Word 2 Vec (text → vec) (take semantic meaning into consideration)

→ state of the art

→ word ⟶ vector

* we will learn full detail in ⟶ deep learning
  (alternative explanation) → Matrix factorization

→ Here, we will learn Geometrically.

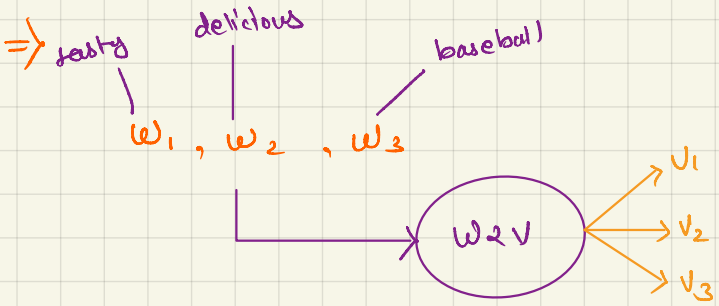
(it takes word not sentence)

words ⟶ dense d-dimensional vector ⟶ d dim vector
⟶ not a sparse vector.

d - typically

50, 100, 200, 300

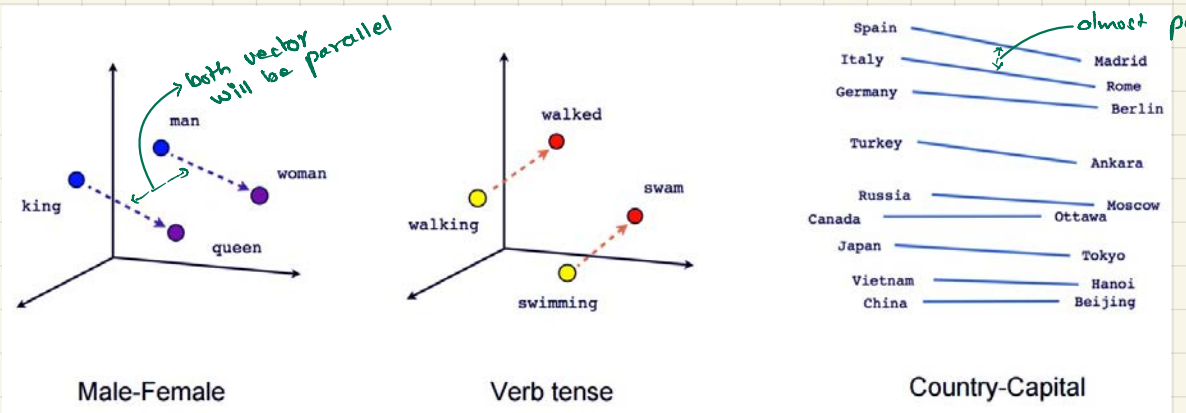
{ Bow / tf-idf
sentence / text → sparse vector



W2V :- 300 - dim

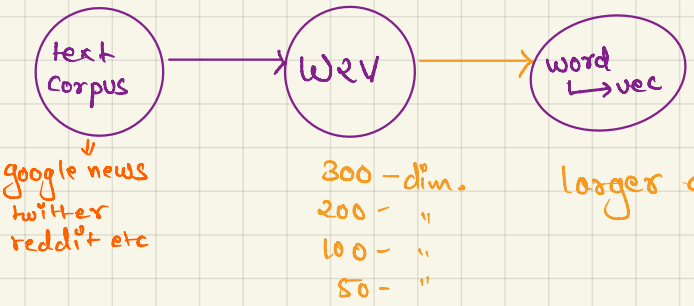
① 'w1' & 'w2' are semantically similar than 'v1' & 'v2' will be closer.

② relationships :-



$(V_{man} - V_{woman}) \parallel (V_{king} - V_{queen})$

Note :- 'w2v' learning all this relationship automatically from 'raw-text'.



if

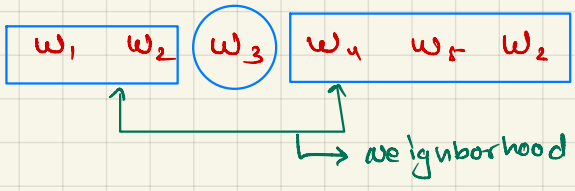
* data corpus size is large ⇒ dimensionality is also large

google-news ⟶ w2v ⟶ '300-dim'

review-text (500k review) ⟶ w2v ⟶ Word → vec

Core :-

w2v :- (intuitive)



neighbourhood of 'wi'

$$w2v(w_3) = \begin{cases} \text{if } N(w_i) \approx N(w_j), \\ \text{then } v_i \approx v_j \end{cases}$$

Avg-w2v, tf-IDF weighted w2v

w2v :- word → vec (d-dim.)

r_i :- sequences of words/sentences

Q) How can I convert sentences to vector using w2v?

