Published in Analytics Vidhya

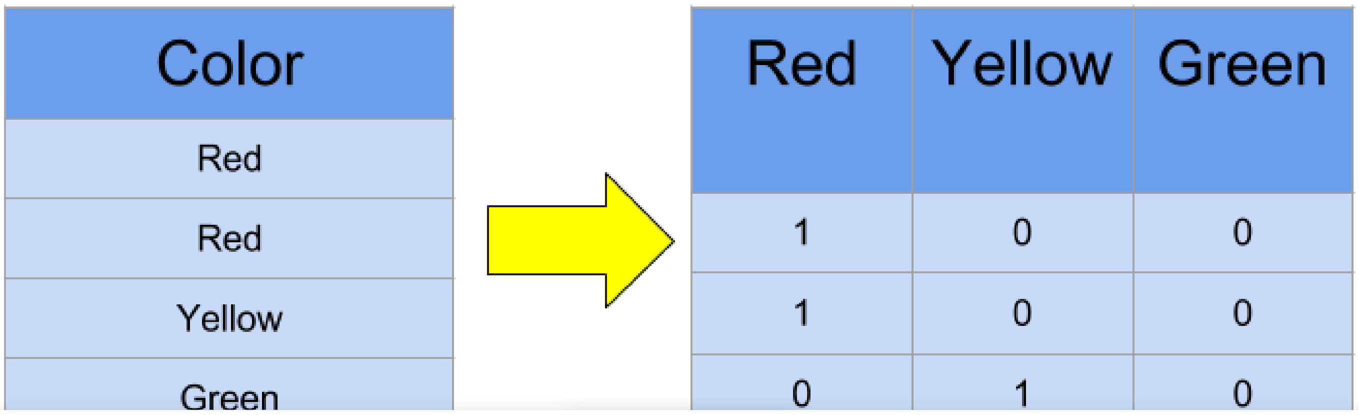
Gaurav Padawe Follow
Oct 27, 2019 · 9 min read · Listen

Word2Vector using Gensim

Introduction :

What is Word2Vec ?

- In layman terms, It is a Algorithm that takes Corpora as an input and outputs it in the form of Vectors.
- A Bit technical Defination : Word2Vec is a model to form / create word Embeddings. Its a modern way of representing Word , wherein each word is represented by a vector (Array of numbers based on Embedding Size). Vectors are nothing but the weights of neurons, so if we set neurons of size 100 then we will have 100 weights and those weights are our Word Embeddings or simply Dense vector.
- Input word must be a One hot encoded. For example :

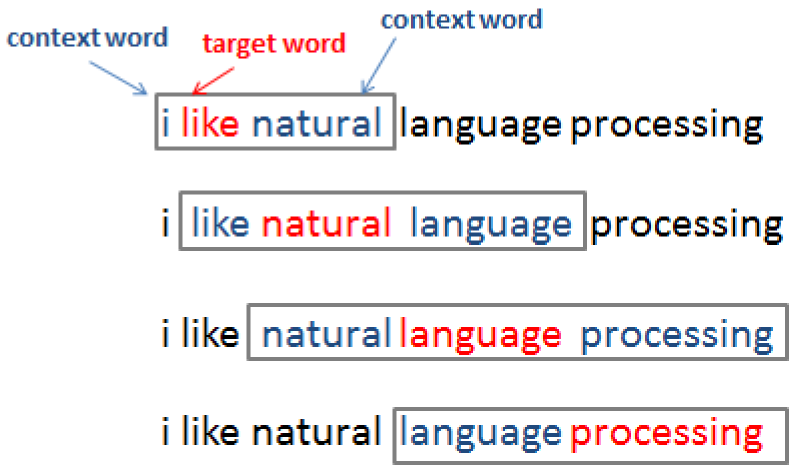


Color
Red
Red
Yellow
Green

Red	Yellow	Green
1	0	0
1	0	0
0	1	0

But Why Word2Vec ?

