Text Processing

Text Processing

Step 1 : Data Preprocessing
Step 2: Feature Extraction

Step 1 : Data Preprocessing

Step 1 : Data Preprocessing

•Tokenization — convert sentences to words

•Removing unnecessary punctuation, tags

•Removing stop words — frequent words such as ”the”, ”is”, etc. that do not have specific semantic

•Stemming — words are reduced to a root by removing inflection through dropping unnecessary characters, usually a suffix.

•Lemmatization — Another approach to remove inflection by determining the part of speech and utilizing detailed database of the language.

Tokenization

Tokenization refers to splitting up a larger body of text into smaller lines, words or even creating words for a non-English language.

The various tokenization functions are in-built into the **nltk** module itself and can be used in programs.

```
import nltk
sentence_data = "The First sentence is about Python. The Second:
about Django. You can learn Python,Django and Data Ananlysis
here."
nltk_tokens = nltk.sent_tokenize(sentence_data)
print (nltk_tokens)
```

```
['The First sentence is about Python.', 'The Second: about
Django.', 'You can learn Python,Django and Data Ananlysis here.’]
```

```
import nltk
word_data = "It originated from the idea that there are readers
who prefer learning new skills from the comforts of their drawing
rooms"
nltk_tokens = nltk.word_tokenize(word_data)
print (nltk_tokens)
```

```
['It', 'originated', 'from', 'the', 'idea', 'that', 'there',
'are', 'readers', 'who', 'prefer', 'learning', 'new', 'skills',
'from', 'the', 'comforts', 'of', 'their', 'drawing', 'rooms']
```

Remove Stopwords

Stopwords are words which does not add much meaning to a sentence. They can safely be ignored without sacrificing the meaning of the sentence.

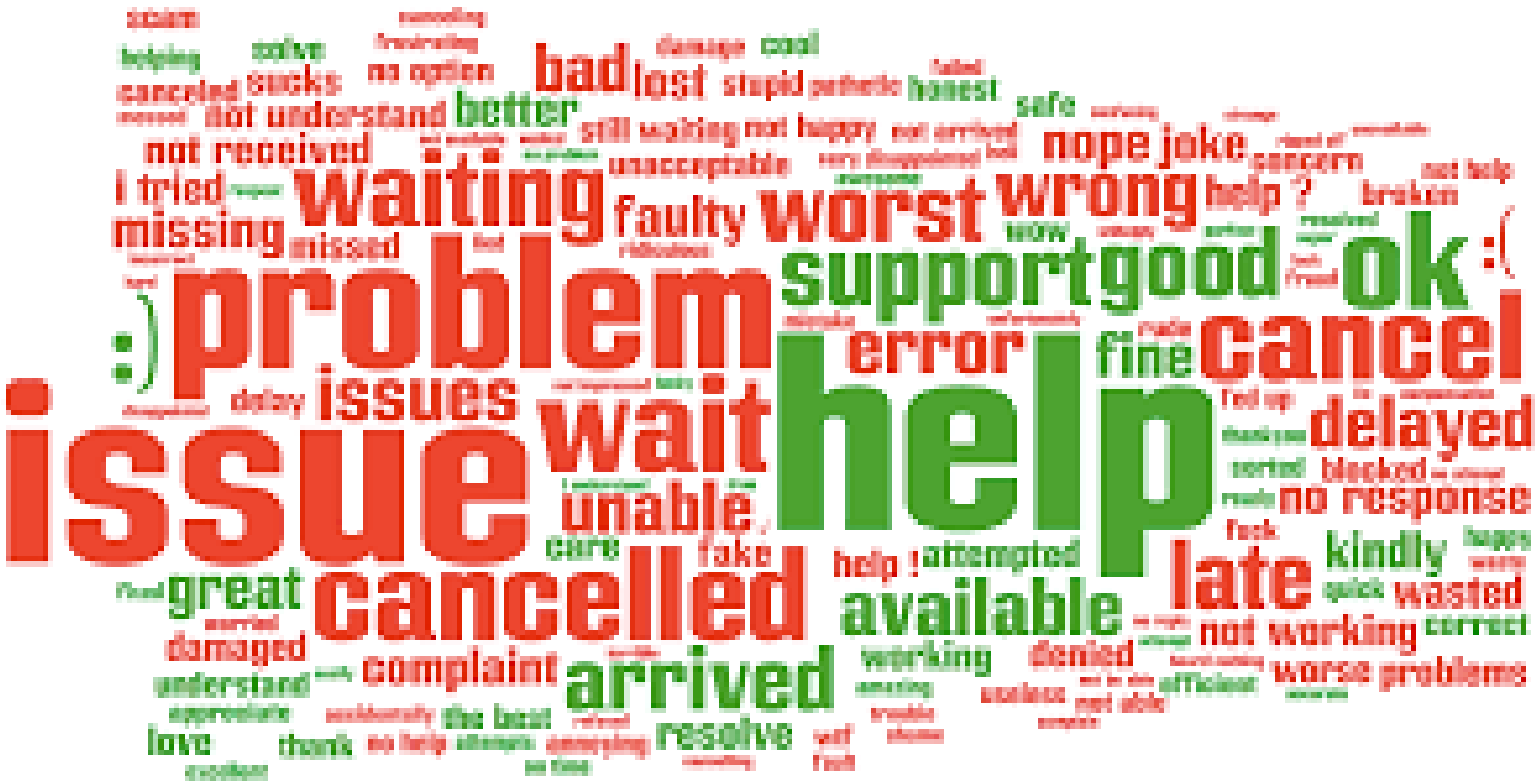
For example, the words like ***the, he, have*** etc. Such words are already captured this in corpus named corpus.

```
from nltk.corpus import stopwords

en_stops = set(stopwords.words('english’))
all_words=['There', 'is', 'a’, 'tree’, 'near’,'the’,
'river’]

for word in all_words:
       if word not in en_stops:
              print(word)
```

```
There
Tree
Near
river
```