A) There are many ways but the simplest technique is

Avg w2v :-

r_1 : w_1 w_2 w_1 w_3 w_4 w_5 (n_1 words) → no. of words in 'r_i'.

v_1 : $\frac{1}{n_1}\left[w2v(w_1) + w2v(w_2) + w2v(w_1) + \cdots + w2v(w_5) \right]$

$w2v(w_1)$ → d-dim

avg − w2v → it's not perfect but it works well enough.
(simple to leverage w2v to build sentences vect.)

tf-idf − w2v :-

r_1 : w_1 w_2 w_1 w_3 w_4 w_5

	w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9
tf-IDF :- of r_1	t_1	t_2	t_3	t_4	t_5	0	0	0	0

tf-IDf-w2v(r_1) :- $$\frac{t_1 * w2v(w_1) + t_2 * w2v(w_2) + \dots + t_5 * w2v(w_5)}{t_1 + t_2 + t_3 + t_4 + t_5}$$ (t_i → tf-IDf)

$$(\text{tfIDf-w2v})_{\text{of } r_1} = \sum_{i:\text{words}} \frac{t_i * w2v(w_i)}{\sum_{i:\text{words}} t_i}$$

t_i → tfIDf(w_i, r_1)

If all of 't_i' = 1 then tfIDf − w2v = avg − w2v

avg − w2v
tfIDf − w2v
} → weighting schemes sentence(s) → vec.

→ sentences (r) → vec → deep learning
thought vectors

Bow Words (code sample)

[7.2.2] Bag of Words (BoW)

→ see documentation of this. (Try to practice with every parameter in this given func^n.)

```
In [82]: #BoW
         count_vect = CountVectorizer() #in scikit-learn
         final_counts = count_vect.fit_transform(final['Text'].values)
```

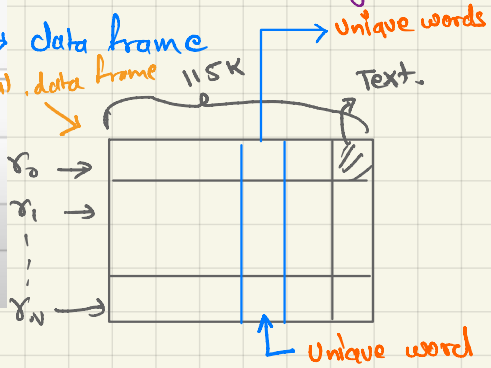
→ data frame (final.data frame)

```
In [83]: type(final_counts)
Out[83]: scipy.sparse.csr.csr_matrix
```

→ sparse matrix

```
In [84]: final_counts.get_shape()
Out[84]: (364171, 115281)
```

364K (No. of reviews)

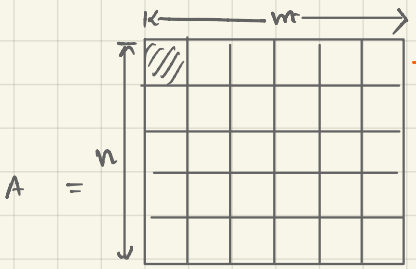


Q) we want to convert every review into vectors using BOW?

↳ code written above.

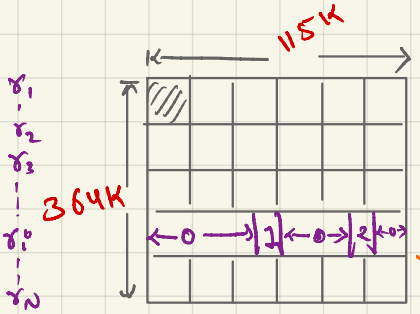
Sparse matrix → If most of the vector in a matrix is sparse vector then it is Sparse matrix.

Sparse vector:- If most of the element in a vector is same then it is a sparse vector.



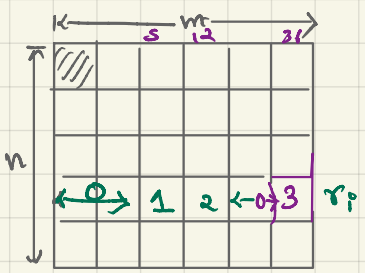
→ To store that matrix 'A', I need 'n x m' space

space complexity = O(nm)



→ In 'BOW' most of the values in any row 'r_i' are zero.