- Word2Vec finds relation (Semantic or Syntactic) between the words which was not possible by our Tradional TF-IDF or Frequency based approach. When we train the model, each one hot encoded word gets a point in a dimensional space where it learns and groups the words with similar meaning.
- The neural network incorporated here is a Shallow.
- One thing to note here is that we need large textual data to pass into Word2Vec model in order to figure out relation within words or generate meaningful results.
- In general the Word2Vec is based on Window Method, where we have to assign a Window size.


context word
target word
context word
i like natural language processing
i like natural language processing
i like natural language processing
i like natural language processing

- In above visual representation , Window size is set to 1. So, 1 word from both the sides of target are being considered. Similarly, in each iteration, window will slides by single stride and our neighbors will keep changing.
- There are 2 types of Algorithms : CBOW and Skip-gram. (We'll see it in Details as we Progress)

Data Collection :

Details :

- unlabeledTrainData — An extra training set with no labels. The tab-delimited file has a header row followed by 50,000 rows containing an id and text for each review.

Data fields :

- id : Unique ID of each review
- review : Text of the review

Objective :

- The goal is to build Word2Vec Word Embedding model with the help of Gensim.

Gensim :

- Gensim is fairly easy to use module which inherits CBOW and Skip-gram.
- We can install it by using **!pip install gensim** in Jupyter Notebook.
- Alternate way to implement Word2Vec is to build it from scratch which is quite complex.
- Read more about Gensim : https://radimrehurek.com/gensim/index.html
- **FYI, Gensim was developed and is maintained by the NLP researcher Radim Řehůřek and his company RaRe Technologies.**

Loading Packages and Data :

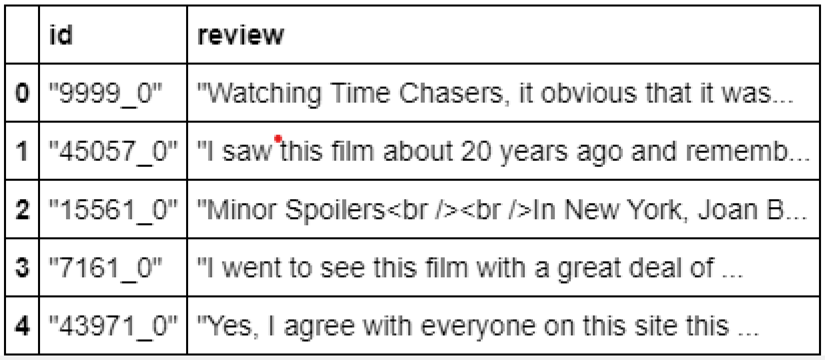
```
#Import packages
import pandas as pd
import gensim

#import beautiful soup , regex
from bs4 import BeautifulSoup
import re, string
```

- Above we're using beautiful soup library to eliminate HTML tags. Alternate way to this is we can use Regex (Regular Expression) to eliminate tags.

```
#reading dataset

df = pd.read_csv('unlabeledTrainData.tsv', header=0, delimiter='\t',
quoting=3)
df.head()
```



	id	review
0	"9999_0"	"Watching Time Chasers, it obvious that it was...
1	"45057_0"	"I saw this film about 20 years ago and rememb...
2	"15561_0"	"Minor Spoilers<br /><br />In New York, Joan B...
3	"7161_0"	"I went to see this film with a great deal of ...
4	"43971_0"	"Yes, I agree with everyone on this site this ...

- The Shape of Data set is 50000 examples having 2 attributes , i.e, id and review.