Visualization (wordcloud)

Word Clouds (also known as wordle, word collage or tag cloud) are visual representations of words that give greater prominence to words that appear more frequently.



issue
help
problem
waiting
wait
cancelled
cancel
support
good
ok
worst
wrong
error
arrived
available
late
delayed
unable
issues
missing
faulty
bad
lost
better
not received
not understand
fine
great
damaged
complaint
resolve
no response
not working
kindly
wasted
attempted
nope
joke
broken
worse
problems
denied
fake
love
stupid
honest
safe
sucks
unacceptable
i tried
missed
concern
blocked
correct
working
understand
the best
thank
scam
solve
helping
canceled
no option
damage
cool
still waiting
not happy
not arrived
care
delay
help ?
help !
quick
useless
no help
annoying


hypocritical
america
hillary
disheartening
russia
historical
fascism
disappointing
racism
irritating
values
rigged
chameleon
shocking
unpopular
polarized
vote
wild
obama
embarrassing
elected
split
revolutionary
wall
china
trump
unforseen
comical
radical
surprising
disastrous

```
from wordcloud import WordCloud

wrds=''
for msg in data['sms']:
    text=re.sub('[^a-zA-Z]',' ',msg)
    text = text.lower()
    text=text.split()
    text=[word for word in text if not word in
set(stopwords.words('english'))]
    for words in text:
        wrds = wrds+words+' '

wcloud = WordCloud(width=1200, height=720,
random_state=101) .generate(wrds)
```

Stemming

Stemming algorithms work by cutting off the end or the beginning of the word, taking into account a list of common prefixes and suffixes that can be found in an inflected word. This indiscriminate cutting can be successful in some occasions, but not always, and that is why we affirm that this approach presents some limitations.

PorterStemmer uses Suffix Stripping to produce stems.

LancasterStemmer (Paice-Husk stemmer) is an iterative algorithm with rules saved externally. One table containing about 120 rules indexed by the last letter of a suffix.

