NLP

Natural Language Processing

NLP helps machine understand the human language
- both Text and Speech

Important; Machines do not understand the raw text so NLP converts words or sentences into Numerical Representations (vectors), which a machine can learn from.

NLP BUILDING BLOCKS

Task	Example
Tokenization	Breaking text into words: `"This is NLP"` → `["This", "is", "NLP"]`
Stopwords removal	Removing common words like "is", "the", "a"
Stemming/Lemmatization	Reducing words to their root form: `"running"` → `"run"`
Part-of-Speech Tagging	Identifying word roles: `"book"` as a noun or verb
Named Entity Recognition (NER)	Detecting names, places, dates
Text classification (spam, ham)	Spam vs. Not Spam, Sentiment: Positive/Negative
Text generation GPT	Writing text automatically like ChatGPT does

VECTORIZATION PROCESS

🔢 How Text is Turned into Vectors (Simplified):

Method	Description	Example
One-hot encoding	1s and 0s representing presence/absence	`"cat"` → `[0, 1, 0, 0]`
TF-IDF weighted Frequency	Weighted frequency of words	More weight to rare, important words
Word2Vec / GloVe	Words in vector space based on meaning	`"king" - "man" + "woman" ≈ "queen"`
BERT / Transformers	Contextual word embeddings	`"bank"` has different vector in "river bank" vs. "money bank"
Sentence embeddings	Whole sentence as a vector	Used for similarity, summarization, etc.

TFIDF = Term Frequency Inverse Doc. Frequency

NLP PIPELINE

🔧 Example NLP Pipeline (for text classification like sentiment analysis):

1. **Raw text**
   → `"This is a fantastic product!"`
2. **Tokenization** / Lemmatization
   → `["This", "is", "a", "fantastic", "product", "!"]`
3. **Lowercasing**
   → `["this", "is", "a", "fantastic", "product", "!"]`
4. **Stopword removal**
   → `["fantastic", "product"]`
5. **Stemming or Lemmatization**
   → `["fantast", "product"]` or `["fantastic", "product"]`
6. **Vectorization (TF-IDF / Word2Vec / BERT)**
   → `[0.01, 0.34, 0.12, ...]` (a vector)
7. **Model input (e.g. Sentiment Classifier)**
   → Predicts: `Positive` ✅

ROAD MAP

ML

- Step I ; Text Preprocessing I — Cleaning Text
  Tokenisation, Stemming, Stopword Removal
- Step II ; Text Preprocessing II — Text to Vector
  Bow, TFIDF, Unigram, Bigram
- Step III ; Text Preprocessing III — Text to Vector
  Word2Vec, Average Word2Vec → DL

DL

- Step IV ; Neural Networks
  RNN, GRU RNN, LSTM RNN → DL
- Step V ; Text Preprocessing
  Word Embeddings
- Step VI ; Transformers
- Step VII ; BERT

Libraries for Machine Learning

ML = NLTK, Spacy

DL = Tensorflow, Pytorch

Tensorflow → Google

Pytorch → Facebook

Use cases: Text to voice
Text translator
Alexa, GPT etc.

1. TOKENIZATION

breaks down text into smaller units (tokens)

Keywords — outputs in list

- Corpus – Paragraphs
- Documents – Sentences
- Vocabulary – Unique words
- words – Total words

Important we can have words or sentences in tokensations as tokens.

Example; (Corpus) "Tajamul is a Data Scientist. (Sentence 1)
He stays in Hyderabad" (Sentence 2)

1 2 3 4 5 6 7 8 9

Tokens (Sentences) = Sentence 1 and Sentence 2
1 2

words = 9 total words

Vocabulary = 9 unique words (vocabulary = 10)
(unique words)

Tokenisation Practicals

NLTK = Natural Longuage Toolkit

Two types of Tokenize = Sent_tokenize } Sentence
word_tokenize } words

```
corpus="""Hello Welcome,to Krish Naik's NLP Tutorials.
Please do watch the entire course! to become expert in NLP.
"""
```
✓ 0.0s

```
## Tokenization
## Sentence-->paragraphs
from nltk.tokenize import sent_tokenize
```
✓ 0.0s Sent_tokenize (Corpus)

```
## Tokenization
## Paragraph-->words
## sentence--->words
from nltk.tokenize import word_tokenize
```
✓ 0.0s word_tokenize (Corpus)

Wordpunct_tokenize ; considers punctuations as separate words as word_tokenize does not

wordpunct_tokenize = 's = ' ① , s ②
word_tokenize = 's = 's only one

```
from nltk.tokenize import wordpunct_tokenize
```
✓ 0.0s

