~~Week 8 Day 4~~

NLP:-

What is NLP?

→ NLP is a subfield of linguistics, computer science and AI concerned with the interactions b/w computers and human language, in particular how to program computers to process and analyze large amounts of data.

→ In neuropsychology, linguistics, a "natural language" is any language that has evolved naturally in humans through use and repetition without conscious planning. Natural languages can take diff forms such as speech or signing. They are distinguished from formal languages such as used to program or to study logic.

Real world Applications:-

→ Contextual advertisments.
→ Email Clients - Spam filtering.
→ Social Media - removing adult content, opinion mining.
→ Search Engines.
→ Chatbots.

Common NLP Tasks:-

→ Text/doc. classification → Newsarticle categories.
→ Sentiment Analysis
→ Pos Tagging
→ Lang detection & Machine translation
→ Conversational agents. (Siri/Chatbots).
→ Knowledge graph & QA systems. (eg:- Search engine)

→ Text Summarization. ← Inshorts app
→ Topic Modelling. (retrive abstract topics).
→ Text generation.
→ Speech to text. → Next word prediction

Approaches to NLP:-

1) Heuristic Methods. (Rule based) - using Reg. expressions
2) Machine learning Methods.
3) Deep learning Methods.

Heuristic approach eg's:-

→ Regular Expressions
→ Wordnet
→ Open Mind Common sense.

Challenges in NLP:-

→ Ambiguity

eg:- I saw the boy on beach with my binoculars.
I have never tasted a cake quite like that before.

→ Contextual words :- same word but diff. meaning.

eg:- I ran to the store because we ran out of milk.

→ Colloquialism

eg:- Piece of cake. (means very easy).

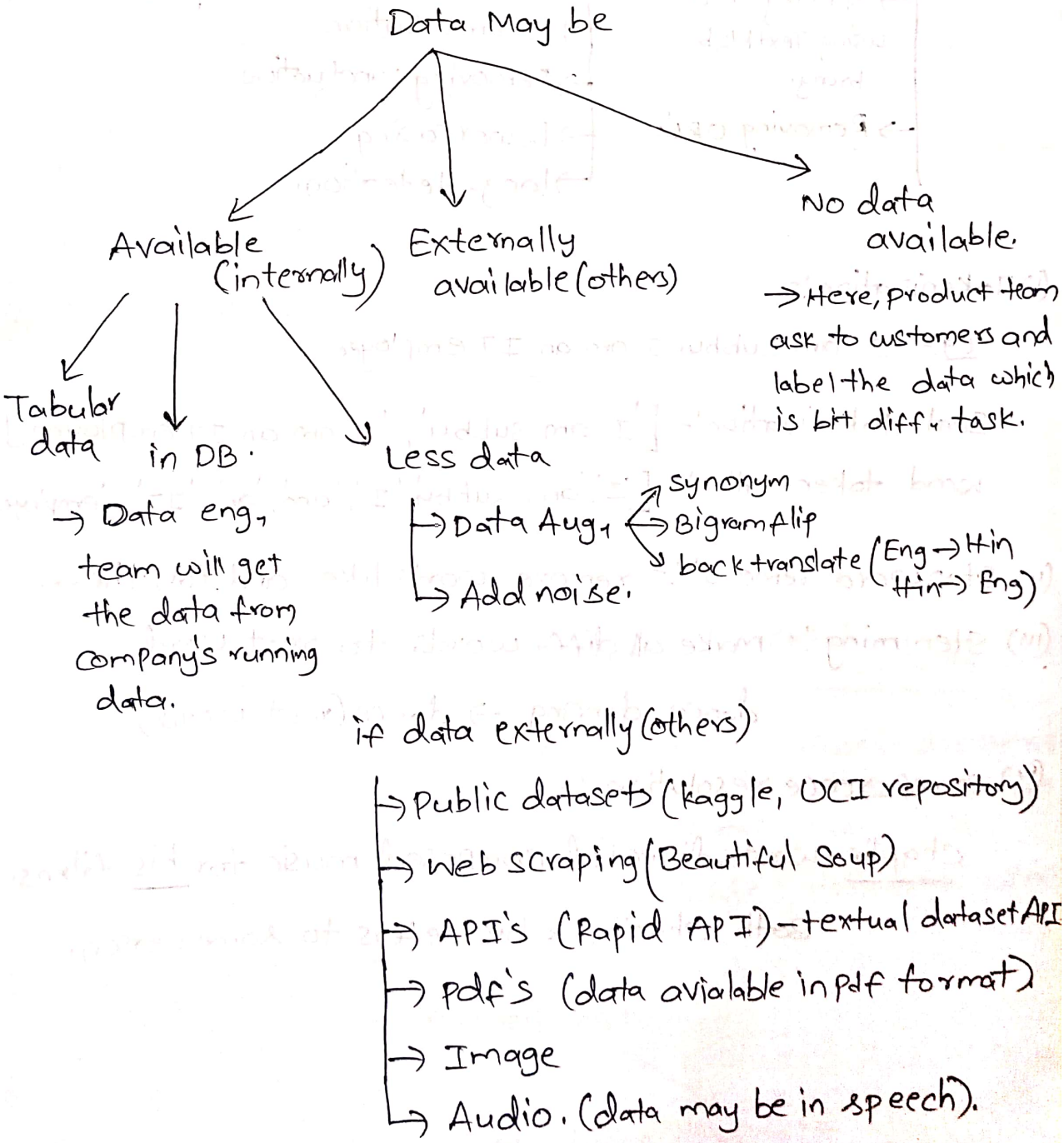
→ Synonyms.
→ Spelling errors.
→ Diversity (so many languages).