Word	Porter	Stemmer Lancaster Stemmer
friend	friend	friend
friendship	friendship	friend
friends	friend	friend
friendships	friendship	friend
stabil	stabil	stabl
destabilize	destabil	dest
misunderstanding	misunderstand	misunderstand
railroad	railroad	railroad
moonlight	moonlight	moonlight
football	footbal	footbal

Lemmatization

Lemmatization on the other hand, takes into consideration the morphological analysis of the words. To do so, it is necessary to have detailed dictionaries which the algorithm can look through to link the form back to its lemma.

Note: We also have to specify the parts of speech of the word in order to get the correct lemma. Words can be in the form of Noun(n), Adjective(a), Verb(v), Adverb(r). Therefore, first we have to get the POS of a word before we can lemmatize it.

Word	Stem
It	It
originated	originated
from	from
idea	idea
that	that
are	are
readers	reader
who	who
prefer	prefer
learning	learning
new	new
skills	skill
the	the
comforts	comfort
their	their
drawing	drawing
rooms	room

```
from nltk.stem import PorterStemmer
from nltk.stem import LancasterStemmer

#create an object of class PorterStemmer
porter = PorterStemmer()
lancaster=LancasterStemmer()

#provide a word to be stemmed
print("Porter Stemmer")
print(porter.stem("cats"))
print(porter.stem("trouble"))

print("Lancaster Stemmer")
print(lancaster.stem("cats"))
print(lancaster.stem("trouble"))
```

Document Collection

- A collection of *n* documents can be represented in the vector space model by a term-document matrix.

- An entry in the matrix corresponds to the "weight" of a term in the document; zero means the term has no significance in the document or it simply doesn't exist in the document.

$$\begin{pmatrix} & T_1 & T_2 & .... & T_t \\ D_1 & w_{11} & w_{21} & \dots & w_{t1} \\ D_2 & w_{12} & w_{22} & \dots & w_{t2} \\ \vdots & \vdots & \vdots & & \vdots \\ \vdots & \vdots & \vdots & & \vdots \\ D_n & w_{1n} & w_{2n} & \dots & w_{tn} \end{pmatrix}$$

Step 2: Feature Extraction

In text processing, words of the text represent discrete, categorical features.

How do we encode such data in a way which is ready to be used by the algorithms?

The mapping from textual data to real valued vectors is called feature extraction. One of the simplest techniques to numerically represent text is **Bag of Words.**

Count Vectorizer

Count Vectorizer: The most straightforward one, it counts the number of times a token shows up in the document and uses this value as its weight.

	Jumps	The	brown	dog	fox	lazy	over	quick	the
Doc1	0	1	1	0	1	0	0	1	0
Doc2	1	0	0	1	0	1	1	0	1

```
from sklearn.feature_extraction.text import CountVectorizer

# create a dataframe from a word matrix
def wm2df(wm, feat_names):
# create an index for each row
    doc_names = ['Doc{:d}'.format(idx) for idx, _ in enumerate(wm)]
    df =
pd.DataFrame(data=wm.toarray(),index=doc_names,columns=feat_names)
    return(df)

# set of documents
corpora = ['The quick brown fox.','Jumps over the lazy dog!']

# instantiate the vectorizer object
cvec = CountVectorizer(lowercase=False)

# convert the documents into a document-term matrix
wm = cvec.fit_transform(corpora)

# retrieve the terms found in the corpora
tokens = cvec.get_feature_names()

# create a dataframe from the matrix
wm2df(wm, tokens)
```

Tf-Idf

Term Frequency (Tf)

Tf is a scoring of the frequency of the word in the current document. Since every document is different in length, it is possible that a term would appear much more times in long documents than shorter ones. The term frequency is often divided by the document length to normalize.

$$TF(t) = \frac{Number\ of\ times\ term\ t\ appears\ in\ a\ document}{Total\ number\ of\ terms\ in\ document}$$

Inverse Document Frequency (Idf)

It is a scoring of how rare the word is across documents. Idf is a measure of how rare a term is. Rarer the term, more is the Idf score.

$$Idf(t) = log_e\left(\frac{Total\ number\ of\ documents}{Number\ of\ documents\ with\ t\ in\ it}\right)$$

Thus,

$$Tf - Idf_{score} = Tf * Idf$$

Example:-

Sentence A : The car is driven on the road.
Sentence B: The tractor is driven on the farm.