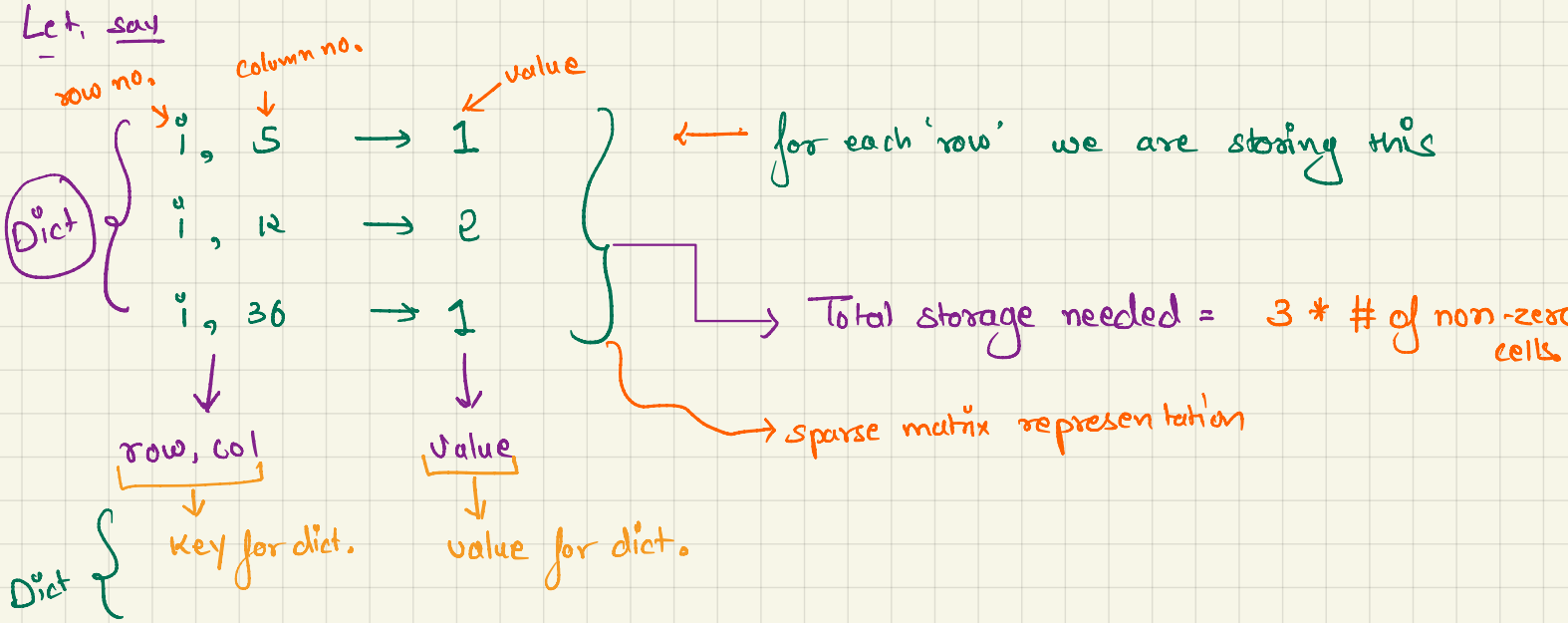
→ to store that matrix we have order of 364K x 115K which is too much. So, how can we solve that?



$r_0, r_1, r_2, \ldots r_i \ldots r_N$

① for each row r_i: we need O(m) space for storage. How can we decrease that?

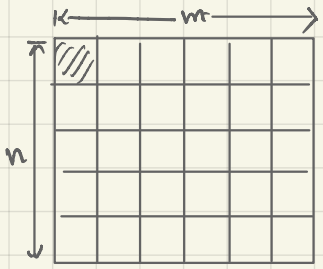
Let, say



→ Ri :- O(m) Space (m = 100) → (row no., col. no., value → 3 * non zero cells)

11 times reduction → 9

Sparsity of a matrix



n x m → cells → k cells have non-zero values rest of them have '0'.

$$\text{Sparsity of 'A'} = \frac{k}{n \times m}$$

→ if sparsity = 0.01 → means only 1% of this matrix have non-zero value.

→ O(n x m) → O(3 x k) → O(k)

Note :- See documentation of sklearn.feature_extraction.text.CountVectorizer

↳ program & see for every parameter in this with example.
(same do with every function you learn)

Text preprocessing (code sample)

[3]. Text Preprocessing.