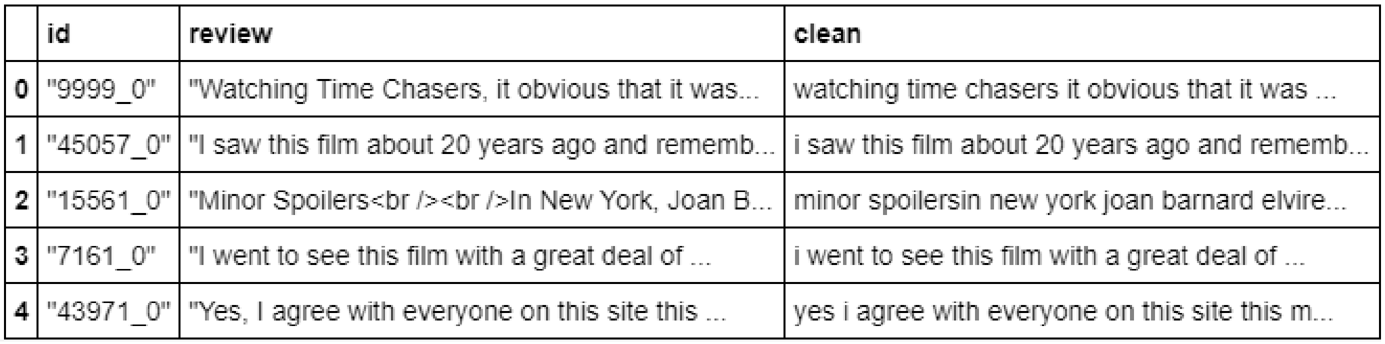
```
'"Watching Time Chasers, it obvious that it was made by a bunch of
friends. Maybe they were sitting around one day in film school and
said, \\"Hey, let\'s pool our money together and make a really bad
movie!\\" Or something like that. What ever they said, they still
ended up making a really bad movie--dull story, bad script, lame
acting, poor cinematography, bottom of the barrel stock music, etc.
All corners were cut, except the one that would have prevented this
film\'s release. Life\'s like that."'
```

- We can observe that our corpus may contain HTML tags, special characters, etc.
- We need to clean the corpus by eliminating these tags and characters.

```
    x = doc.lower()                         #lowerthe case
    x = BeautifulSoup(x, 'lxml').text       #html tag removal
    x = re.sub('[^A-Za-z0-9]+', ' ', x) #separate words by space
    clean.append(x)

#assigning clean list to new attribute
df['clean'] = clean
```

- Above we're using beautiful soup library to eliminate HTML tags. Alternate way to this is we can use Regex (Regular Expression) to eliminate tags.



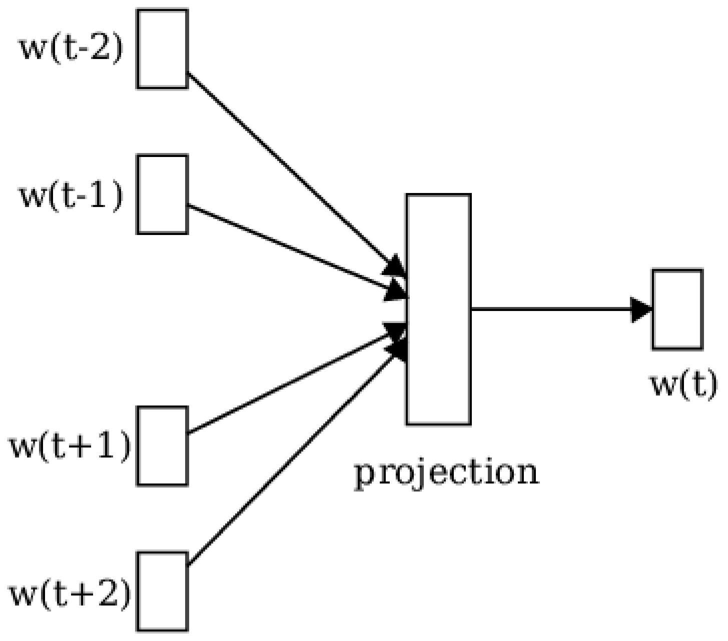
	id	review	clean
0	"9999_0"	"Watching Time Chasers, it obvious that it was...	watching time chasers it obvious that it was ...
1	"45057_0"	"I saw this film about 20 years ago and rememb...	i saw this film about 20 years ago and rememb...
2	"15561_0"	"Minor Spoilers<br /><br />In New York, Joan B...	minor spoilersin new york joan barnard elvire...
3	"7161_0"	"I went to see this film with a great deal of ...	i went to see this film with a great deal of ...
4	"43971_0"	"Yes, I agree with everyone on this site this ...	yes i agree with everyone on this site this m...