Word	Tf		Idf	Tf*Idf	
	A	B		A	B
The	2/7	2/7	log(2/2) = 0	0	0
Car	1/7	0	log(2/1) = 0.3	0.043	0
Tractor	0	1/7	log(2/1) = 0.3	0	0.043
Is	1/7	1/7	log(2/2) = 0	0	0
Driven	1/7	1/7	log(2/2) = 0	0	0
On	1/7	1/7	log(2/2) = 0	0	0
Road	1/7	0	log(2/1) = 0.3	0.043	0
Farm	0	1/7	log(2/1) = 0.3	0	0.043

Given a document containing terms with given frequencies:

Kent = 3; Ohio = 2; University = 1

and assume a collection of 10,000 documents and document frequencies of these terms are:

Kent = 50; Ohio = 1300; University = 250.

THEN

Kent:	Tf = 3/3;	Idf = log(10000/50) = 5.3;	Tf-Idf = 5.3
Ohio:	Tf = 2/3;	Idf = log(10000/1300) = 2.0;	Tf-Idf = 1.3
University:	Tf = 1/3;	Idf = log(10000/250) = 3.7;	Tf-Idf = 1.2

```
from sklearn.feature_extraction.text import
TfidfVectorizer

vect = TfidfVectorizer(min_df=3).fit(X_train)
print('Vocabulary len:', len(vect.get_feature_names()))
print('Longest word:', max(vect.vocabulary_, key=len))

X_train_tfidf = vect.transform(X_train)
X_test_tfidf = vect.transform(X_test)
```

Embeddings

Word embedding is one of the most popular representation of document vocabulary. It is capable of capturing **context of a word** in a document, semantic and syntactic similarity, relation with other words, etc.

Word2Vec is one of the most popular technique to learn word embeddings using shallow neural network. It was developed by Tomas Mikolov in 2013 at Google.

Consider the following similar sentences: *Have a good day* and *Have a great day.* They hardly have different meaning.

The one-hot encodings will be as below.
Have = [1,0,0,0,0] a=[0,1,0,0,0] good=[0,0,1,0,0]
great =[0,0,0,1,0] day=[0,0,0,0,1]

Each word occupies one of the dimensions and has nothing to do with the rest. This means 'good' and 'great' are as different as 'day' and 'have', which is not true.

Our objective is to have words with **similar context occupy close spatial positions**.