Now that we have finished deduplication our data requires some preprocessing before we go on further with analysis and making the prediction model.

Hence in the Preprocessing phase we do the following in the order below:-

1. Begin by removing the html tags → < > , < /b> ---
2. Remove any punctuations or limited set of special characters like , or . or # etc.
3. Check if the word is made up of english letters and is not alpha-numeric
4. Check to see if the length of the word is greater than 2 (as it was researched that there is no adjective in 2-letters)
5. Convert the word to lowercase
6. Remove Stopwords
7. Finally Snowball Stemming the word (it was obsereved to be better than Porter Stemming)

After which we collect the words used to describe positive and negative reviews

find sentences containing HTML tags.

```
i=0;
for sent in final['Text'].values:
    if (len(re.findall('<.*?>', sent))):
        print(i)
        print(sent)
        break;
    i += 1;
```

→ HTML text

→ regular expression

→ break → new line.

```
6
I set aside at least an hour each day to read to my son (3 y/o). At this
 point, I consider myself a connoisseur of children's books and this is o
ne of the best. Santa Clause put this under the tree. Since then, we've r
ead it perpetually and he loves it.<br /><br />First, this book taught hi
m the months of the year.<br /><br />Second, it's a pleasure to read. Wel
l suited to 1.5 y/o old to 4+.<br /><br />Very few children's books are w
orth owning. Most should be borrowed from the library. This book, howeve
r, deserves a permanent spot on your shelf. Sendak's best.
```

→ you will have to understand it

Learn about regulare expression (from this site) [re]

→ https://pymotw.com/2/re/

→ regular expressions → Compiler design, TOC, 'grep' command (unix)

```
import re
# Tutorial about Python regular expressions: https://pymotw.com/2/re/
import string
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer
from nltk.stem.wordnet import WordNetLemmatizer

stop = set(stopwords.words('english')) #set of stopwords
sno = nltk.stem.SnowballStemmer('english') #initialising the snowball stemm

def cleanhtml(sentence): #function to clean the word of any html-tags
    cleanr = re.compile('<.*?>')
    cleantext = re.sub(cleanr, ' ', sentence)
    return cleantext
def cleanpunc(sentence): #function to clean the word of any punctuation or
    cleaned = re.sub(r'[?|!|\'|"|#]',r'',sentence)
    cleaned = re.sub(r'[.|,|)|(|\|/]',r' ',cleaned)
    return  cleaned
print(stop)
print('************************************')
```

→ storing all stop words in 'english' to stop.

→ here I am giving language coz nltk can no 'non-english' language to.

→ substitute.

```
    return  cleaned
print(stop)
print('************************************')
print(sno.stem('tasty'))
```

```
{'their', 'isn', 'such', 'where', 'this', 'they', 'while', 'about', 'ther
e', 'myself', 'from', 'mightn', 'was', 'between', 'who', 'are', 'only',
 'our', 'those', 'through', 'any', 'is', 'a', 'nor', 'mustn', 'shouldn',
 'yourself', 'no', 'itself', 'that', 'himself', 'out', 'what', 'my', 'aga
inst', 'below', 's', 'for', 'be', 'into', 'few', 'needn', 'you', 'aren',
 'when', 'all', 'him', 'but', 've', 'yours', 'being', 'why', 'own', 'up',
'whom', 're', 'and', 'she', 'me', 'of', 'than', 'doesn', 'both', 'same',
 'too', 'am', 'how', 'not', 'her', 'd', 'until', 'o', 'your', 'yourselve
s', 'by', 'other', 'once', 'an', 'just', 'to', 'these', 'don', 'its', 'ha
ven', 'having', 'some', 'shan', 'theirs', 'under', 'we', 'ain', 'it', 'a
t', 'in', 'y', 'the', 'off', 'herself', 'down', 'because', 'i', 'now', 't
hemselves', 'each', 'or', 'were', 'if', 'can', 'did', 'm', 'which', 'coul
dn', 'ourselves', 'hadn', 'has', 'wasn', 'with', 'here', 'further', 'the
m', 'hasn', 'should', 'ma', 'then', 'he', 'very', 'above', 'been', 'did
n', 'during', 'most', 'hers', 'will', 'have', 'doing', 'again', 'had', 'd
o', 'before', 'as', 'wouldn', 'his', 'after', 'ours', 'does', 'so', 'on',
 'more', 't', 'won', 'weren', 'over', 'll'}
************************************
tasti
```

