SQLite :-

→ `pd.read_sql_query("""SELECT name FROM sqlite_master WHERE type='table' ORDER BY name; """, con)` → Prints list of tables.

	name
0	Reviews

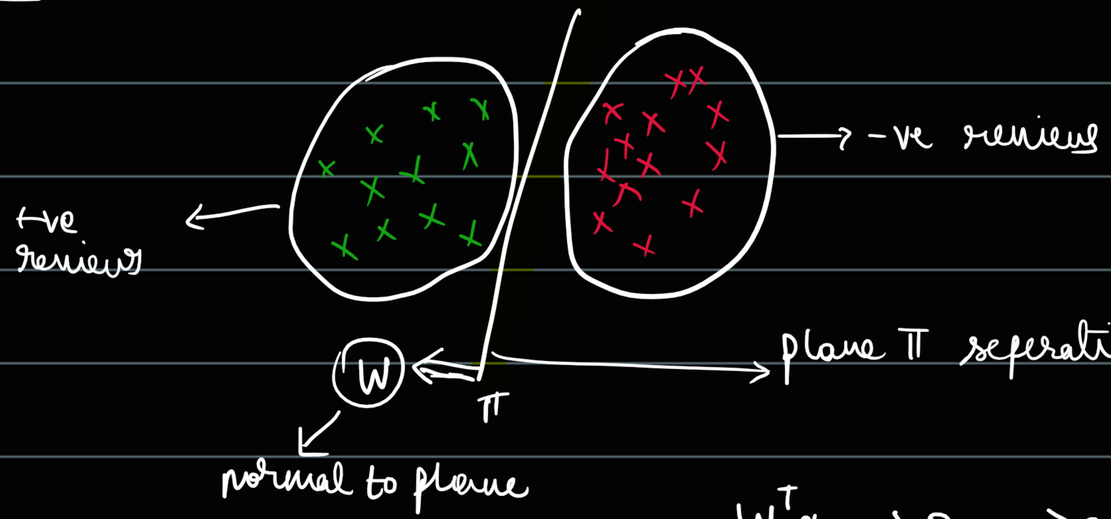
→ df.drop_duplicates(subset={" ", " ",... "}, keep="first") (Pandas)

↳ drops duplicate values with these column name & keeps the first occurance.

→ Use common sense to clean data. No general technique.

→ how to convert text to numerical vectors?

ex:- Review text ⟶ n-dim vector



(reviews are represented by 'X' after converting to vector)

$W^T x_1 > 0$ ⇒ review x_1 is a positive review (on same side)

$W^T x_2 < 0$ ⇒ review x_1 is a negative review (on diff side)

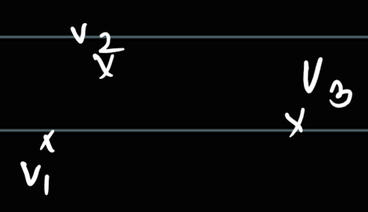
→ Rules for conversion from text to vector :-

r_1 r_2 r_3

assume semantic similarity b/w $(r_1, r_2) > (r_1, r_3)$

then distance b/w $(v_1, v_2) <$ dist b/w (v_1, v_3)

ie; if they are similar, vectors must be close.