NLP Pipeline:

- Data Acquisition
- Text Preprocessing
- Feature Engineering
- Modelling/Evaluation
- Deployment.
- Monitor & Update.

→ NLP pipeline is set of steps followed to buid end to end product.

(1) Data Acquisition:-

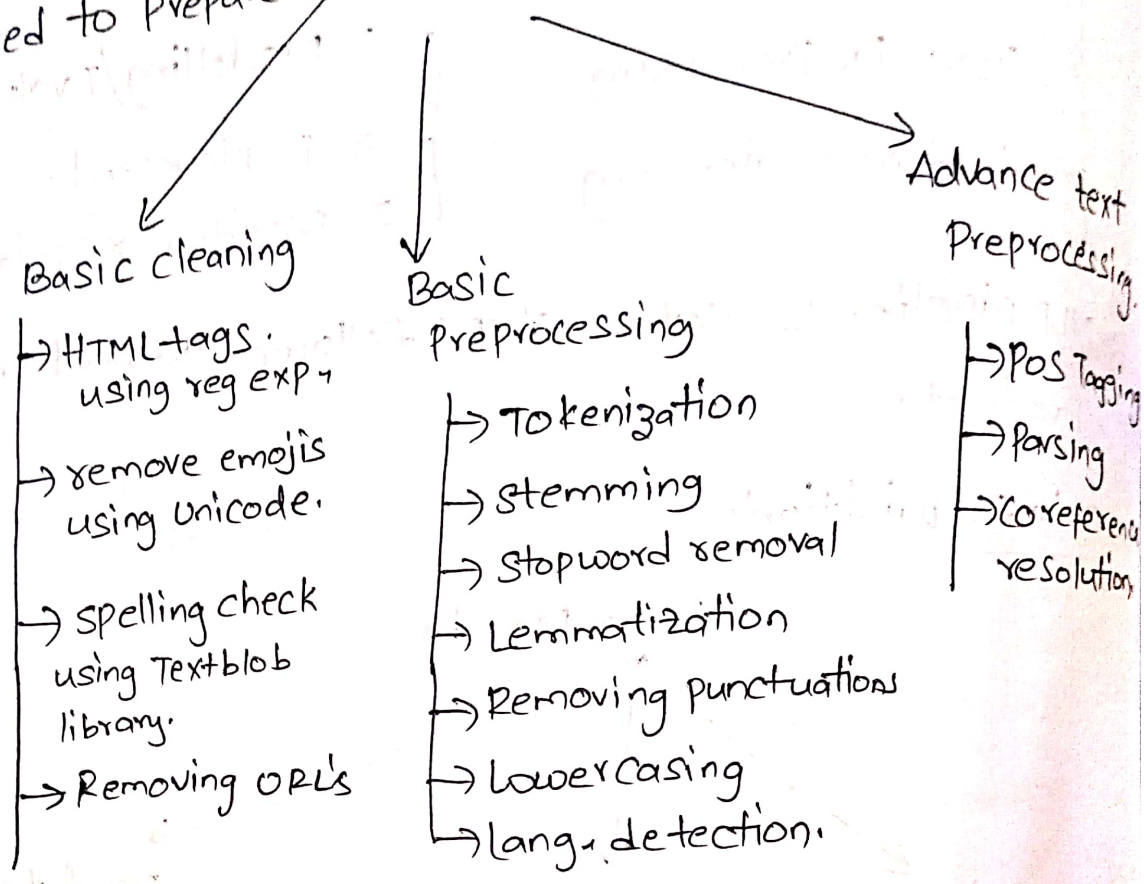


if data externally (others)

- → Public datasets (kaggle, UCI repository)
- → web scraping (Beautiful Soup)
- → API's (Rapid API) - textual dataset API
- → pdf's (data aviable in pdf format)
- → Image
- → Audio. (data may be in speech).

(2) Text preprocessing

→ We need to prepare text.



(i) Tokenization :-

eg:- I am subbu. I am an IT employee.

Sent tokenization :- ['I am subbu', 'I am an IT employee']

word tokenization :- ['I', 'am', 'subbu', 'I', 'am', 'an', 'IT', 'employe

(ii) Stopword removal :- remove words like and, for, the...

(iii) Stemming :- Make all diff words to root word.

dance, dancing → dance (root word)

(iv) Coreference resolution :-

Chaplin wrote, directed, composed music for his films.

Both Chaplin & his refers to same person.

(3) Feature Engineering:-

→ Here, we convert text to numbers.