Considers punctuation as separate word!

```
wordpunct_tokenize(corpus)
```

Imp; Most of the we will be using just

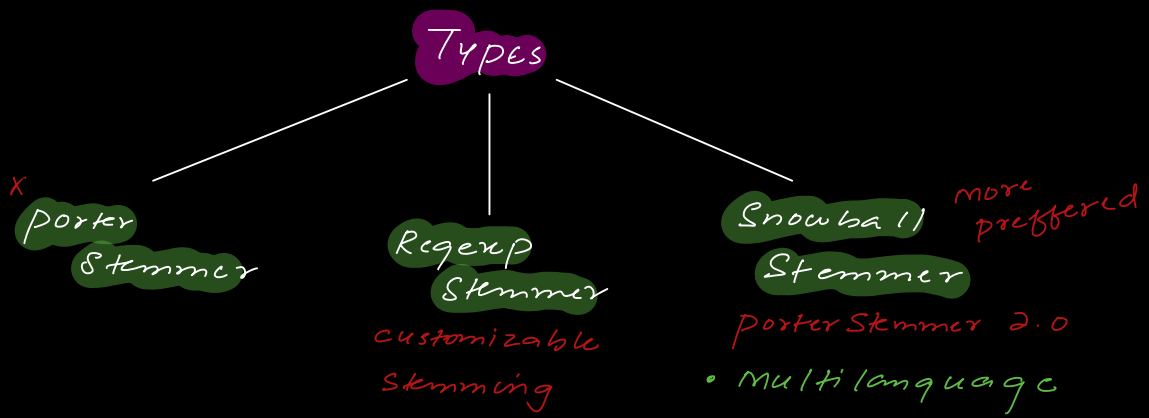
Sent_tokenize()
word_tokenize()

2. Stemming

is the process of reducing word to its stem

example: going, goes, gone } Stem } go

Stemming helps reduce redundant inputs



Imp. Porter stemmer = some words may not be stemmed properly like

Congratulations → Stem → Congratul

Stemming Practicals

Porter Stemmer

```
from nltk.stem import PorterStemmer

stemming=PorterStemmer()
stemming.stem('congratulations')
```

✓ 0.0s

```
'congratul'
```

Regexp Stemmer

```
from nltk.stem import RegexpStemmer

#ing$ means remove last ing
reg_stemmer=RegexpStemmer('ing$|s$|e$|able$', min=4)
reg_stemmer.stem('ingesting')
reg_stemmer.stem('eating')
```

✓ 0.0s

```
'eat'
```

Snowball Stemmer

```
from nltk.stem import SnowballStemmer
snowballsstemmer=SnowballStemmer('english')

# Porter Stemmer
stemming.stem("fairly"),stemming.stem("sportingly")
# Snowball Stemmer
snowballsstemmer.stem("fairly"),snowballsstemmer.stem("sportingly")
```

✓ 0.0s

```
('fair', 'sport')
```

```
# Problems with Snowball Stemmer

# Snowball Stemmer
print(snowballsstemmer.stem("goes"))
print(snowballsstemmer.stem("history"))
```

✓ 0.0s

```
goe
histori
```

→ Problems with Snowball

Imp Snowball Stemmer is better but it is still not reliable for spam classification projects, so we do lemmatization, because it covers vast dict of words.

wordnet dict.

3. LEMMATIZATION

More Accurate

is more like advanced version of Stemming

✅ **In short:**

- **Stemming** = Fast, crude, not always real words. ⟶ No word meaning Imp
- **Lemmatization** = Slower, smarter, returns actual base forms. ⟶ Word Meon Aware

example	Stemming	Lemmatization
Better	Better	Good (More Accurate)

- The output we get after lemmatization is "lemma"

Imp NLTK provides WordNetLemmatizer()

- It also keeps (POS) into consideration

```
lemmatizer.lemmatize("goes",pos='v')
```

Pos (Part Of Speech) Tags	
Noun	n
verb	v
Adjective	a
Adverb	r

```
'''
Noun-n ⟶ Names
verb-v
adjective-a
adverb-r
'''
lemmatizer.lemmatize("going",pos='v')
```

- Use nltk.download('wordnet')

LEMMATIZATION PRACTICALS

```
import nltk
#nltk.download('wordnet') ⟶ Download wordnet

## Q&A,chatbots,text summarization
from nltk.stem import WordNetLemmatizer  Imp
lemmatizer = WordNetLemmatizer()

lemmatizer.lemmatize("fairly",pos='v'),lemmatizer.lemmatize("sportingly")
```

✓ 0.0s

```
('fairly', 'sportingly')
```

Q// which one takes more time wordnet Lemmatizer or Stemming ?

wordnet lemmatizer takes more time because it compares words to wordnet corpus

Imp Mostly used for Q&A, chatbots, text summarize

4. Stopword Removal

where we remove common, insignificant words from the text

example: is, a, the, on, in, for, to

```
stopwords.words('english')
✓ 0.0s
['a',
 'about',
 'above',
 'after',
 'again',
 'against',
 'ain',
```

etc.

```
from nltk.corpus import stopwords

nltk.download('stopwords')
✓ 0.0s
```

All these words won't play huge impact on our spam classification model

```
# Tokenize
sentences=nltk.sent_tokenize(paragraph)

# Lemmatizer
from nltk.stem import WordNetLemmatizer
lemmatizer=WordNetLemmatizer()

## Apply Stopwords And Filter And then Apply Snowball Stemming
for i in range(len(sentences)):
    #sentences[i]=sentences[i].lower()
    words=nltk.word_tokenize(sentences[i])
    words=[lemmatizer.lemmatize(word.lower(),pos='v') for word in words if word not in set(stopwords.words('english'))]
    sentences[i]=' '.join(words)# converting all the list of words into sentences
```

Imp. It is more preffered to have a customized list of stop words.