tasti → stem word for 'tasty'. (acc to snowball stemmer)

```
#Code for implementing step-by-step the checks mentioned in the pre-process
# this code takes a while to run as it needs to run on 500k sentences.
i=0
str1=' '
final_string=[]
all_positive_words=[] # store words from +ve reviews here
all_negative_words=[] # store words from -ve reviews here.
s=''
for sent in final['Text'].values:
    filtered_sentence=[]
    #print(sent);
    sent=cleanhtml(sent) # remove HTML tags
    for w in sent.split():
        for cleaned_words in cleanpunc(w).split():
            if((cleaned_words.isalpha()) & (len(cleaned_words)>2)):
                if(cleaned_words.lower() not in stop):
                    s=(sno.stem(cleaned_words.lower())).encode('utf8')
                    filtered_sentence.append(s)
                    if (final['Score'].values)[i] == 'positive':
                        all_positive_words.append(s) #list of all words use
                    if (final['Score'].values)[i] == 'positive':
                        all_positive_words.append(s) #list of all words use
                    if(final['Score'].values)[i] == 'negative':
                        all_negative_words.append(s) #list of all words use
                else:
                    continue
            else:
                continue
    #print(filtered_sentence)
    str1 = b" ".join(filtered_sentence) #final string of cleaned words
    #print("*****************************************************************
    final_string.append(str1)
    i+=1
```

list of final words after all the processing → doing it for the future.

```
final['CleanedText']=final_string #adding a column of CleanedText which dis
```

data frame → 'final'

creating new column for 'CleanedText' in data frame.

```
final.head(3) #below the processed review can be seen in the CleanedText Co


# store final table into an SQLite table for future.
conn = sqlite3.connect('final.sqlite')
c=conn.cursor()
conn.text_factory = str
final.to_sql('Reviews', conn, flavor=None, schema=None, if_exists='replace'
```

storing my final data in 'final.sqlite' so, I do not have to repeat again & again.

Bi-Grams and n-grams (code sample)

[7.2.4] Bi-Grams and n-Grams.

Motivation

Now that we have our list of words describing positive and negative reviews lets analyse them.

We begin analysis by getting the frequency distribution of the words as shown below

```
freq_dist_positive=nltk.FreqDist(all_positive_words)
freq_dist_negative=nltk.FreqDist(all_negative_words)
print("Most Common Positive Words : ",freq_dist_positive.most_common(20))
print("Most Common Negative Words : ",freq_dist_negative.most_common(20))
```

```
Most Common Positive Words :  [(b'like', 139429), (b'tast', 129047), (b'g
ood', 112766), (b'flavor', 109624), (b'love', 107357), (b'use', 103888),
 (b'great', 103870), (b'one', 96726), (b'product', 91033), (b'tri', 8679
1), (b'tea', 83888), (b'coffe', 78814), (b'make', 75107), (b'get', 7212
5), (b'food', 64802), (b'would', 55568), (b'time', 55264), (b'buy', 5419
8), (b'realli', 52715), (b'eat', 52004)]
```

+ve +ve

```
Most Common Positive Words :  [(b'like', 139429), (b'tast', 129047), (b'g
ood', 112766), (b'flavor', 109624), (b'love', 107357), (b'use', 103888),
 (b'great', 103870), (b'one', 96726), (b'product', 91033), (b'tri', 8679
1), (b'tea', 83888), (b'coffe', 78814), (b'make', 75107), (b'get', 7212
5), (b'food', 64802), (b'would', 55568), (b'time', 55264), (b'buy', 5419
8), (b'realli', 52715), (b'eat', 52004)]
Most Common Negative Words :  [(b'tast', 34585), (b'like', 32330), (b'pro
duct', 28218), (b'one', 20569), (b'flavor', 19575), (b'would', 17972),
 (b'tri', 17753), (b'use', 15302), (b'good', 15041), (b'coffe', 14716),
 (b'get', 13786), (b'buy', 13752), (b'order', 12871), (b'food', 12754),
 (b'dont', 11877), (b'tea', 11665), (b'even', 11085), (b'bex', 10844),
 (b'amazon', 10073), (b'make', 9840)]
```

(not tasty) -ve

not like

364K reviews

like occur a lot of time in both positive & negative review.
This is because probably it is occuring as "not like" in negative review.
So, if I take sequence of information I can capture the information eg:- 'not like'