Text	Sentiment
— —	0
— —	
— —	1
— —	

(50000, 2)

⇒ (Text Vectorization)

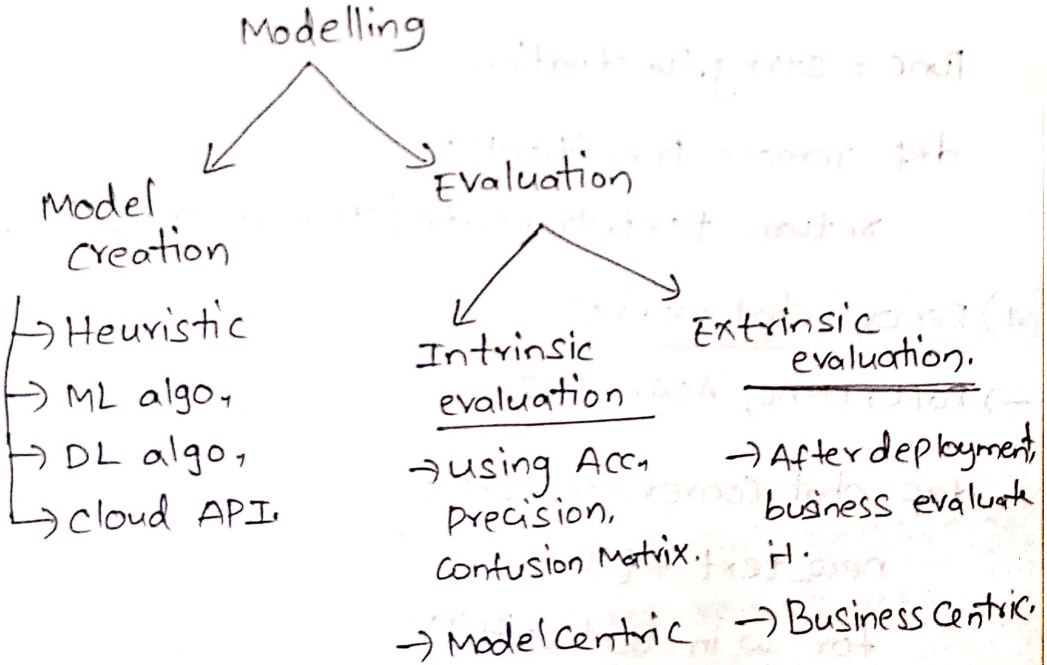
# of +ive	# of -ive	# of neutral	sentiment
5	2	6	0
3	1	6	1

(50000, 4)

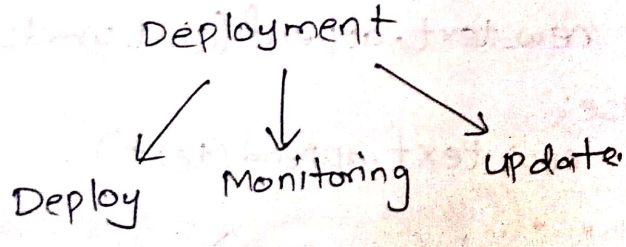
→ Some other techniques includes Bag of Words, Tf Idf, one Hot Encoding, word2vec.

→ Feature engineering techniques depends on the problem that we are working.

(4) Modelling:-



(5) Deployment:-



Text Preprocessing:-

1) Lowercase:-

```
import re
def remove_html_tags(text):
    pattern = re.compile('<.*?>')
    return pattern.sub(r' ', text)

df['review'].apply(remove_html_tags)
```

2) Remove URL's:-

```
def remove_url(text):
    pattern = re.compile(r'https?://\S+|www\.\S+')
    return pattern.sub(r' ', text)
```

3) Remove Punctuation:-

```
import string

Punc = string.Punctuation

def remove_Punc(text):
    return text.translate(str.maketrans(' ', ' ', Punc))
```

4) Remove chat words:-

→ ROFL, LMAO, ASAP, FYI ....

```
def chat_conversion(text):
    new_text = []
    for w in text.split():
        if w.upper() in chat_words:
            new_text.append(chat_words[w.upper()])
        else:
            new_text.append(text)
    return " ".join(new_text)
```

5) Spelling Correction :-

```
from textblob import TextBlob

textblb = TextBlob(incorrect text)
textblb.correct().string
```

6) Removing Stopwords :-

→ a, the, of, and, on, by....

```
from nltk.corpus import stopwords

stopwords.words('english')

def remove_stopwords(text):
    new_text = []
    for word in text.split():
        if word in stopwords.words('english'):
            new_text.append('')
        else:
            new_text.append(word)

    x = new_text[:]
    new_text.clear()
    return ' '.join(x)
```

7) Handling emojis :- (Convert emojis to machine understandable)

```
def remove_emoji(text):
    emoji_pattern = re.compile("["u"\U0001F600-\U0001F64F"]+",
                               flags=re.UNICODE)

    return emoji_pattern.sub(r'',text)
```

→ Instead of removing, we can replace emojis with text.

```
import emoji
Print(emoji.demojize('Python is 😀'))
```

8) Tokenization:-

Process of breaking text document to tokens.

Problems:-

→ Prefix : He gave me $20 ($, (, ")

→ suffix : He ran 10km today (km,), ., !, ")

→ Infix : He stays in U.S.A (shouldn't split U.S.A)

→ sentences may end with . (or) , (or) ? (or) "

```
from nltk.tokenize import word_tokenize, sent_tokenize

sent1 = "I am going to Delhi!"
word_tokenize(sent1)

['I', 'am', 'going', 'to', 'Delhi', '!']

Sent2 = 'I have a Ph.D in A.I'
Sent 3 = 'Mail to us at anup@gmail.com'
Sent 4 = 'A 5km ride cost $20'.

import spacy
nlp = spacy.load('en_core_web_sm')
doc1 = nlp(sent1)
doc2 = nlp(sent2)
for token in doc2:
  print(token)
```

9) Stemming & Lemmatization:-

→ In grammar, "inflection" is the modification of a word to express diff. grammatical categories such as tense, case, voice, gender etc.