5. Parts of Speech Tags

Grammatical Role of each word in sentence

e.g., NN = Noun Singular, NNS = Noun plural

```
import nltk
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')

text = "The quick brown fox jumps."
tokens = nltk.word_tokenize(text)
pos_tags = nltk.pos_tag(tokens)

print(pos_tags)
✓ 0.0s
[('The', 'DT'), ('quick', 'JJ'), ('brown', 'NN'), ('fox', 'NN'), ('jumps', 'NNS'), ('.', '.')]
```

Imp If you want to know pos_tag, avoid using lemma / Stemming.

5. Named Identity Diagram

Identify & classify Named Entities into Categories

Example

Entity Type	Examples
Person	"Elon Musk", "Sachin Tendulkar"
Organization	"Google", "UN", "NASA"
Location	"India", "New York", "Amazon River"
Date	"June 29, 2025", "last Friday"
Time	"5 PM", "noon"
Money	"$100", "₹5000", "10 euros"
Percent	"40%", "half"
Miscellaneous	"iPhone", "Python", "FIFA"

Revision

1. **Tokenisation** — sent_tokenize, word_tokenize
   - corpus → sentences / words
2. **Stemming** — SnowballStemmer.stem( )
   - Fast but less Accurate
   - porter,
   - Regexp,
   - Snowball → more preffered
3. **Lemmatization** → Slower & more Accurate
   - wordnet lemmatizer → word_net Lemmatizer
   - based on wordnet_corpus
4. Stop word Removal — import stopwords.

Life Cycle of NLP Project

1. Text Preprocessing
2. Vectorization
3. Machine Learning models.

ONE HOT ENCODING; is a text preprocessing technique used to convert categorical data into numerical format that ML models can understand

WORKING Food is Good D_1

Amazing Food D_2

STEP I; Calculates Vocabulary

Food	is	Good	Amazing
1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1
0	0	0	0

$V = 5$

STEP II; Consider Sentences

$D_1 \longrightarrow \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \longrightarrow 3 \times 4$

$D_2 \longrightarrow \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \longrightarrow 2 \times 4$

I/P No Fixed size hence can't train.

Q. Why is it not preffered for NLP?

- Overfitting (Sparse Matrix)
- No fixed size input
- How words are related is not captured
- OOV = Out of Vocabulary.

Imp; For any ML Algo fixed I/P is required

BAG OF WORDS

V.Important Text classification

is a technique used for vectorization of text

Imp. It is frequency based, more frequently occurring has more value

Imp

Binary BOW vs Normal BOW		
Feature	Normal BOW (Count-based)	Binary BOW (Presence/Absence)
Meaning	Counts frequency of words	Only marks presence (1) or absence (0)
Values	Integers (0, 1, 2, ...)	Binary (0 or 1)

WORKING OF BOW

Text	O/P
He is a good boy	1
She is a good girl	1
Boy and girl are good	1

Lower all the words case

Stopwords

S1 → good boy

S2 → good girl good

S3 → Boy girl good school

Test

Vocabulary	frequency
good	3
boy	2
girl	2

	[good	boy	girl]	O/p
S1	[1	1	0]	1
S2	[1	0	1]	1
S3	[1	1	1]	1

Binary BOW and BOW

{1 and 0} {Count will get updated based on frequency}

ADVANTAGES:

- I/p is fixed
- Simple & Intuitive

DISADVANTAGES:

- Sparse Matrix
  - ↳ Overfitting
- Ordering of words change means the meaning changes
- OOV - Out of Voces

new word in test data X

Difference b/w OHE & BOW?

✅ **Your corrected understanding:**

- ❌ "OHE → vector for every word" → ✅ **Correct**
- ❌ "BOW → vector for each sentence" → ✅ **Correct**
- ✅ But they are not the same kind of vectors — OHE is word-level, BOW is sentence-level.

BOW PRACTICALS

BOW = Count vectorizer

```
## Create the Bag OF Words model
from sklearn.feature_extraction.text import CountVectorizer
## for Binary BOW enable binary=True
cv=CountVectorizer(max_features=100,binary=True)
```

Most frequently occuring words

```
X=cv.fit_transform(corpus).toarray()
```

Imp Max features means, I need 100 words that are more intensive

N GRAMS

continuous sequence of n items (usually words or characters)

💡 **Types of N-grams:**

N	Name	Example (from sentence: "I love NLP")
1	**Unigram**	`["I", "love", "NLP"]`
2	**Bigram**	`["I love", "love NLP"]`
3	**Trigram**	`["I love NLP"]`

You can go higher: 4-grams, 5-grams, etc., but they're less commonly used due to sparsity.