Observation:- From the above it can be seen that the most common positive and the negative words overlap for eg. 'like' could be used as 'not like' etc.
So, it is a good idea to consider pairs of consequent words (bi-grams) or q sequnce of n consecutive words (n-grams)

```
#bi-gram, tri-gram and n-gram

#removing stop words like "not" should be avoided before building n-grams
count_vect = CountVectorizer(ngram_range=(1,2) ) #in scikit-learn
final_bigram_counts = count_vect.fit_transform(final['Text'].values)
```

get me all unigram
get me till bigram

```
final_bigram_counts.get_shape()
```

```
(364171, 2910192)
```

→ earlier we had 115K dim when I only used unigram.
→ Now, we have 2.9M dim (~20 times more) when I call for bigram.
(coz no. of bigram always more) so, as n↑ our dim. also ↑.

→ ngram_range : tuple (min_n, max_n)

if min_n → 1 → Uni gram
max_n → 2 → Bi gram

ngram_range = (1,4)
get all unigram
get all bigram
till fourgram

TF-IDF (code)

- you can also find 'ngram' using 'TfidfVectorizer' in sklearn.

[7.2.5] TF-IDF

→ see the documentation of this

scikit-learn

```
tf_idf_vect = TfidfVectorizer(ngram_range=(1,2))
final_tf_idf = tf_idf_vect.fit_transform(final['Text'].values)
```

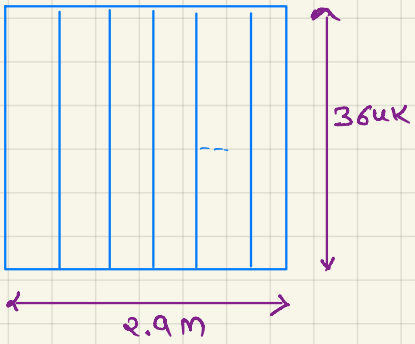
→ considering uni & bi gram

→ no stopping, no stemming

```
final_tf_idf.get_shape()
```

```
(364171, 2910192)
```

→ same as above




```
features = tf_idf_vect.get_feature_names()
len(features)
```

```
2910192
```

r_1 [w_1 w_2 w_3 w_4 ...]

```
features[100000:100010]
```

→ seeing only 10 features

→ bigram

```
['ales until',
 'ales ve',
 'ales would',
 'ales you',
 'alessandra',
 'alessandra ambrosia',
 'alessi',
 'alessi added',
 'alessi also',
 'alessi and']
```

→ unigram

```
# covnert a row in saprsematrix to a numpy array
print(final_tf_idf[3,:].toarray()[0])
```

→ getting all the column of row 3 (review 3)

→ [0. 0. 0. ..., 0. 0. 0.]

→ converting into numpy array.

```
[ 0.  0.  0. ...,  0.  0.  0.]
```

```
# source: https://buhrmann.github.io/tfidf-analysis.html
def top_tfidf_feats(row, features, top_n=25):
    ''' Get top n tfidf values in row and return them with their correspond
    topn_ids = np.argsort(row)[::-1][:top_n]
    top_feats = [(features[i], row[i]) for i in topn_ids]
    df = pd.DataFrame(top_feats)
    df.columns = ['feature', 'tfidf']
    return df

top_tfidf = top_tfidf_feats(final_tf_idf[1,:].toarray()[0],features,25)
```

```
top_tfidf
```

→ here we are getting, top 25 tf-idf values of row '1' (review 1)

r_1 →

V_1 → [0.17 | 0.17 | ...] (1 ... 25)

	feature	tfidf
0	sendak books	0.173437
1	rosie movie	0.173437
2	paperbacks seem	0.173437
3	cover version	0.173437
4	these sendak	0.173437
5	the paperbacks	0.173437
6	pages open	0.173437
7	really rosie	0.168074
8	incorporates them	0.168074
9	paperbacks	0.168074

→ got top 25 terms in 'review 1'.