Word2Vec

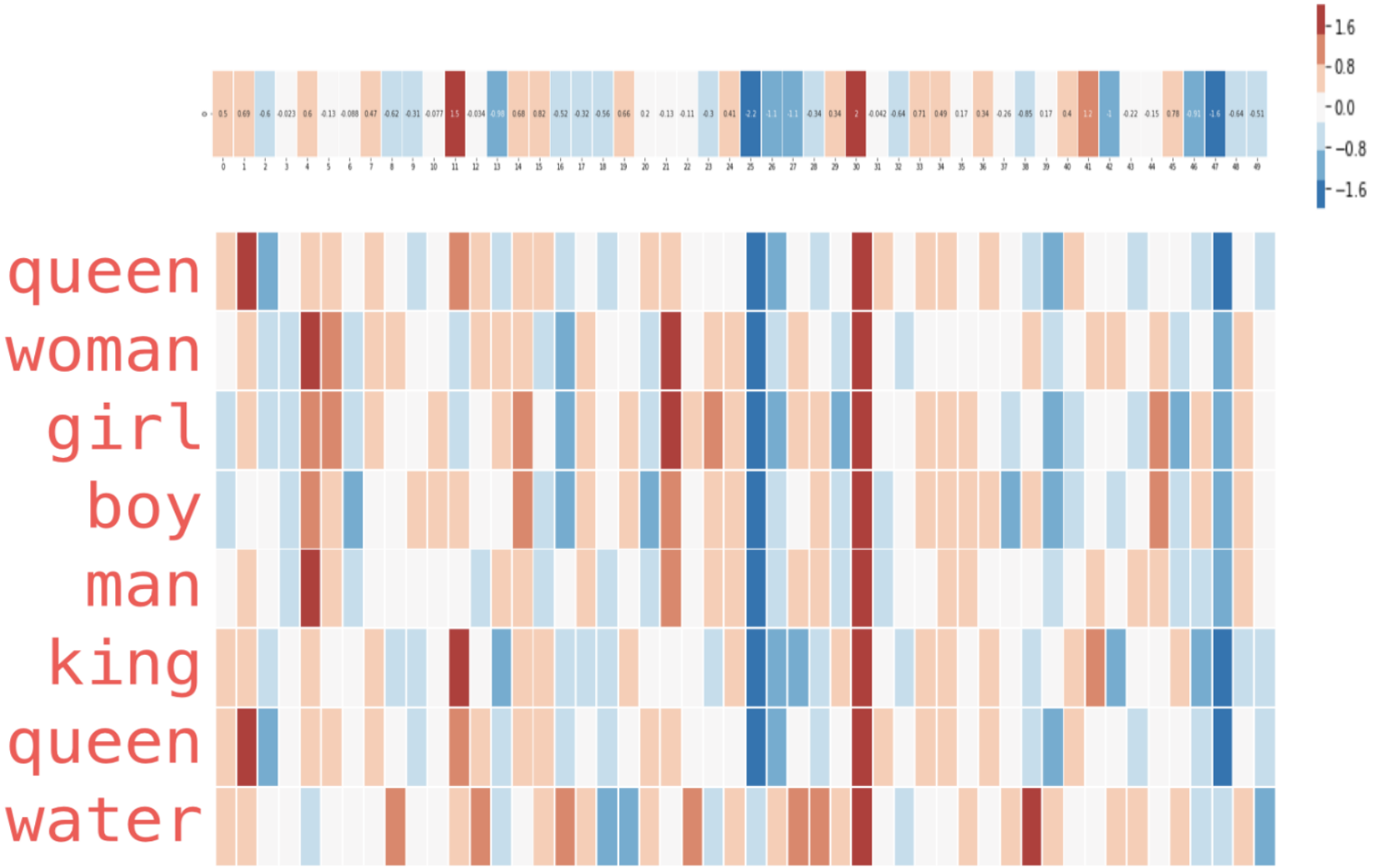
“You shall know a word by the company it keeps” J.R. Firth

Intuitively, we introduce some ***dependence*** of one word on the other words.

- In one hot encoding representations, all the words are ***independent*** of each other, as mentioned earlier.

We train a neural network with a single hidden layer to predict a target word based on its context (neighboring words). The assumption here is that the meaning of a word can be inferred by the **company it keeps**.

In the end, the goal of training with a neural network, is not to use the resulting neural network itself. Instead, we are looking to **extract the weights** from the hidden layer with the believe that these **weights encode the meaning** of words in the vocabulary.