⚙️ **Why Use N-grams?**

- **Capture context:** Unigrams lose context, but bigrams/trigrams capture word relationships (e.g., "New York" vs. just "New").
- **Better feature extraction** for models like Naive Bayes or Logistic Regression in text classification.
- Useful in:
  - Language modeling
  - Text generation
  - Spelling correction
  - Plagiarism detection

INTUITION

Unigrams don't capture context

S_1 = Food is Good = | 1 | 0 | 1 |

S_2 = Food is not Good = | 1 | 1 | 1 |

Vocab — Food, Not, Good

Not Much Difference

Bigram Vocab = Food Good, Food Not, Not Good

S_1 =	1	0	0
S_2 =	0	1	1

→ More Interpretable

N-gram Table

N-gram Range	Meaning
(1, 1)	**Unigrams only** – single words like `["I", "love", "NLP"]`
(2, 1)	❌ **Invalid** – the min value (2) > max (1); will raise an error
(2, 2)	**Bigrams only** – pairs of consecutive words like `["I love", "love NLP"]`
(2, 3)	**Bigrams and Trigrams** – combinations of 2 and 3 words like: `["I love", "love NLP", "I love NLP"]`
(1, 3)	**Unigrams, Bigrams, and Trigrams** – all n-grams from 1 to 3 words: `["I", "love", "NLP", "I love", "love NLP", "I love NLP"]`

Less → more

PRACTICALS

Hyperparameters = n-gram_range, max_features, binary = True

```
## Create the Bag OF Words model with ngram
from sklearn.feature_extraction.text import CountVectorizer
## for Binary BOW enable binary=True
cv=CountVectorizer(max_features=100,binary=True,ngram_range=(2,3))
X=cv.fit_transform(corpus).toarray()
```

(1, 1) = Unigrams only

(1, 2) = Uni, bi

(1, 3) = Uni, bi, trigrams

TF-IDF

Term Frequency - Inverse Document Frequency

→ is a technique to convert text data into vectors

BOW → word frequency

TF IDF → weighted frequency

	BOW	TF-IDF
🎯 **Focus**	Frequency only	Frequency **and** uniqueness
⚖️ **Importance Handling**	Treats all words equally	Penalizes common words, boosts rare but important words
Imp. 🔴 **Stop Words Effect**	Common words dominate vector	Common words are down-weighted even if not removed
🧠 **Captures Meaning?**	❌ No context, just counts	❌ No deep meaning, but better than raw counts
📊 **Vector Output**	Sparse vectors with word counts	Sparse vectors with **weighted scores**
🛠️ **Example Tool**	`CountVectorizer` in sklearn	`TfidfVectorizer` in sklearn extraction

INTUITION

$$\text{Term Frequency} = \frac{\text{Frequency of words in sentence}}{\text{No. of words in sentence}}$$

$$IDF = \log_e\left(\frac{\text{No. of sentences}}{\text{No. of sentences cont. the word}}\right)$$

S_1 = good boy

S_2 = good girl

S_3 = boy good girl

TF.

	S_1	S_2	S_3
good	1/2	1/2	1/3
boy	1/2	0	1/3
girl	0	1/2	1/3

Imp Two Matrix products

- hadamard (element wise)
- Dot

IDF

good	$\log_e(3/3) = 0$
boy	$\log_e(3/2)$
girl	$\log_e(3/2)$

FINAL TF-IDF ⟶ TF x IDF

	good	boy	girl
S_1	$1/2 \times 0$	$1/2 \times \log_e 3/2$	$1/3 \times \log_e 3/2$
S_2	$1/2 \times 0$	0	$1/3 \times \log_e 3/2$
S_3	0×0	$1/2 \times \log_e 3/2$	$1/3 \times \log_e 3/2$

VV Imp In TF-IDF, the word importance is captured which means the word which occurs in every sentence is given less value, and more value to uniqueness

🧠 How TF-IDF Still Helps:

Even **after removing stopwords**, your data will still have many words that:

- Appear frequently in **one document** (important)
- Appear across **many documents** (less informative)

TF-IDF gives more weight to the first kind.

Advantages

word importance is captured

Disadvantages

- Sparsity exists
- No semantic context

TFIDF is better than BOW

PRACTICALS

```
from sklearn.feature_extraction.text import TfidfVectorizer

tfidf=TfidfVectorizer(max_features=100)
X=tfidf.fit_transform(corpus).toarray()
```

WITH N GRAMS

N-Grams

```
tfidf=TfidfVectorizer(max_features=100,ngram_range=(2,2))
X=tfidf.fit_transform(corpus).toarray()
```

WORD EMBEDDINGS

Dense

Semantic Aware

Word Embeddings are dense vector representations of words, similar words have similar vectors

📌 In Short:

Technique	Word Meaning Captured?	Vector Type	Example
BoW	❌ No	Sparse (1s/0s)	[0,1,0,0,0,1,0...]
TF-IDF	❌ No	Sparse (weighted)	[0.3, 0.0, 0.7, ...]
Embeddings	✅ **Yes**	**Dense** (float)	[0.12, -0.33, 0.88...]

💡 What's special about Word Embeddings?

They **capture semantic meaning**, so:

- `"king" - "man" + "woman" ≈ "queen"`
- `"happy"` and `"joyful"` are **close** in vector space
- `"apple"` is close to `"fruit"`, not `"car"`

Word meaning is captured

V. V. Important

✅ Can you say BoW, TF-IDF, and OHE are types of word embeddings?

> ❌ **Technically, no. They are not true word embeddings.**
> But...
> ✅ They are **text vectorization techniques** — they convert words into vectors.

Imp

🔬 What is true word embedding?

✅ A **word embedding** is a **dense**, **learned**, and **semantic-aware** representation of a word — typically trained using large text corpora.