Bag of words :-

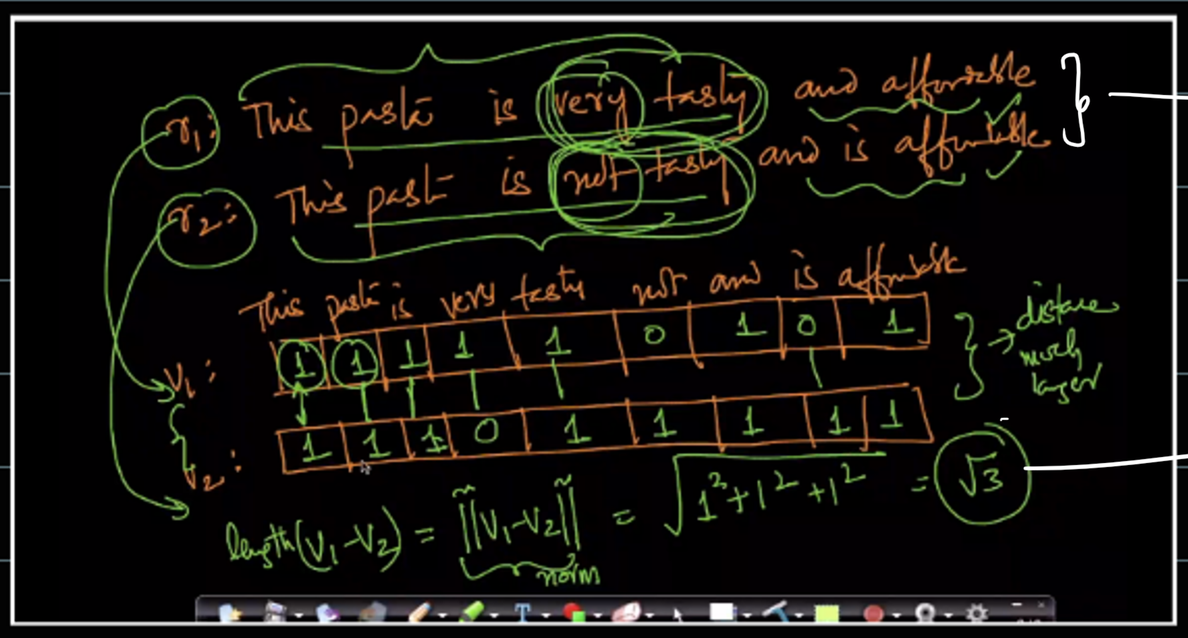
step ① :- Constructing dictionary (set of all unique words)

step ② :- Assume there are d unique words across all 'n' documents/reviews. [Collection of documents is called a corpus]

For each review (d dimensional vector)

The number in pos represents the number of times that word occured in the review.

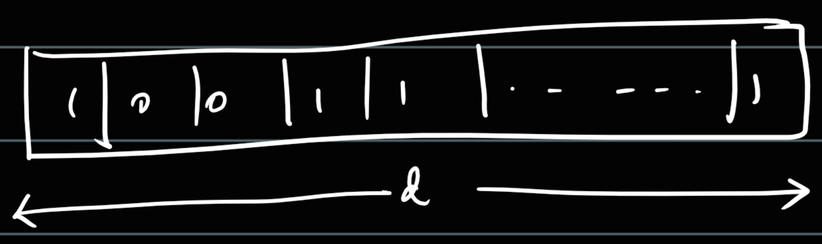
Each word is a different dimensions.



→ Diff sentences but distance is less compared to length 'd' (9)

Boolean bag of words / binary bag of words :-

Instead of counting number of occurances, it only counts occurances -



$\|v_1 - v_2\| \approx \sqrt{\#\text{ of differing words}}$

→ We can notice that some of the words in BoW don't matter (is, and, a, the etc.,) → Stopwords

→ Bag of words is still used for creating baselines with Generalized Linear Models.

→ Removing stop word = Smaller dimensions data.

These are called text preprocessing steps.

→ from nltk.corpus import stopwords

→ We also make everything lowercase.

→ Stemming :- tasty, tasteful, tasteY all have same meaning.

converting all into one word 'taste'. This is called stemming.

PorterStemmer, Snowball Stemmer } two techniques for stemming.

Snowball Stemmer → more powerful.

→ tokenization :- Breaking sentences into words. eg:- if we use space as delimiter, New York are seperate words.

→ In BoW we are not taking semantic meaning.

eg:- delicious tasty convey same meaning. We use word2Vec to solve this.

Lemmatisation is the algorithmic process of determining the lemma of a word based on its intended meaning. Unlike stemming, lemmatization depends on correctly identifying the intended part of speech and meaning of a word in a sentence, as well as within the larger context surrounding that sentence, such as neighboring sentences or even an entire document.

→ Algorithm combines inflected words into a common word called lemma.

{go, going, went} => 'go' is the lemma given by algo.

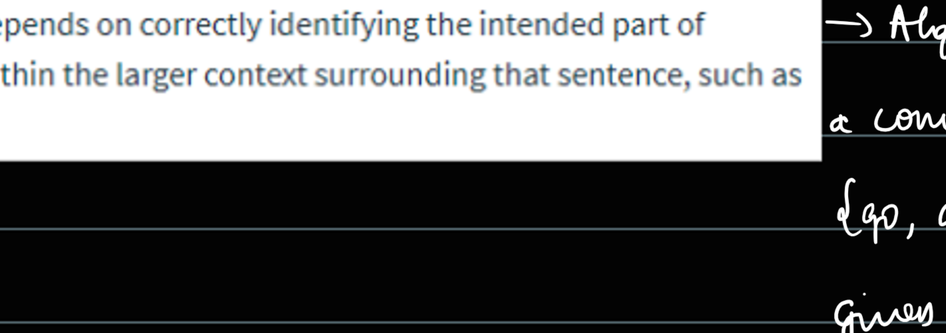
→ NLTK makes use of parts of speech. This is called tagging. Makes sense of context and helps pick the right lemma.