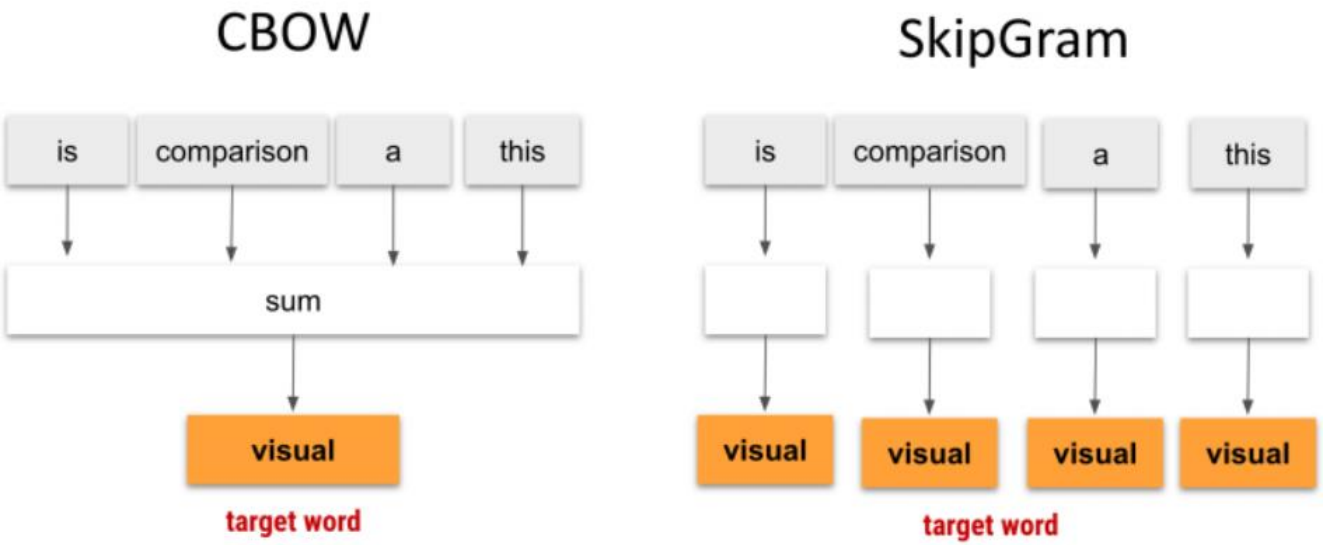
king – man + woman ~= queen

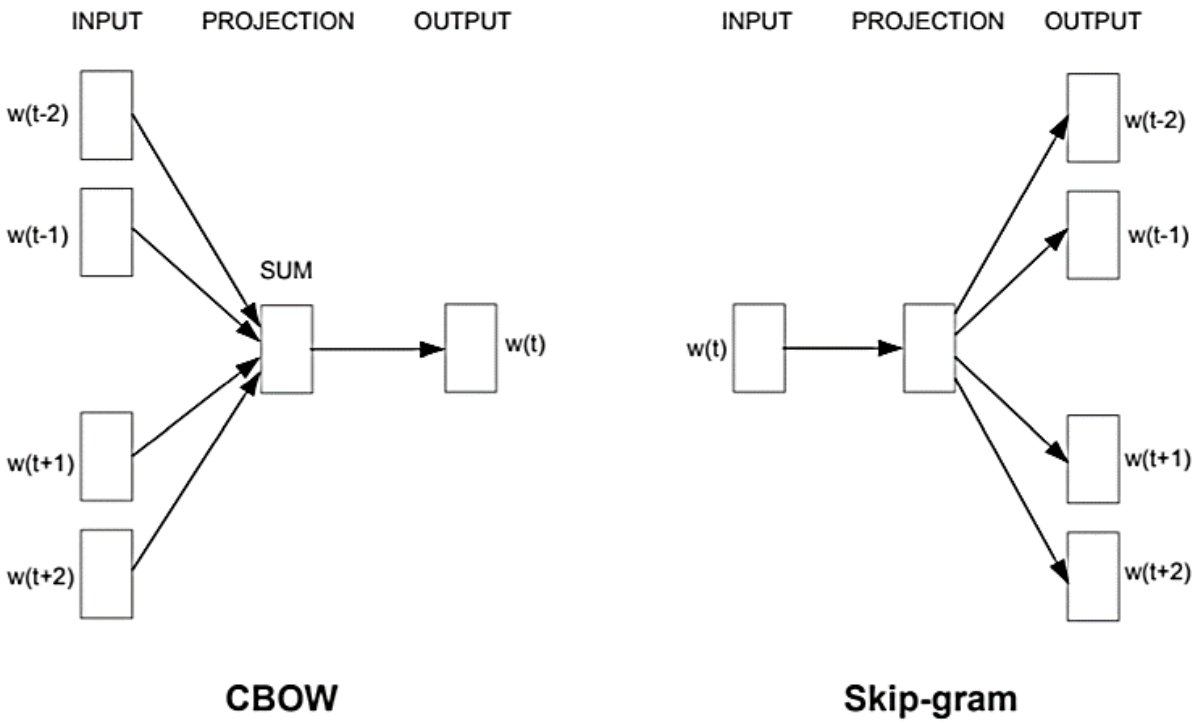
There are two main ways to implement word embeddings using word2vec: Continuous Bag of Words (CBOW), and skipgram.