- Finally, we are appending the clean set to list and forming a new attribute as clean.

(a) Continuous Bag of Words (CBOW) :

- In here neighboring words are provided as Input to predict the Target. In other words, A context is Provided as input to predict the Target. For example :

- Let us look at this in visual representation :



- We can observe that **W(t)** is being predicted given that the context / neighboring words are fed as input. P-value is estimated for target word based on the input that is fed to the network.
- We've a shallow neural network here. Size of each word embedding or individual dense vector is depended on the number of neurons we've at disposal.
- These vectors are weights learnt by each neuron.
- Regardless of how big or small the word is, it will be represented by size of embedding we set.

Advantages :

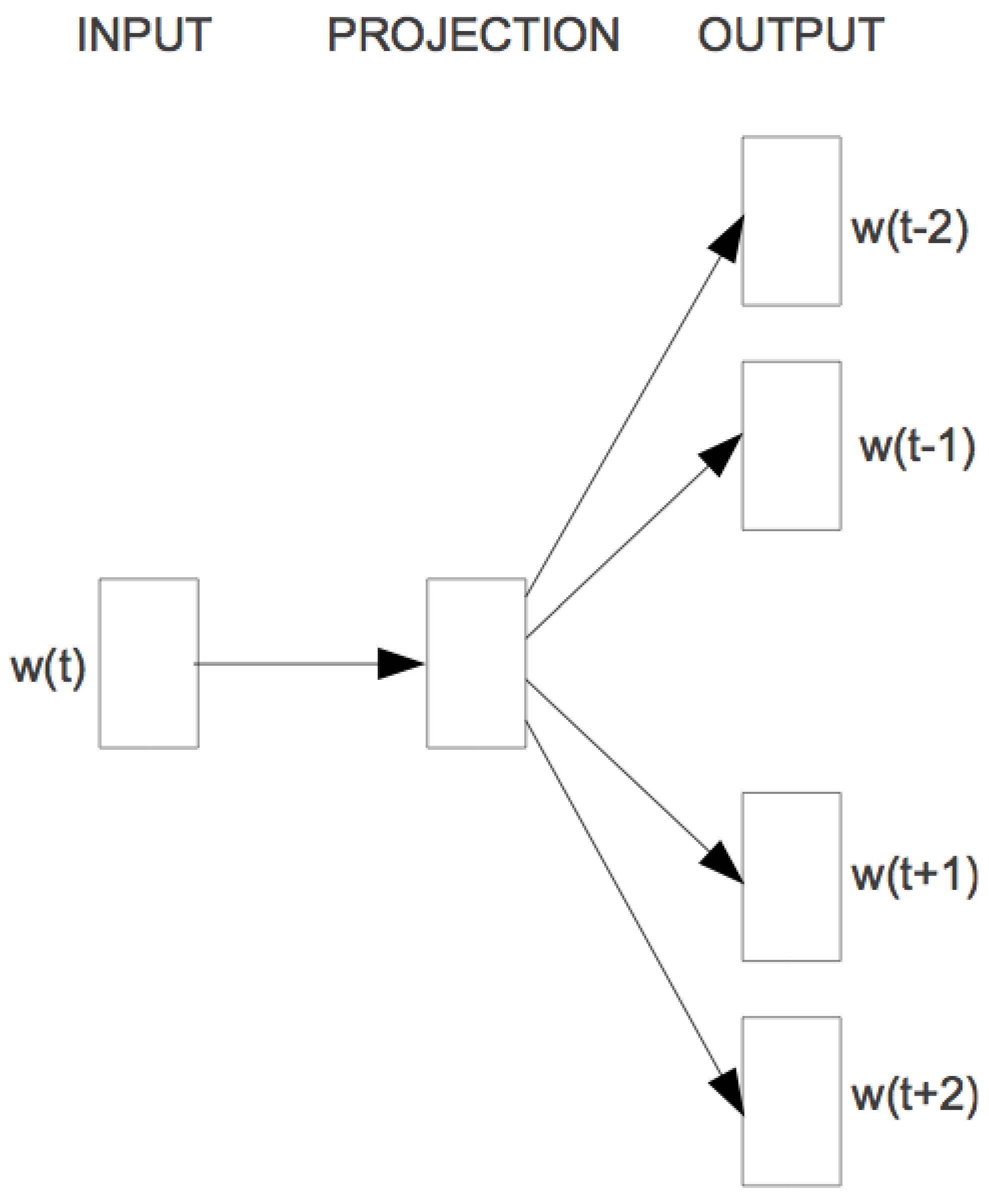
- CBOW is faster and represents frequent words more better.
- When it comes to memory utilization, CBOW tends to consume Low memory.