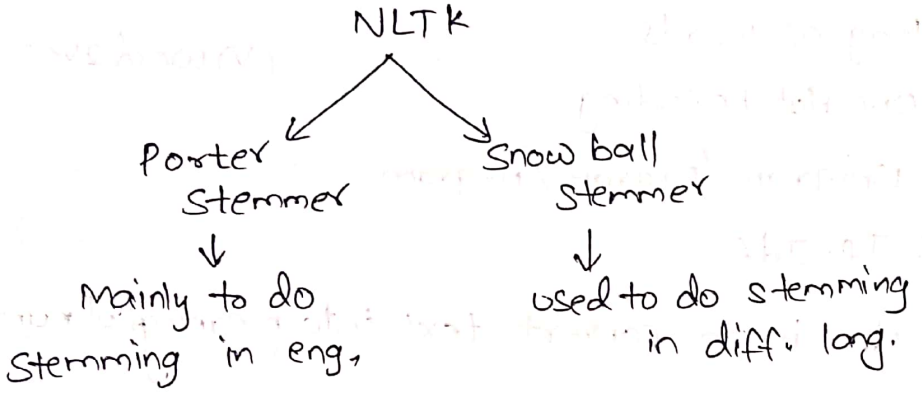
→ (Walk) Walked, Walking, Walks

→ (Play) Played, Playing, Plays, unplayable

Stem word

→ Stemming is the process of reducing inflection in words to their root forms such as mapping a group of words to the same stem even if stem itself is not a valid word

→ Stemming is mainly used in Info. retrieval systems like search engines.




```
from nltk.stem.porter import Porterstemmer
ps = Porterstemmer()
def stem_words(text):
    return " ".join([ps.stem(word) for word in text.split()])
```

→ Problem with stemming is stem word is not a valid word. Instead, we can use Lemmatization.

↓

Here, root word is Lemma.

→ In stemming there are algo's. In lemmatization it will see the words in dictionary i.e. "WordNet lemmatizer."

Lexical dictionary.

```
import nltk
from nltk.stem import WordNetLemmatizer
wordnet_lem = WordNetLemmatizer()
wordnet_lem.lemmatizer(word, pos='v')
```

↓

Parts of Speech is Verb.

→ Problem with lemmatization is, it is very slow.

otherwise go with stemming.

Text Representation :- / Text Vectorization :-

→ After text preprocessing, our main step is to convert text to numbers (or) vectors. We have many approaches.

(I) Bag of words
(II) One Hot Encoding
(III) Unigram / Bigram / N-gram
(IV) Tf-Idf
(V) word2vec.

→ Our core idea is to convert text into meaningful numbers

Common Terms :-

Corpus → Total no. of words in the entire data. (including duplicates)

Vocabulary → Total no. of unique words in our data.

Document → Each review in the data is 'document'.

(I) One Hot Encoding :-

		output
D1	People watch campusX	1
D_2	campusX watch campusX	1
D_3	People write comment	0
D_4	CampusX write comment	0

Corpus (C) = [People, watch, campusX, campusX, watch, campusX, People, write, comment, campusX, write, comment]

Vocabulary (V) = [People, watch, campusX, write, comment]

→ In oneHot Encoding, in your document convert each word to V-dimensional (5) vector.

D_1 = [[1,0,0,0,0], [0,1,0,0,0], [0,0,1,0,0]] → (3,5)

D_2 = [[0,0,1,0,0], [0,1,0,0,0], [0,0,1,0,0]] → (3,5)

Pros:-

→ very intuitive and easy to implement.

Cons:-

→ Sparsity. (If my corpus & voc. is very big, vector representation of each word becomes very big. In that vector, only one value is 1 & rest all are zero. It becomes sparse which leads to overfitting).

→ If size of the doc. vary from other doc. shape will change and model willn't work. "No fixed size".

→ Other problem is 'out of vocabulary'. It means in test data, new word may come which is not in vocabulary.

→ No capturing of Semantic Meaning.

↳ There should be meaning among words /vectors.

	Walk	run	shoe	bottle
Walk	1	0	0	0
run	0	1	0	0
shoe	0	0	1	0
bottle	0	0	0	1

All 4 vectors are equidistant. But bottle is completely diff. compared to three. We are not capturing semantic meaning among vectors.

(2) Bag of Words (BOW):-

Voc = 5

	People	watch	campusX	write	Comment
D1	1	1	1	0	0
D2	0	1	2	0	0
D3	1	0	0	1	1
D4	0	0	1	1	1

D2 → campusX came 2 times & watch came 1 time.

→ In vocabulary, how many times those words came in current doc?

→ We can perform

```
from sklearn.feature_extraction.text import CountVectorizer

cv = CountVectorizer()
bow = cv.fit_transform(df['text'])
print(cv.vocabulary_)
{'campusx': 0, 'comment': 1, 'people': 2, 'watch': 3, 'write': 4}
print(bow[0].toarray())
cv.transform(["campusx watch and write comment"]).toarray()
```

of campusx

[[2, 1, 0, 1, 1]] → It excludes and, of and ignores.

CountVectorizer() – Hyperparameters

- → binary = False (If binary is True, occurrance more than 1 treats as 1).
- → ngram_range = (1,1)
- → max_features (Builds voc, that considers only max features)

Pros:-

→ simple & easy. → Unlike OHE, fixed size problem will'nt be there.