→ Stopwords are not removed in stemming. All words are reduced to base words and the tails are removed.

unigram / Bi-gram / n-gram :-

unigram :- each word is considered as a dimension

bigrams :- pair of words



n = 3 → trigrams

⋮

→ unigram BoW completely discards sequence information

Bigrams, trigrams, ... retains partial sequence informations.

→ But the catch is $n(\text{bigrams}) > n(\text{unigrams})$

→ As n increases, dimensionality increases drastically.

→ But if there are no repeated words in text, #unigrams > #bigrams > #trigrams, ...

TF-IDF:- (term frequency inverse document frequency) (industrialized version of BoW)

r_1: w_1 w_3 w_2 w_5 w_1

N docs { r_2: w_3 w_1 w_4 w_1 w_6 w_2

⋮

⋮

w_1	w_2	w_3	w_4	w_5
2	1	1	0	1

⋮

$$TF(w_i, r_j) = \frac{\#\text{ of times } w_i \text{ occurs in } r_j}{\text{total number of word in } r_j}$$

→ TF-IDF was originally meant for informations retrieval.

→ TF gives how often word occurs. (Probability)

$$0 \leq TF(w_i, r_j) \leq 1$$

IDF:-

D_c = Document corpus { r_1: — w_1 —, r_2: —, ⋮ — w_1 —, ⋮ — w_1 —

$$\rightarrow IDF(w_i, D_c) = \log\left(\frac{N}{n_i}\right)$$

n_i → Number of docs that contains w_i

$$D_c = \{r_1, r_2, \ldots, r_n\}$$

$n_i \leq N \Rightarrow \frac{N}{n_i} \geq 1$

$\Rightarrow \log\left(\frac{N}{n_i}\right) \geq 0$

⇒ As n_i increases, N/n_i decrease. As N/n_i decrease, $\log(N/n_i)$ also decreases as log is monotonic function.

If a word is more frequent in corpus, then its IDF is less. If it's less frequent in corpus, then it's IDF is high

→ Combining both

D_c { r_1 :- ——, r_2 :- ——, ⋮, r_n :- ——

V_1 :- [| | ..... | | ...]

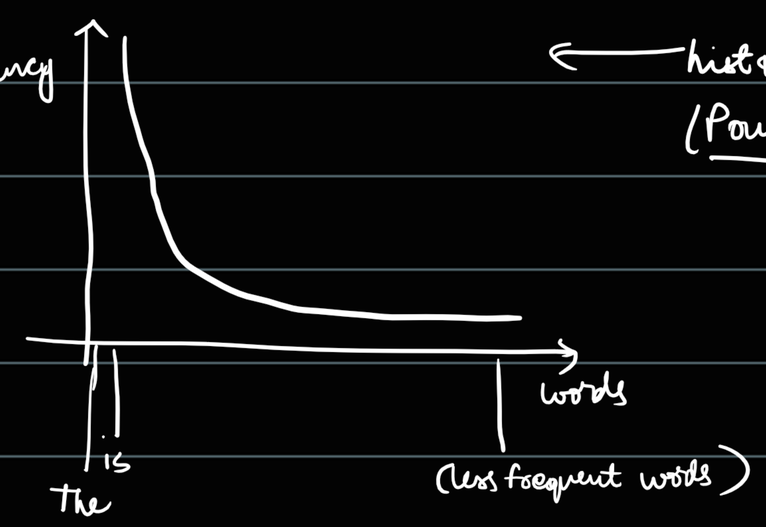
$$w_i = TF(w_i, r_1) * IDF(w_i, D_c)$$

→ Advantage is that we give more importance if → word is more frequent (from TF)
→ word is less frequent (from IDF)

→ Drawback:- Still no semantic importance.