🔄 Let's break it down:

Technique	Converts Text to Vectors?	Captures Meaning (Semantics)?	True Embedding?
One-Hot Encoding (OHE)	✅ Yes	❌ No (all words equally distant)	❌ No
Bag of Words (BoW)	✅ Yes	❌ No	❌ No
TF-IDF	✅ Yes	❌ No (weights importance, not meaning)	❌ No
Word2Vec, GloVe, BERT	✅ Yes	✅ **Yes** (semantic relationships)	✅ **Yes**

Frequency

Semantic Awareness; means model understands the relationship b/w the words based on

- their meaning
- not their frequency / count

Google word2vec
↳ 3 billion words

WORD2VEC

- Dense vectors
- Semantic Aware

is a deep learning model — which helps to represent each word as a dense vector, where words with similar meaning have similar vectors.

Example:

After training Word2Vec, you might get:

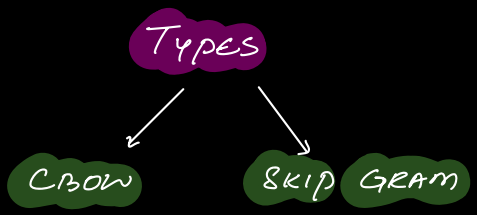
- `vector("king") ≈ [0.25, 0.51, -0.33, ...]`
- `vector("queen") ≈ [0.26, 0.49, -0.30, ...]`

They're **numerically close** because their **meanings and usage are similar**.

Why Word2Vec is powerful

Unlike BoW or TF-IDF, Word2Vec:

✅ **Understands context**
✅ **Captures meaning & similarity**
✅ Produces **dense, low-dimensional vectors**



How Word2Vec Works (Two Models)

Model	Idea
CBOW (Continuous Bag of Words)	Predict the current word from surrounding context
Skip-gram	Predict surrounding words from the current word

> 📌 Both models are trained on a large corpus using neural networks.

Imp ! In word2vec, each word is represented by vector of 300 Numbers

These numbers **capture hidden features** like:

- Gender
- Royalty
- Profession
- Power
- And many others (not named, just learned from usage)

v. Imp → Vector Representation

Let's say; Vocab = Man, Woman, King, Queen

Feature Represent.

	Man	Woman	King	Queen
Gender	-1	+1	-0.92	+0.92
Royal	-0.01	+0.02	0.95	+0.96

A very popular example;

King − Man + Woman = Queen

Removes Man
Left with Royal

Royal + Woman ≈ Queen

MATHEMATICAL INTUITION

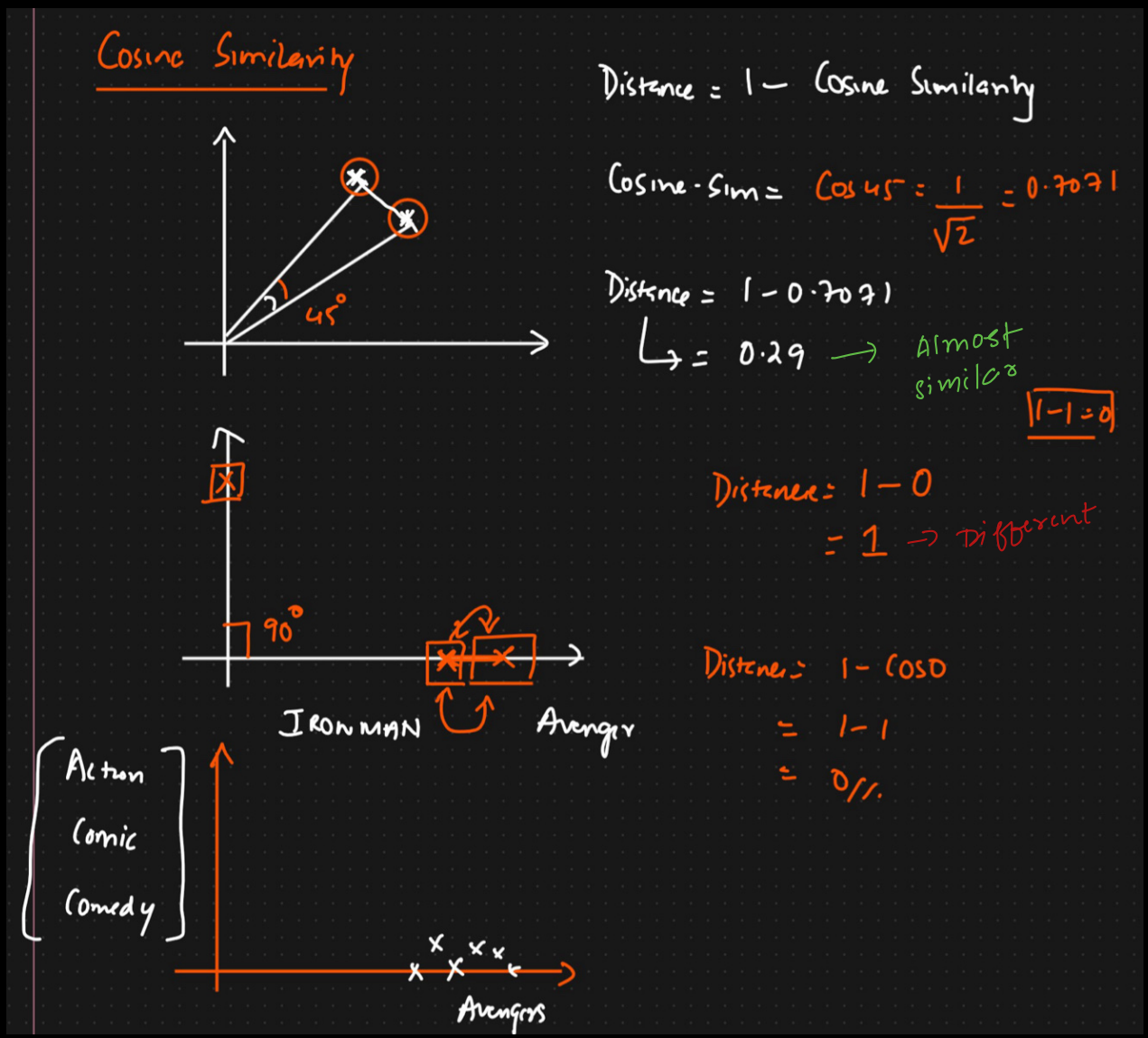
king [0.95, 0.96] Man [0.95, 0.90]