Cons:-

→ sparsity
→ still oov is there.
→ ordering will not consider.
→ can't caputure sentences logic.

eg:- This is very good movie.
This is not very good movie.

} Acc. to Bow, these vectors are very close. But those meanings are completely diff. & opposite

→ This method doesn't preserve word order and sentence meaning will change.

(3) N-grams :-

→ This is quite similar to BOW. Instead of taking one word from VOC, we'll take multiple words.

D1	People watch campusx	1
D2	campusx watch campusx	1
D3	People write comment	0
D4	Campusx write comment	0

People watch | watch campusx |
campusx watch | people write |
write comment | campusx write

D1 → 1 1 0 0 0 0 D3 → 0 0 0 1 1 0

D2 → 0 1 1 0 0 0 D4 → 0 0 0 0 1 1

```
cv = CountVectorizer(n_gram_range=(2,2))
bow = cv.fit_transform(df['text'])
Print(cv.vocabulary_)
```

→ If ngram_range is (1,2), both onegram and bigram will come.
If ngram_range is (1,3), unigram, bigram and trigram will come.

Eg:- This move is very good. (D_1)
This move is not good. (D_2)

Unigram :- [This, movie, is, very, good, not]

$D_1 \rightarrow [1,1,1,1,1,0]$ $D_2 \rightarrow [1,1,1,0,1,1]$

Bigram :- This movie | movie is | is very | very good | is not | not

$D_1 \rightarrow [1\ 1\ 1\ 1\ 0\ 0]$
$D_2 \rightarrow [1\ 1\ 0\ 0\ 1\ 1]$

} compared to unigram, here distance is bit far

Pros :-

→ Able to capture semantic meaning of sentences.

→ No. of words

→ Slows down the algo. because of dimensionality.

→ Still OOV problem exists.

(4) Tf-Idf:-

→ In all previous techniques, we calculated the importance of each word.

→ Here, we'll calculate the frequencies. w.r.t. entire data.

TF → Term Frequency – Freq. of a word in doc.

IDF → Inverse document frequency. → imp. of word in corpus

$$TF = \frac{(\text{No. of occurrences of term } t \text{ in document } d)}{(\text{Total No. of terms in document } d)}$$

$$IDF = \log_e \frac{(\text{Total no. of documents in corpus})}{(\text{No. of documents with term } 't' \text{ in them})}$$

D_1 → People watch CampusX

D_2 → CampusX watch CampusX

D_3 → People write comment

D_4 → CampusX write comment

TF(People, D_1) = 1/3

TF(CampusX, D_2) = 2/3

TF(write, D_4) = 1/3

	IDF
People	log(4/2)
Watch	log(4/2)
CampusX	log(4/3)
Write	log(4/2)
Comment	log(4/2)

→ Multiply TF(t,d) * IDF

	People	Watch	CampusX	Write	Comment
D_1	$\frac{1}{3}*0.3$	$\frac{1}{3}*0.3$	$\frac{1}{3}*0.125$	0	0
D_2	0	$\frac{1}{3}*0.3$	$\frac{2}{3}*0.125$	0	0
D_3	$\frac{1}{3}*0.3$	0	0	$\frac{1}{3}*0.3$	$\frac{1}{3}*0.3$
D_4	0	0	$\frac{1}{3}*0.125$	$\frac{1}{3}*0.3$	$\frac{1}{3}*0.3$

```
from sklearn.feature_extraction.text import TfidfVectorizer
tfidf = TfidfVectorizer()
tfidf.fit_transform(df['text'].toarray())
Print(tfidf.idf_)
Print(tfidf.get_feature_names_out())
```

Why log for IDF?

$$IDF = \left(\frac{N}{n}\right)$$

Let's say I have 10,000 movie reviews. Encyclopedia is the rarest word in all reviews

$$\Rightarrow \left(\frac{10,000}{1}\right) \Rightarrow 10000$$

[IDF will become very big and eventually we'll multiply with TF. IDF dominates over TF].

That's why we take 'log' as both TF and IDF are equally important.

Pros:-

→ very useful in Info. retrieval systems like Search Engines

Cons:-

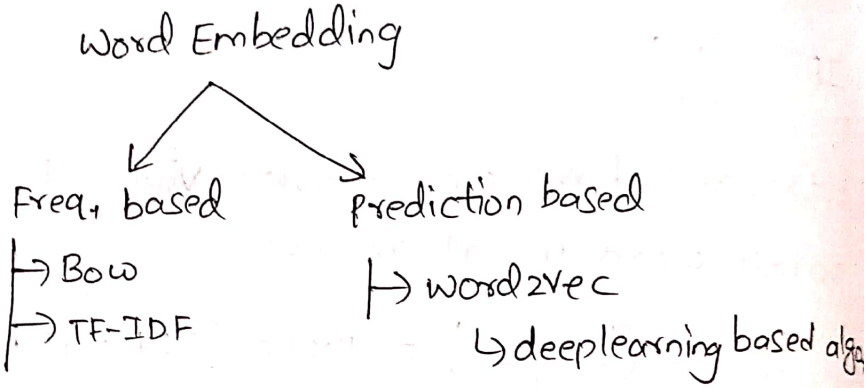
→ sparsity.

→ still OOV exists and no semantic meaning captures.

(5) Word2Vec :-

Word Embeddings :-

→ In NLP, word embedding is a term used for representation of words for text analysis, typically in the form of real valued vector that encodes the meaning of the word such that words are closer in vector space are expected to be similar in meaning.