CBOW:- The CBOW model learns to predict a target word in its neighborhood, using all words. To predict the target word, the sum of the background vectors is used.

Skip-Gram:- Skipgram model predicts the context for a given word. The skip-gram model is the exact opposite of the CBOW model.

Sentence:- This is a visual comparison





In the **CBOW** model, the distributed representations of context (or surrounding words) are combined to **predict the word in the middle**. While in the **Skip-gram** model, the distributed representation of the input word is used to **predict the context**.

CBOW

Thou shalt not make a machine in the likeness of a human mind

Sliding window across running text

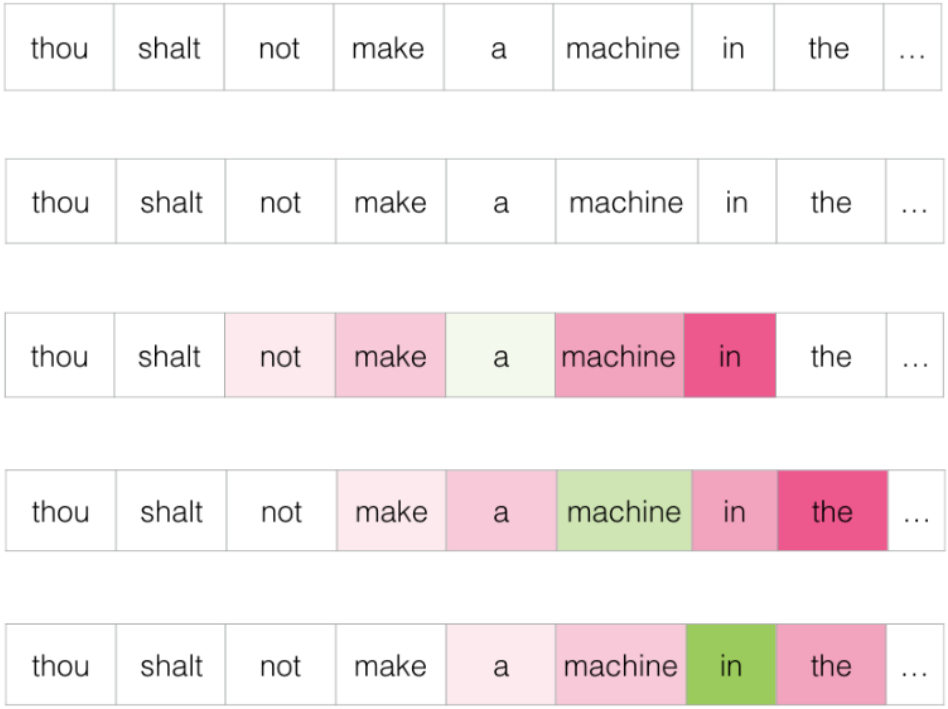
thou	shalt	not	make	a	machine	in	the	…
thou	shalt	not	make	a	machine	in	the	
thou	shalt	not	make	a	machine	in	the	
thou	shalt	not	make	a	machine	in	the	
thou	shalt	not	make	a	machine	in	the	