Queen [-0.96, -0.95] woman [-0.98, -0.95]

Imp To understand distance b/w 2 vectors, we need **cosine similarity**

Cosine Similarity

Distance = 1 − Cosine Similarity (Angle b/w 2 vectors)



Before getting into CBOW intuition, lets clear our heads with → DL concepts
Activation fns
Optimizers

manual ML vs DL → Black Box

🧠 Basic Definition:

Concept	Machine Learning (ML)	Deep Learning (DL)
What is it?	A branch of AI that uses algorithms to learn from data	A subset of ML that uses **neural networks**
Inspired by	Statistics and math	Human brain (neurons and layers)
Feature Handling	Manual feature engineering often required	Learns features automatically

🎯 Key Differences:

Aspect	Machine Learning (ML)	Deep Learning (DL)
Data Need	Works well with **small/medium data**	Needs **large data** to perform well
Model Type	Decision Trees, SVM, KNN, etc.	Neural Networks (ANN, CNN, RNN)
Feature Engineering	✅ Manual (you design features)	❌ Not needed (learns features automatically)
Training Time	Fast	Slow (requires GPUs)
Hardware Needed	CPU is usually enough	GPU/TPU preferred for speed
Interpretability	Easier to understand	Hard to interpret (black box)

📦 Examples:

Problem	Use ML?	Use DL?
House price prediction	✅ ML (Linear Reg)	❌ DL not needed
Spam detection (emails)	✅ ML (Naive Bayes)	✅ DL (LSTM for text)
Image classification	❌ ML fails	✅ DL (CNN shines here)
Chatbot/Voice assistant	❌ ML limited	✅ DL (transformers, RNNs)

> - **"ML is like teaching a child with rules ("If weather = rain → carry umbrella")"**
> - **"DL is like letting the child observe and learn everything by themselves — they just figure it out!"**

Types of DL ;

Recurrent Neural Net.

📑 Summary Table

Feature	ANN (Word2vec)	CNN	RNN (Keeps memory)
Full Form	Artificial Neural Network	Convolutional Neural Network	Recurrent Neural Network
Best for	Tabular data	Images, videos	Time series, text, audio
Has memory?	❌ No	❌ No	✅ Yes
Sequence-aware?	❌ No	❌ No	✅ Yes
Key component	Dense (fully connected) layers	Convolutional + pooling layers	Loops & hidden states
Data requirement	Medium	High	Very high (needs sequences)

📌 Example Use Cases:

Task	ANN	CNN	RNN
House price prediction	✅	❌	❌
Image classification	❌	✅	❌
Stock price prediction	❌	❌	✅

ANN
- Basic form of Neural Network
- No Memory — Basic Word2vec
- Tabular Data, Simple classification problems

🎯 3. Loss Function -- What the model tries to minimize

❓ Why do we need a loss?

It tells the model:

> "Hey, your prediction is off by this much — fix it!"

📦 In Word2Vec CBOW:

- The loss is high if the predicted word is far from the actual word
- It gets lower as the model **learns better word associations**

✅ Common Loss in Word2Vec:

- **Cross-Entropy Loss** (when predicting words)
- **Negative Sampling Loss** (when using efficiency tricks)

⚙️ 4. Optimizers — How the model learns better over time

❓ What do optimizers do?

They update the weights to **reduce the loss** — like giving the model feedback.

🏋️ Most common optimizer:

- **SGD (Stochastic Gradient Descent)** — basic optimizer used in CBOW
- Others: Adam, RMSProp (used in advanced models)

⚙️ Example in CBOW:

1. It sees: `["the", "on", "the"]`
2. Predicts: `"mat"`
3. Compares prediction with actual word
4. Computes loss
5. **Optimizer updates weights** to get better next time

🛠️ What is an Optimizer?

An **optimizer = smart version of gradient descent**
It tells the model:

> "Here's how to update your weights using gradient descent (and some extra tricks) to learn faster and better."