Why do we use log in IDF ? :-

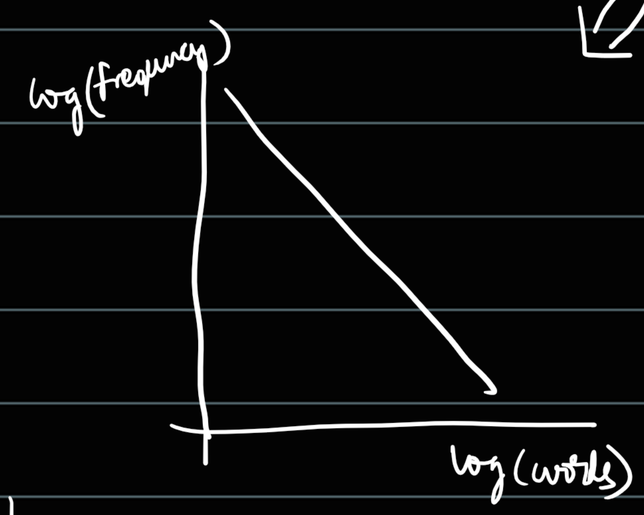
Zipf's law :- frequency



← histogram
(Power law distributions)

↳ If X is a power law distribution, it can be converted to Gaussian Distribution by using transforms like
Box-Cox transform (we take log(X))

When tf idf was introduced, no mathematical explanation was given as to why log was used in idf



→ Another reason :- N/n_i

"the" → almost all docs ≈ 1

"is" → " ≈ 1

⋮

Less frequent pages → if 1 in 1000 → 1000 → ≈ 6.9

$\log(N/n_i)$

≈ 0

"

≈ 0

→ lot of difference. During calculation of TF-IDF, IDF will dominate if there's no log.

→ sometimes if vocabulary is passed explicitly, words might not occur in training phase, but might appear in testing phase. As a result of this N/n_i is going to be ∞. To prevent this, we sometimes do

$$1 + \log\left(\frac{1+N}{1+n_i}\right)$$

(sk-learn implementation)

→ TF :- How frequently word occurs in a document.

IDF :- How rare is the word in the whole corpus. If word is more frequent in the document corpus, it is less important.

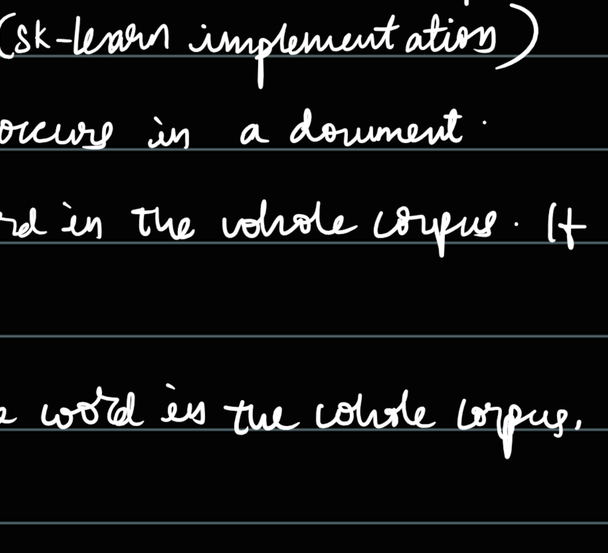
TF-IDF :- Importance of the word in the whole corpus.

Word 2 Vec :- State of the art word to vectors conversion that takes semantic meaning into considerations.

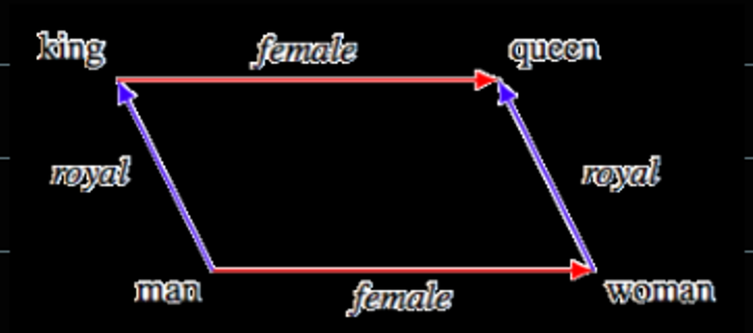
→ The dimensions are usually really high (like 300 ...)

→ It tries to achieve the following :-

① Semantically similar words have closer vectors



② It also tries to keep relationships



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

→ Even for country capitals etc.

→ It does this automatically without being programmed to do so. Came out in 2013.