Dataset

input 1	input 2	output
thou	shalt	not
shalt	not	make
not	make	a
make	a	machine
a	machine	in

Skip-Gram

Thou shalt not make a machine in the likeness of a human mind



input word	target word
not	thou
not	shalt
not	make
not	a
make	shalt
make	not
make	a
make	machine
a	not
a	make
a	machine
a	in
machine	make
machine	a
machine	in
machine	the
in	a
in	machine
in	the
in	likeness

When to use which:

Skip-gram:- works well with small amount of the training data, represents well even rare words or phrases.

CBOW:- several times faster to train than the skip-gram, slightly better accuracy for the frequent words

At the end, since different applications have distinct criteria, the best practice is to try a few tests to see what works best.

GloVe

(GloVe stands for Global Vectors for word representation)

You can derive semantic relationships between words from the co-occurrence matrix.

The advantage of GloVe is that, unlike Word2vec, GloVe does not rely just on **local statistics** (local context information of words), but incorporates global statistics (word co-occurrence) to obtain word vectors. But keep in mind that there's quite a bit of synergy between the GloVe and Word2vec.

- This is created by Stanford University. Glove has pre-defined dense vectors for around every 6 billion words of English literature along with many other general use characters like comma, braces, and semicolons.

There are 4 varieties available in glove:
50d, 100d, 200d and 300d.

- Here d stands for dimension. 100d means, in this file each word has an equivalent vector of size 100. Glove files are simple text files in the form of a dictionary. Words are key and dense vectors are values of key.

Model Overview

The main intuition underlying the model is that ratios of word-word **co-occurrence probabilities** have the potential for encoding some form of meaning. For example, consider the co-occurrence probabilities for target words *ice* and *steam* with various probe words from the vocabulary. Here are some actual probabilities from a 6 billion word corpus:

Probability and Ratio	$k = solid$	$k = gas$	$k = water$	$k = fashion$
$P(k\|ice)$	1.9×10^{-4}	6.6×10^{-5}	3.0×10^{-3}	1.7×10^{-5}
$P(k\|steam)$	2.2×10^{-5}	7.8×10^{-4}	2.2×10^{-3}	1.8×10^{-5}
$P(k\|ice)/P(k\|steam)$	8.9	8.5×10^{-2}	1.36	0.96

As one might expect, *ice* co-occurs more frequently with *solid* than it does with *gas.* Both words co-occur with their shared property *water* frequently, and both co-occur with the unrelated word *fashion* infrequently.

Large values (much greater than 1) correlate well with properties specific to ice, and small values (much less than 1) correlate well with properties specific of steam.

In this way, the ratio of probabilities encodes some crude form of meaning associated with the abstract concept of thermodynamic phase.

The training objective of GloVe is to learn word vectors such that their dot product equals the logarithm of the words' probability of co-occurrence. Owing to the fact that the logarithm of a ratio equals the difference of logarithms, this objective associates (the logarithm of) ratios of co-occurrence probabilities with vector differences in the word vector space.

Because these ratios can encode some form of meaning, this information gets encoded as vector differences as well. For this reason, the resulting word vectors perform very well on word analogy tasks.

GloVe — Intuition behind Loss Function:

Let us generalize the intuition behind GloVe Embeddings — *"ratio of conditional probabilities represents the word meanings"*. The major objective behind any Neural Network model is to 'model' a target function. So, let us consider a function 'F' that models the ratio of probabilities relationship

$$F(w_i, w_j, \overline{w_k}) = \frac{P_{ik}}{P_{jk}}$$

$$w_i, w_j = words\ in\ context$$

$$\overline{w_k} = word\ out\ of\ context$$

$$P_{ik}, P_{jk} = derived\ from\ corpus$$

Cost function

$$J = \sum_{i,j} f(X_{ij})(w_i^T \overline{w}_j - logX_{ij})^2$$

Thank You