→ 'word2vec' is a word embedding technique.

→ Word2Vec can capture semantic meaning among words.

→ In Bow & TF-IDF, vectors are high-dimensional. Here, vectors are low-dimensional.

→ Here vectors are dense. (very less zero values will be there)
↳ overfitting won't happen.

→ There is already a pretrained model of word2vec, that was trained on Google corpus containing 3 billion words. This model consists of 300 dimensional vectors for 30 million words.

```
import gensim
from gensim.models import Word2Vec, KeyedVectors

model = KeyedVectors.load_word2vec_format
        ('GoogleNews-vectors-negative300.bin.gz', binary=Tru

model['man'].shape
```

How word2vec works?

→ Basically, word2vec generates some features based on words in vocabulary.

	King	Queen	Man	Woman	Monkey
Gender	1	0	1	0	1
Wealth	1	1	0.3	0.3	0
Power	1	0.7	0.2	0.2	0
Weight	0.8	0.4	0.6	0.5	0.3
Speak	1	1	1	1.	0

features

Corpus/voc
{king, queen, man, woman, monkey}

King → [1, 1, 1, 0.8, 1]

Monkey → [1, 0, 0, 0.3, 0]

King − Man + Woman ≈ Queen

→ Here problem is, features will get generates based on the vocabulary words. If voc. is big, handling these features is a challenging task and that is where Neural N/w's come into picture.

→ NN, generates features & we don't know what the features are

Types of word2vec
- CBOW
- Skip-gram

} Both are shallow NN's.

Above both techniques are similar, but there is some change in architecture.

Fasttext / word2vec / Glove → Trained Models

NLTK / Spacy / gensim → libraries.

CBOW:-

→ Continuous Bag of words.

→ We'll try to solve one fake problem and as a byproduct, we'll get vectors.

Eg:- Watch CampusX for data science.

My o/p vector should be 3-dimensional. Therefore, my window size is '3'.

Watch CampusX for data science.

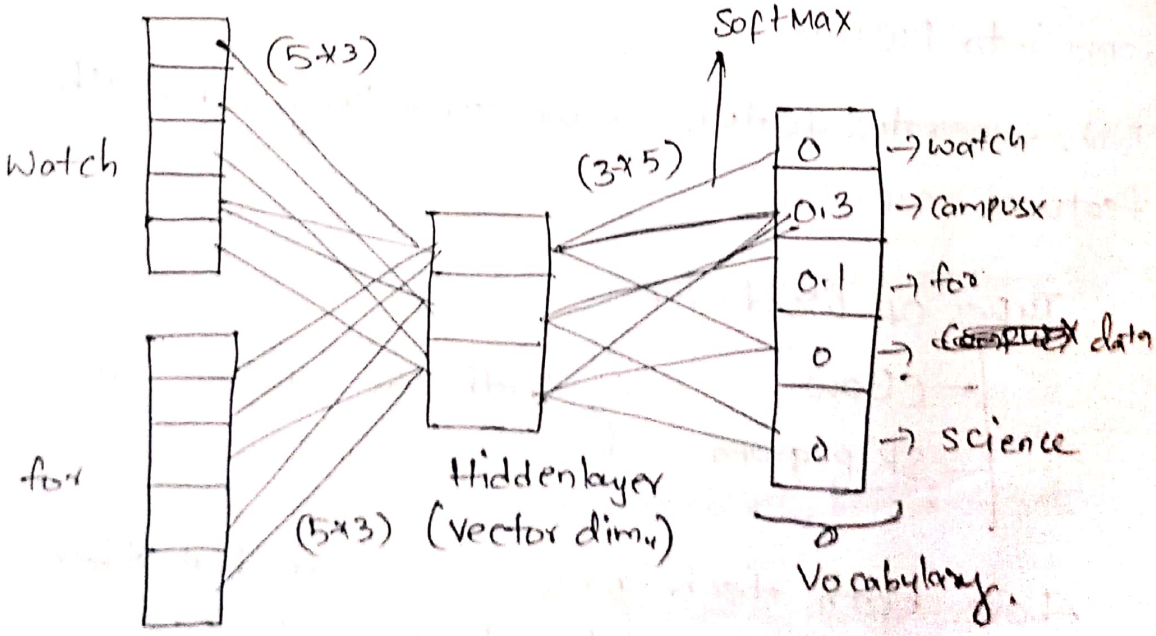
↓ Context word ↓ target word ↓ context word

Here, my dummy problem is, "Given a context word, try to predict target word."

watch → [1 0 0 0 0]

	X	Y
window-1	watch, for	CampusX
window-2	CampusX, data	for
window-3	for, science	data