Disadvantages :

- Needs huge textual data. (***Note : Can't say it is disadvantage since it consumes low memory but its worth mentioning***)

(b) Skip-gram :

- Skip-gram is opposite / inverse of CBOW, wherein a target word is provided as output in order to predict the Contextual / Neighboring words.
- Let us look at this in visual representation :



based on our Window Size). Furthermore, P-value is being calculated for context words being close to the input word we fed to the network.

Advantages :

- Skip-gram works well with smaller data sets.
- Its able to represent rare words well.

Disadvantages :

- Slow on training if data set is huge.
- And is not memory efficient.

What Problems we might face ?

- Both techniques are based on probability estimation.
- Let us consider a example here, we've a corpus of 1000 words. We need to predict either a target word (CBOW) or context (Skip-gram). Now, we specify a fixed Window size for sliding through the corpus. Let's say our Window size is 1 , i.e, In case of CBOW the Input will be 2 Words (Words on both sides of target) while in case of Skip-gram input will be a single word in order to predict context (2 words).
- In this case the probabilities will be estimated for words within a certain window and those probabilites will be higher than that for those words which are far from neighborhood.
- Computing probabilities for huge corpus (e.g.: Millions of words) would be computationally expensive since algorithm will estimate probabilities for all the words at each iteration. In this case, The model may give irrelevant results too. And so to overcome this issue **Negative Sampling** was proposed.
- In **Negative Sampling** the probabilities are estimated for word within our fixed Window size, just like before but only few random words are chosen out of fixed window. By this the load on p-value estimation is low compared to what it was before.

- It depends on nature of problem.
- Both have their own set of benefits.

Word2Vec Modeling

- Further we'll look how to implement Word2Vec and get Dense Vectors.

```
#Word2vec implementation

model = gensim.models.Word2Vec(docs,
                               min_count=10,
                               workers=4,
                               size=50,
                               window=5,
                               iter = 10)
```

Here are few parameters which one could play with :

- **sentences :** The sentences accepts a list of lists of tokens. (Better to have large document list in case of CBOW)
- **size :** Number of Neurons to incorporate in hidden layer or size of Word Embeddings. By default its set to 100.
- **window :** Window Size or Number of words to consider around target. If size = 1 then 1 word from both sides will be considered. By default 5 is fixed Window Size.
- **min_count :** Default value is 5. Words which are infrequent based on minimum count mentioned will be ignored. (*We had chosen 10 as Threshold*)
- **workers :** Number of CPU Threads to use at once for faster training.
- **sg :** Either 0 or 1. Default is 0 or CBOW. One must explicitly define Skip-gram by passing 1.

Open in app Get started

- We've 28656 words in our vocabulary for usage after eliminating words which appeared less than 10.
- And each word will be represented by 50 numbers / weights (which is our Embedding size).

```
#uncomment to view vocabulary
#model.wv.vocab
```

- Above piece of code will disclose the complete Vocabulary.

```
#word2vector representation
model.wv['great']
```

