in its own vocabulary. For this reason, it is one step ahead of other alternatives in problems where word mistakes may occur.

The vectorization methods described above are the most commonly used techniques today. Each of them has different uses. In studies where word vectorization is needed, the problem should be determined first, and then the vectorization technique should be preferred according to this problem. As a matter of fact, there are different situations where each technique is strong.

In addition to these, there are also contextual representations such as ELMO and BERT[5]. These issues will be discussed in the next article.

Github

All codes can be found at https://github.com/ademakdogan/word2vec_generator . In this project, word2vec training can be done autonomously according to the column to be determined in any csv file desired. The word2vec model that will be created as a result of the operations is saved under the “model” folder. Example usage can be seen below. This project can also run on docker. Parameters in src/training.py can be optimized according to data size.

```
python3 src/w2vec.py -c </Users/.../data.csv> -t <target_column_name>
```

Github : https://github.com/ademakdogan

Linkedin : https://www.linkedin.com/in/adem-akdo%C4%9Fan-948334177/

Referanslar

[1]Stevens, S. S. (1946). “On the Theory of Scales of Measurement”. *Science, New Series*, 103.2684,677–680.

[2]Aizawa Akiko, (2003). “An information-theoretic perspective of tf–idf measures”. *Information Processing and Management*. **39** (1), 45–65. doi:10.1016/S0306–4573(02)00021–3

[3]Tomas Mikolov, et al. (2013). “Efficient Estimation of Word Representations in Vector Space”. arXiv:1301.3781

[4]Armand Joulin, Edouard Grave, Piotr Bojanowski, Tomas Mikolov,(2017). “Bag of Tricks for Efficient Text Classification”, Conference: Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 2.

[5]Devlin Jacob, Chang Ming-Wei, Lee Kenton, Toutanova Kristina,(2018). “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”

I explain my research on natural language generation for automated…

Nlg 3 min read

Read more from Towards Data Science

Recommended from Medium

Andrew Benesh

GA COVID-19 Report August 22, 2020

Maxim… in Artefact Engineering a…

Sales forecasting in retail: what we learned from the M5 competition

Andrew Benesh

GA COVID-19 Report August 24, 2021

Yaning Wu

Can you hear the sirens?

Priya

PANDAS—A data manipulation and analysis library in python.

Santhanalakshmi Ponnurasan

Power BI's Magical Trio- Bookmark, Buttons & Selection Pane

The Geospatial

Identifying landslide risk zones using LiDAR

Santhanalakshmi Ponnurasan

Analytics Pane in Power BI