} Convert these words to numbers using OHE, and give it to Fully Conn. layer



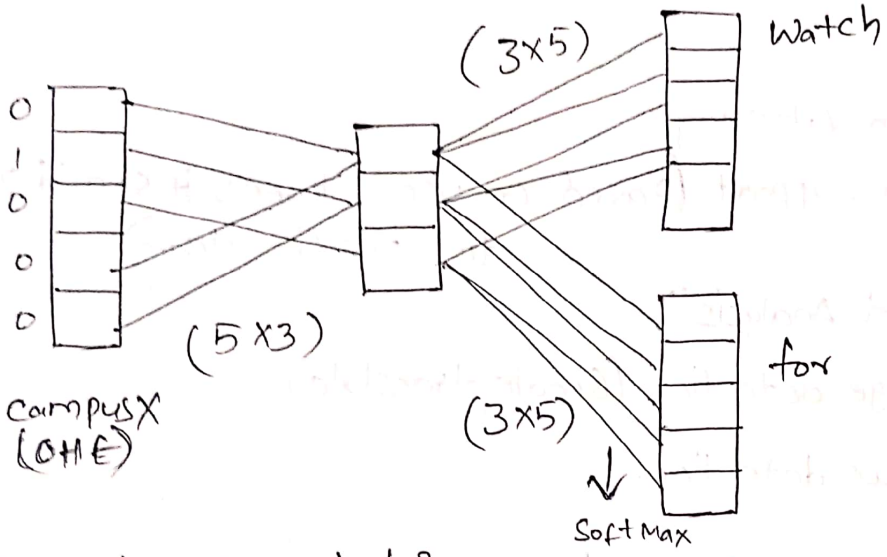
→ DO Forward Prop → make prediction → Calculate loss → Apply backward Prop → Predic o/p using SoftMax.

→ Stop the iteration till the loss gets minimum and get the final weights. These weights are the final vectors for the specific word.

Skipgram:-

→ Architecture will change from CBow.

→ Given the target word, predict context words.



When to use what?

→ If dataset is small, go with CBow and it is accurate.
If you are working with large data, go with skip-gram.

Want to increase embedding quality of word2vec:-

→ Increase the training data.

→ Increase the vector dimension. (# HL's increas

→ Increase window size.

→ CBow can't capture the 2 semantics for single word. Skipgram can capture 2 semantics.

Eg:- word 'apple' can be a company or a fruit

Text Classification:-

→ Classifying the text into different categories.

Newspaper
- → Sports Category
- → Entertaiment Category
- → Business Category
- → Politics.

Applications:-

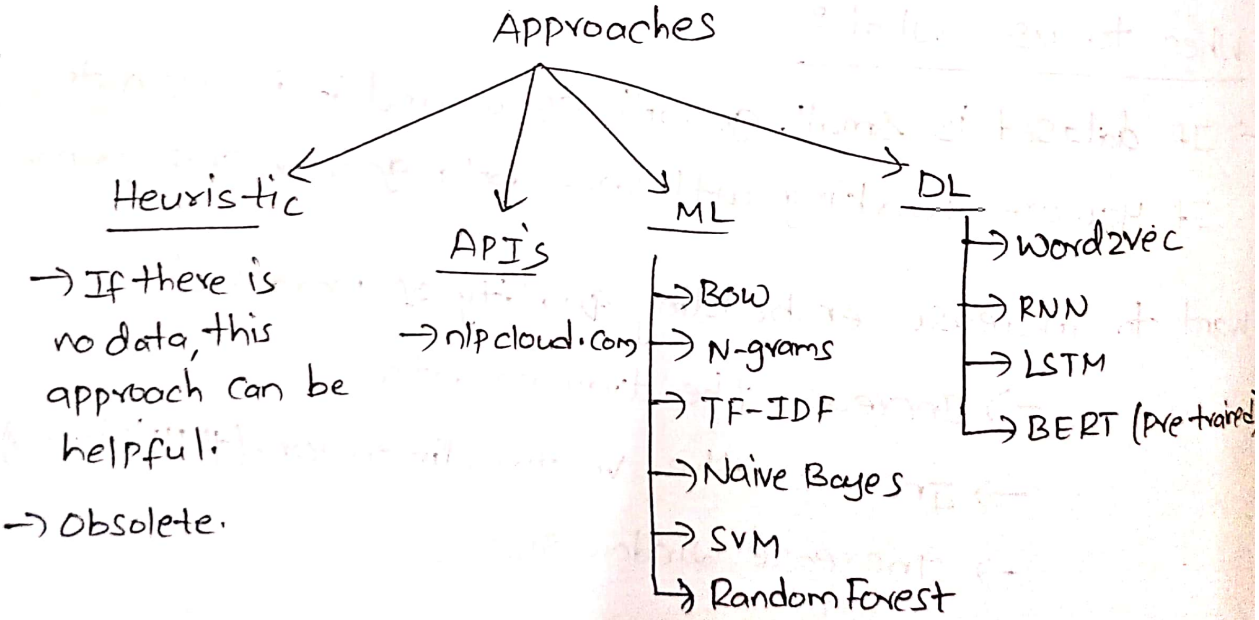
(I) Email Spam filtering

(II) Customer Support (Based on user's tweet, it should route to specific team).

(III) Sentiment Analysis.

(IV) Language detection (Google translate)

(V) Fake News detection.



→ Till now, we have seen vector creation for words. for doc. just add all the vectors of words in that specific doc.

Pos Tagging:-

→ It is very useful in advanced chatbots, Q&A systems.

→ In simple words, Pos-tagging is a task of labelling each word in a sentence with its appropriate parts of speech.

Applications:-

(I) Named Entity Recognition & Topic Modelling.

(II) Question Answering System.

(III) word sense disambiguation.

(IV) Chatbots.

```
import spacy

nlp = spacy.load('en_core_web_sm')

doc = nlp(u'I Play cricket')

doc[0].pos_   # Verb
doc[0].tag_   # Verb, Past-tense:-
spacy.explain('PRP')
```

How POS works?