Output :

array([-2.7726305 , -1.0800452 , -1.788979 , 2.340867 , -0.32861072, 0.0651653 , 1.6166486 , -3.2617207 , -3.6233435 , -3.32576 , -1.7012835 , 1.0813012 , -0.24166667, -1.1136819 , 1.7357157 , -2.852496 , 0.2456542 , 0.9012077 , -0.8035166 , 1.7389616 , 1.5673314 , 2.0869598 , 3.3215692 , 0.8369672 , 0.07051245, 2.9767258 , -0.92073804, 0.6535899 , 2.716228 , 2.5288267 , 0.18343763, 1.5990931 , -2.1080818 , 1.5348029 , 0.19268313, -1.7983583 , -0.22839952, -0.22228098, 4.939321 , 2.071981 , -0.2585357 , -1.6617067 , 1.3812392 , -3.7641723 , 1.650655 , -1.4870547 , 2.4975944 , 2.2064195 , -2.383971 , -1.3767233], dtype=float32)

- Above we can observe that "great" word is represented by 50 Weights.

```
#find similar words to the given word
model.wv.most_similar('dumb')
```

```
[('stupid', 0.899315595626831),
 ('lame', 0.8604525923728943),
 ('silly', 0.8508281111717224),
 ('generic', 0.7499771118164062),
 ('pathetic', 0.7478170394897461),
 ('retarded', 0.7461548447608948),
 ('corny', 0.7456734776496887),
 ('cheesy', 0.7454365491867065),
 ('ridiculous', 0.7351659536361694),
 ('bad', 0.7300432920455933)]
```

- One beautiful thing about Gensim is that we have such methods which can give us similar words aligned by highest probability.
- Cosine Similarity is computed to between the words to find most similar words.
- As we passed “dumb” we can observe that how most similar words are aligned by their respective P-values.
- Given “dumb” , most similar words are stupid, lame, silly, etc.

```
#words which doesn't match
model.wv.doesnt_match('house rent trust apartment'.split())
```

Output :

```
'trust'
```

- We can also identify which words is not closely related.
- Above we passed “house, rent, trust, apartment” and output is “trust” — which obviously is Satisfactory.

Output :

```
('girl', 0.7452774047851562)
```

- Above we had performed basic arithmetic problem.
- If Woman + Man = King - ?
- Output is “Girl” having p-value 0.74 , which is acceptable, If we would have larger corpora then we may get output as “Queen”.

```
#word embeddings
model.wv.vectors
```

Output :

```
array([[-1.0976436 , -1.0300254 ,  1.0157741 , ...,  2.0884573 ,
         1.2929492 , -0.50073916],
       [-0.10799529,  1.1408263 ,  0.23857029, ..., -0.6748595 ,
         1.4341863 , -0.03276591],
       [-0.67246455,  1.097886  ,  0.97168344, ..., -0.7108262 ,
        -0.20907082, -0.37925354],
       ...,
       [ 0.44054744,  0.13000782,  0.04757666, ...,  0.09963967,
         0.2213286 , -0.13963003],
       [ 0.42221656,  0.37493396,  0.28681585, ..., -0.1619114 ,
         0.2634282 ,  0.0619109 ],
       [ 0.43446574,  0.11213642,  0.05965824, ..., -0.02202358,
         0.1439297 , -0.01608269]], dtype=float32)
```

- Above is our array having 50 dimensions.
- Which we can further use for Sentiment Analysis with the help of Embedding Layer.

```
#saving model
model.wv.save('word2vector-50.bin')
```

- Further, we can also load the model using load() method.

```
#loading model
model = gensim.models.Word2Vec.load('word2vec-50.bin')
```

- We can also load Google Word2Vec model by downloading file from here : https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21pQmM/edit
- It is a pre-trained model by Google which was trained on Google News Data , where each word is represented by 300 embedding size.
- There are many such pre-trained models available, few to name are GloVe (Global Vectors for Word Representation) by Stanford, FastText by Facebook AI Research Lab. One can load them and examine which works better.

Conclusion

- In this case study we learnt CBOW and Skip-gram in detail.
- We also seen what obstacle we were facing and how researchers overcame it by adding Negative Sampling.
- We trained the model by passing a clean Tokenized corpora.
- Finally, we looked at how we can Save / Load model and use it for future purposes.

What's next ?

- We can use word embeddings as feature for Sentiment Analysis with the help of Embedding Layer.