🔁 So:

🔸 **Gradient Descent = The basic idea**

➡️ Move weights opposite to the gradient of the loss.

🔸 **Optimizer = An algorithm that uses gradient descent (and often improves it)**

Q1 Difference b/w RNN and Transformers ?

Feature	RNN	Transformer
Architecture	Sequential	Attention-based
Processing	Step-by-step	All at once (parallel)
Speed	Slow	Fast
Long Context	Struggles	Handles well
Use Cases	Time-series, speech	NLP, translation, chatbots
Examples	LSTM, GRU	BERT, GPT, T5

> ✅ RNN = good for short, sequential data
> ⚡ Transformer = best for long, complex text tasks

Q1 Difference b/w forword & back propagation ?

🧠 1. Forward Propagation (Prediction Phase)

This is when the input data **flows through the network** to produce an output (prediction).

Steps:

- Input data is passed into the model.
- Each layer applies weights and bias, then an activation function.
- The final output (prediction) is produced at the last layer.

📦 Think of it like a **factory line** where raw material (input) goes through various stations (layers) and gives a final product (output).

🔁 2. Backward Propagation (Learning Phase)

This is when the model **learns from the error** by adjusting weights to reduce the loss.

Steps:

- Calculate loss (difference between prediction and actual value).
- Compute gradients of the loss w.r.t. each weight using the **chain rule** (calculus).
- Update weights using an optimizer like **Gradient Descent**.

🛠️ Think of it like a **feedback loop** where you see the mistake and adjust the system to perform better next time.

🤝 Relationship:

- **Forward Propagation**: Calculates predictions.
- **Backward Propagation**: Improves model by reducing error.

One word prediction

WORD 2 VEC CBOW INTUITION

softmax
window size

🧠 What CBOW Does

> **Given the context (surrounding words), predict the center (target) word.**

word2vec can be pretrained or it can be built from scratch

Example; Let's assume window size = 2; this means take 2 words before & 2 words after
left Right

`window_size = 2`	2 words left + 2 words right = 4 context words

STEPS IN CBOW

🪜 Step-by-Step (with Intuition)

Let's say you have this sentence:

👉 **"The cat sits on the mat"**

You choose a **window size = 2**, and want to predict the **center word = "sits"**.

✅ 1. Context Selection

Context words (2 before, 2 after):

- `["The", "cat", "on", "the"]`

Target word:

- `"sits"`

✅ 2. Convert Context Words → One-Hot Vectors

Each word in your vocabulary becomes a **one-hot vector** (e.g., a vector with one `1` and the rest `0`s).

So, for example:

- "cat" = `[0, 1, 0, ..., 0]`
- You do this for **all 4 context words**

Now you have 4 one-hot vectors.

✅ 3. Projection Layer (Hidden Layer)

Each one-hot vector is multiplied by a **shared weight matrix** to get a dense word embedding.

If embeddings are of size 100, each context word becomes a 100-d vector.

Now:

- You have **4 embeddings** of size 100
- These are **averaged** to get **one vector** (size 100)

📌 **This is where averaging happens — not on output, but on input embeddings.**

✅ 4. Prediction Layer (Output Layer)

This average vector is passed through another weight matrix to get **a score for every word in the vocabulary** (say, 10,000 words).

Then a **softmax** is applied to turn scores into probabilities.

The word with the highest probability should be:

> **"sits"** (your target word)

✅ 5. Training (Backward Propagation)

If the predicted word is wrong:

- The model **calculates error**
- Uses **backpropagation** to update the weights (embedding matrix + output layer)

Embedding matrix
= Vocab size x embed size

✅ Yes — you choose the embedding size manually.

It's a **hyperparameter** that you set before training the model.

🔁 Simple Summary:

Step	Action
1 Window Size	Choose context words around the target
2	Convert context words to one-hot vectors
3	Map to embeddings using a shared weight matrix
4	**Average** embeddings into one vector
5	Feed to output layer → predict target word
6	Update weights via backpropagation

1. Window size (context words) (Calculate vocab)
2. OHE (vectorization of words) w.r.t vocab
3. Calculate Embed matrix (Dense vectorization) (lood V) → Vocab x
4. Average – Softmax (probability)
5. Training - Back propogate

Multi word prediction

Skip Gram Intuition

Softmax

🧠 What Skip-gram Does

> **Given the center (target) word, predict the surrounding context words.**
> The reverse of CBOW!

🚶 Step-by-Step (with Intuition)

Let's say you have this sentence:

👉 **"The cat sits on the mat"**

You choose a **window size = 2**, and pick the **target word = "sits"**

✅ 1. Context Generation

With `window size = 2`, your model tries to **predict the 2 words before and 2 after** the target word.

So:

```python
Target Word:    "sits"
Context Words:  ["The", "cat", "on", "the"]
```

Unlike CBOW, where we used multiple inputs to predict **one output**,
in Skip-gram we use **one input** (target word) to predict **multiple outputs** (each context word).