S (N) Nitish (V) loves (N) campusX E

S (M) Can (N) Nitish (V) google (N) campusX E

S (M) will (N) Ankita (V) google (N) campus X E

S (N) Ankita (V) loves (N) will E

S (N) will (V) loves (N) google E

} will will google campusx

→ We calculate emission probability for each unique word in above data as shown below.

→ In Hidden Markov Model, we'll calculate

(I) Emission probability

(II) Transition probability

(I) Emission Probability:-

	N	M	V
Nitish	2/10	0	0
loves	0	0	3/5
campusx	3/10	0	0
can	0	1/2	0
google	1/10	0	2/5
will	2/10	1/2	0
Ankita	2/10	0	0

→ Calculate probabilities for each word, how many time respective POS ca

Nitish(N) → $\frac{2}{2+3+1+2+2}$ → 2/10

google(V) → $\frac{2}{3+2}$ → 2/5

can(M) → $\frac{1}{1+1}$ → 1/2

(II) Transition Probability:-

	N	M	V	End
Start	3/5	2/5	0	0
N	0	0	5/10	5/10
M	2/2	0	0	0
V	5/5	0	0	0

→ Calculate how many times N, M, V, E came from S, N, M, V

→ Calculate probability row wise

Test data:-

→ Here, we have many combinations.

S — N — N — N — N — E ⇒ 0
(3/5, 2/10, 0, 1/10, 0, 3/10, 0, 5/10)

S — N — N — N — V — E ⇒ 0
(3/5, 0, 0, 0, 0)

S — M — N — V — N — E ⇒ $\frac{2}{5} \times \frac{1}{2} \times 1 \times \frac{2}{10} \times \frac{1}{2} \times \frac{2}{5} \times 1 \times \frac{3}{10} \times \frac{1}{2}$
(2/5, 1, 1/2, 2/10, 1, 2/5, 1, 3/10, 1/2)

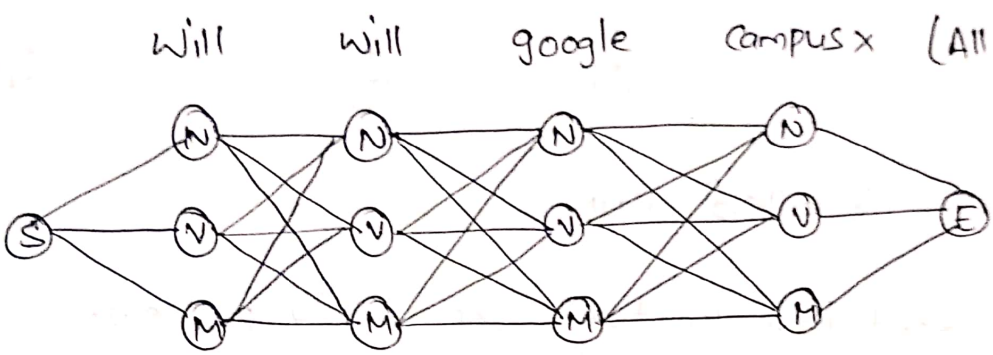
(M) (N) (V) (N)

#0

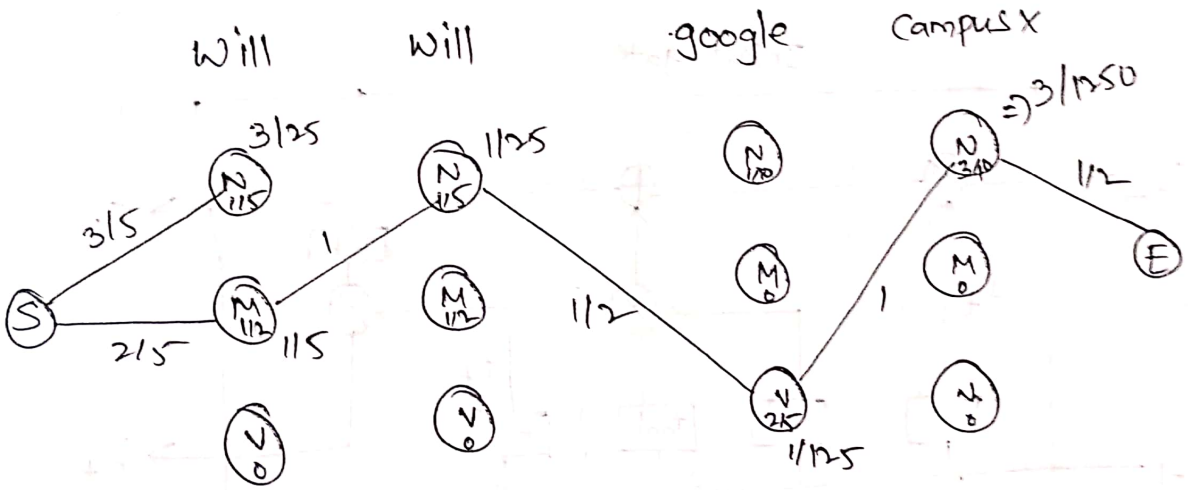
If We have 3 Pos & 4 words in our voc.

$\Rightarrow 3^4 = 81$ combinations. (exponential).

→ If voc. is big means obviously it is not feasible. we need to optimize this. For this, we use "Viterbi Algo."



Viterbi Algo:-



→ Inside the nodes, we have emission probabilities for all Pos.

→ we have to take the path which is getting highest probability

→ once done with the path, backtrack from End to start.

(M) (N) (V) (N) → Final Pos tagging.