25

Word 2 vec

→ If you want to train your own model → Data corpus (36 MB) → w2v → w → v

→ If you want to use google news data → w2v – google news → very large data corpus → w → v

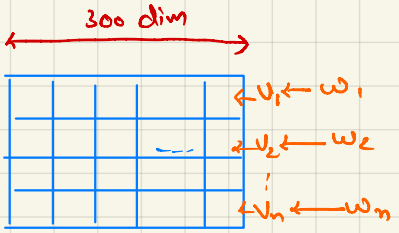
w2v model → check documentation of this.

```
# Using Google News Word2Vectors
from gensim.models import Word2Vec
from gensim.models import KeyedVectors
import pickle

# in this project we are using a pretrained model by google
# its 3.3G file, once you load this into your memory
# it occupies ~9Gb, so please do this step only if you have >12G of ram
# we will provide a pickle file wich contains a dict ,
# and it contains all our courpus words as keys and  model[word] as values
# To use this code-snippet, download "GoogleNews-vectors-negative300.bin"
# from https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21pQmM/edit
# it's 1.9GB in size.

model = KeyedVectors.load_word2vec_format('GoogleNews-vectors-negative300.bin')
```

this data-set is a big table where ever word has a vector of 300 dim. (here for million & million words it has vector representation)



→ w2v for word 'computer'.

```
model.wv['computer']
```

```
array([  1.07421875e-01,  -2.01171875e-01,   1.23046875e-01,
         2.11914062e-01,  -9.13085938e-02,   2.16796875e-01,
        -1.31835938e-01,   8.30078125e-02,   2.02148438e-01,
         4.78515625e-02,   3.66210938e-02,  -2.45361328e-02,
         2.39257812e-02,  -1.60156250e-01,  -2.61230469e-02,
         9.71679688e-02,  -6.34765625e-02,   1.84570312e-01,
         1.70898438e-01,  -1.63085938e-01,  -1.09375000e-01,
         1.49414062e-01,  -4.65393066e-04,   9.61914062e-02,
         1.68945312e-01,   2.60925293e-03,   8.93554688e-02,
         6.49414062e-02,   3.56445312e-02,  -6.93359375e-02,
        -1.46484375e-01,  -1.21093750e-01,  -2.27539062e-01,
         2.45361328e-02,  -1.24511719e-01,  -3.18359375e-01,
        -2.20703125e-01,   1.30859375e-01,   3.66210938e-02,
        -3.63769531e-02,  -1.13281250e-01,   1.95312500e-01,
         9.76562500e-02,   1.26953125e-01,   6.59179688e-02,
```

→ total 300 dim.

Gensim give us very powerful functionality.

```
model.wv.similarity('woman', 'man')
```

```
0.7664012230995351 8
```

→ that much similar 'man' & 'woman'.

similar words have value → 1

non-" " " " → 0

```
model.wv.most_similar('woman')
```

```
[('man', 0.7664012312889099),
 ('girl', 0.7494641542434692),
 ('teenage_girl', 0.7336830496788025),
 ('teenager', 0.6317086219787598),
 ('lady', 0.6288787126541138),
 ('teenaged_girl', 0.6141784191131592),
 ('mother', 0.607630729675293),
 ('policewoman', 0.6069462299346924),
 ('boy', 0.5975908041000366),
 ('Woman', 0.5770982503890991)]
```

→ caps

w2v is not a perfect technique, but it predicts really well.

w2v :- some of the stem or root word may or may not be in the corpus. so, be careful about that. It depends on data you train your model with.

```
model.wv.most_similar('tasti')  # "tasti" is the stemmed word for tasty, ta
```

```
---------------------------------------------------------------------------
--
KeyError                                  Traceback (most recent call las
t)
<ipython-input-14-275c3585c172> in <module>()
----> 1 model.wv.most_similar('tasti')  # "tasti" is the stemmed word for
 tasty, tastful

/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/site-pack
ages/gensim/models/keyedvectors.py in most_similar(self, positive, negati
ve, topn, restrict_vocab, indexer)
```

Train your own model

→ first you will have to convert every sentences into words and then remove all puncuations, 'alpha numeric word ___ .

```
# Train your own Word2Vec model using your own text corpus
import gensim → see documentation.
i=0
list_of_sent=[]
for sent in final['Text'].values:
    filtered_sentence=[]
    sent=cleanhtml(sent)
    for w in sent.split():
        for cleaned_words in cleanpunc(w).split():
            if(cleaned_words.isalpha()):
                filtered_sentence.append(cleaned_words.lower())
            else:
                continue
    list_of_sent.append(filtered_sentence)
```

```
print(final['Text'].values[0])
print("*****************************************************************")
```

```
this witty little book makes my son laugh at loud. i recite it in the car
as we're driving along and he always can sing the refrain. he's learned a
bout whales, India, drooping roses:  i love all the new words this book
 introduces and the silliness of it all.  this is a classic book i am  wi
lling to bet my son will STILL be able to recite from memory when he is
 in college
*****************************************************************
['this', 'witty', 'little', 'book', 'makes', 'my', 'son', 'laugh', 'at',
 'loud', 'i', 'recite', 'it', 'in', 'the', 'car', 'as', 'were', 'drivin
g', 'along', 'and', 'he', 'always', 'can', 'sing', 'the', 'refrain', 'he
s', 'learned', 'about', 'whales', 'india', 'drooping', 'i', 'love', 'al
l', 'the', 'new', 'words', 'this', 'book', 'introduces', 'and', 'the', 's
illiness', 'of', 'it', 'all', 'this', 'is', 'a', 'classic', 'book', 'i',
 'am', 'willing', 'to', 'bet', 'my', 'son', 'will', 'still', 'be', 'abl
e', 'to', 'recite', 'from', 'memory', 'when', 'he', 'is', 'in', 'colleg
e']
```