→ very large data corpus → word2vec → word : vectors

The more the dimensions of vector, the more information rich it is.

→ At it's core, word2vec looks at the neighborhood of words and looks for words with similar neighborhood.

$w_1\ w_2\ w_3\ w_4\ w_5\ w_6$

neighbourhood → $N(w_3)$ ⟹ if $N(w_i) \approx N(w_j)$ then $V_i \approx V_j$

→ Google trained word2vec on it's Google News dataset & this word2vec gives 300 dimension vectors.

Avg word2vec, tfidf weighted word2vec :-

word ⟶ word2vec ⟶ vector.

→ For sentences, sentence2vec exists but there are other methods like average word2vec.

Average word2vec :- $r_1 :- w_1, w_3, w_4, w_2, w_5$

↳ $v_1 :- \frac{1}{n_1}\left[w2v(w_1) + w2v(w_3) + w2v(w_4) + w2v(w_2) + w2v(w_1) + w2v(w_5)\right]$

n_1 = number of words in r_1.

→ Not perfect but simplest method.

tfidf w2v :- first tfidf vector of all words of sentence is calculated.

then each word's w2v is multiplied with it's tfidf score. and whole thing is added & divided by sum of tfidf of all words.

$$\text{tfidf-w2v}(r_1) = \frac{[t_1 * w2v(w_1) + t_2 * w2v(w_2) + t_3 * w2v(w_3) + \cdots + t_n * w2v(w_n)]}{(t_1 + t_2 + t_3 + \cdots + t_n)}$$

t_n → tfidf of word w_n

$$= \frac{\sum_{i=1}^{n} (t_i * w2v(w_i))}{\sum_{i=1}^{n} t_i}$$

if tfidf of all words is 1, then their avg will be 1, then tfidf w2v becomes avg w2v

→ tfidf w2v & avg w2v are weighted schemes and are simple models. They are not always perfect.

→ thought vectors is a technique that is bit advanced. (Seen in deeplearing chapters)

→ Each row of sparse matrix is called sparse vector.

→ If an $n \times m$ matrix has only 'k' non zero elements, sparsity of the matrix $= \frac{(n*m) - k}{(n*m)}$

density of matrix = 1 − sparsity

vallakatla Charith — 9 Votes

How to avoid removing stop word 'not' from text ?

Reply — May 04, 2018 11:19 AM

AppliedAICourse

you can remove the 'not' from stopwords set, syntax: set.remove(element) [stopwords.remove('not')]
will do the work

Reply — May 04, 2018 11:55 AM

Pavan Kalyan Reddy Ramireddy — 5 Votes

Can you explain the parameters of CountVectorizer such as 'preprocessor ' and 'tokenizer '

Reply — Apr 23, 2019 11:22 AM

Applied AI Tech Admin

So far you have seen preprocessing like converting the uppercase words into lower case, stopword removal, removal of punctuations, etc.

preprocess:

When you apply CountVectorizer, it splits the text into features directly into the words on the basis of spaces. So before in case if we need to perform any addition preprocessing on the text before splitting the text into the features we have to define those additional preprocessing steps in the form of a function(say f1) and pass this function to the parameter **preprocess**(ie., preprocess=f1)

tokenizer:

If we want to go for a customized way of tokenizing the text(ie., rather than splitting the text into words on the basis of white space, if we want to split the text in a different and a customized way), then define those new tokenizing steps in the form of a function(say f2) and pass the function name as a value to the parameter **tokenizer**. (ie., tokenizer=f2)

Reply — Apr 23, 2019 13:41 PM

→ min_df = 0.1 → ignore words that appear in less then 10% of documents
= 10 → ignore words that appear in less than 10 documents.

max-df :- Ignore words that appear too frequently. AKA corpus specific stop words.

→ In W2V (Google), words & their stems might not be similar

Vijay Maurya — 27 Votes

Hello Team I have downloaded google model bu this method

1. For download

```
!wget -c "https://s3.amazonaws.com/dl4j-distribution/GoogleNews-vectors-negative300.bin.gz"
```

2. For extraction

```
!7z e GoogleNews-vectors-negative300.bin.gz
```

3. Google model

```
# from gensim.models import KeyedVectors
# w2v_model_google=KeyedVectors.load_word2vec_format('GoogleNews-vectors-negative300.bin',
binary=True)
```

Don't need download anything run it in Google Colab

→ To load big datasets in collab w/o downloading.