This sentences are now converted into list.

After converting here I'm training my model.

```
w2v_model=gensim.models.Word2Vec(list_of_sent,min_count=5,size=50, workers=4)
```

→ see documentation.
→ dim. of the vector (w → v)
→ no. of cores you wanna use.
→ if a word doesnot occur at least 'five' times don't construct w2v for it.

```
words = list(w2v_model.wv.vocab)
print(len(words))
```

33783 → your dictionary (364K).

```
w2v_model.wv.most_similar('tasty')
```

→ here, I am finding 'similar' words using my corpus.

```
[('tastey', 0.909038245677948),
 ('satisfying', 0.8556904792785645),
 ('yummy', 0.8543208837509155),
 ('filling', 0.8233586549758911),
```

→ as you can see the word you are getting is very similar to 'tasty'. you know your model and training went well.

```
w2v_model.wv.most_similar('like')
```

```
[('resemble', 0.7139369249343872),
 ('mean', 0.651788055896759),
 ('prefer', 0.646423876285553),
 ('dislike', 0.6413203477859497),
 ('overpower', 0.6197512745857239),
 ('think', 0.5993291735649109),
 ('overwhelm', 0.5929116606712341),
 ('enjoy', 0.5895069241523743),
 ('gross', 0.5853881239891052),
 ('alright', 0.5824472308158875)]
```

(opp. meaning)

(opp.)

```
count_vect_feat = count_vect.get_feature_names() # list of words in the BoW
count_vect_feat.index('like')
print(count_vect_feat[64055])
```

(want to find index of like)

```
like
```

Avg w2v, TFIDF-w2v

```
# average Word2Vec
# compute average word2vec for each review.
sent_vectors = []; # the avg-w2v for each sentence/review is stored in this list
for sent in tqdm(list_of_sentance): # for each review/sentence
    sent_vec = np.zeros(50) # as word vectors are of zero length 50, you might need to change this to 300 if you use google's w2v
    cnt_words =0; # num of words with a valid vector in the sentence/review
    for word in sent: # for each word in a review/sentence
        if word in w2v_words:
            vec = w2v_model.wv[word]
            sent_vec += vec
            cnt_words += 1
    if cnt_words != 0:
        sent_vec /= cnt_words
    sent_vectors.append(sent_vec)
print(len(sent_vectors))
print(len(sent_vectors[0]))
```

→ i'm using my model

```
100%|| 4986/4986 [00:03<00:00, 1330.47it/s]
4986
50
```

```
# TF-IDF weighted Word2Vec
tfidf_feat = model.get_feature_names() # tfidf words/col-names
# final_tf_idf is the sparse matrix with row= sentence, col=word and cell_val = tfidf

tfidf_sent_vectors = []; # the tfidf-w2v for each sentence/review is stored in this list
row=0;
for sent in tqdm(list_of_sentance): # for each review/sentence
    sent_vec = np.zeros(50) # as word vectors are of zero length
    weight_sum =0; # num of words with a valid vector in the sentence/review
    for word in sent: # for each word in a review/sentence
        if word in w2v_words and word in tfidf_feat:
            vec = w2v_model.wv[word]
#             tf_idf = tf_idf_matrix[row, tfidf_feat.index(word)]
            # to reduce the computation we are
            # dictionary[word] = idf value of word in whole courpus
            # sent.count(word) = tf valeus of word in this review
            tf_idf = dictionary[word]*(sent.count(word)/len(sent))
            sent_vec += (vec * tf_idf)
            weight_sum += tf_idf
    if weight_sum != 0:
        sent_vec /= weight_sum
    tfidf_sent_vectors.append(sent_vec)
    row += 1
```

$tf\ idf\ (w_i, r_j, D_c)$