🔁 Simple Summary:

Step	Action
1	Choose a target word
2	Identify context words within window
3	Convert target word to one-hot vector
4	Map to embedding using weight matrix
5	Use embedding to predict each context word (one at a time)
6	Backpropagate loss and update weights

✅ 2. Convert Target Word → One-Hot Vector

You convert `"sits"` to a **one-hot vector** based on the vocabulary.
E.g., if vocab size = 10,000, `"sits"` = `[0, 0, 1, 0, ..., 0]`

✅ 3. Projection Layer (Hidden Layer)

The one-hot vector is multiplied by a **weight matrix** (embedding matrix)
→ Produces an **embedding vector** (say, size 100)

This embedding now represents the **meaning** of "sits"

→ Vocab × embedding size
Hyperparameter

✅ 4. Output Layer — Predict Context Words

Now, we use this embedding vector to **predict each context word** (one by one):

- Predict: `"The"`
- Predict: `"cat"`
- Predict: `"on"`
- Predict: `"the"`

Each prediction is a softmax over the whole vocabulary (10,000 words).
So you get **4 softmax outputs** from one input.

STEPS

1. window size (ie., 4 words to be predicted)
2. Target word to OHE
3. Embedding matrix

✅ 5. Training (Backward Propagation)

For each context word, the model:

- Calculates prediction error
- Uses backpropagation to adjust:
  - Embedding of the **target word**
  - Output layer weights for each context word

Google Word2Vec
- 3 billion words
- Feature representation of 300D vector

Q1 When to apply CBOW vs. SkipGram

CBOW = Small Data Sets

SKIPGRAM = Large Data Sets

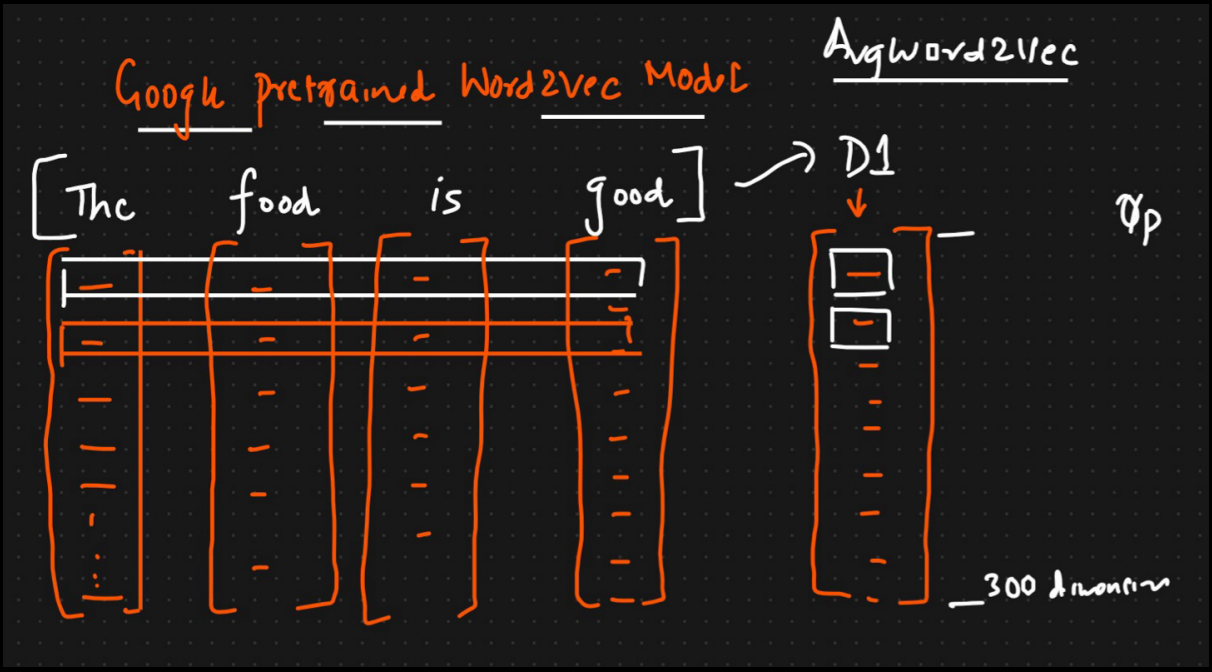
Benifits of Word2Vec :

- No sparsity ⟶ Dense Matrix
- No OOV (out of vocab) All words covered 3 B
- fast, scalable & pretrained.
- Semantic Aware (most Import)

Avg Word2Vec

> **Given a sentence**, it calculates the average of Word2Vec vectors of all the words in that sentence.

This gives you **one single vector** representing the **whole sentence or document**.



✅ 3. Use It in ML Models

This single vector can now be used as input to:

- A classifier (Logistic Regression, SVM, etc.)
- A clustering algorithm
- Similarity search, etc.

Q) what is Gensim (Toolbox) & GloVe (Stanford)

🧰 Gensim

> A popular **Python library** for working with **word embeddings and topic modeling**.

✅ Used to:

- Load **Word2Vec**, **GloVe**, and **FastText** models
- Train your own Word2Vec or Doc2Vec
- Find similar words, sentence vectors, etc.

📌 Think of Gensim as the **toolbox** to use or train embedding models.

📦 GloVe (Global Vectors for Word Representation)

> A word embedding technique developed by **Stanford**.

✅ Key idea:

- Learns word meanings by **counting how often words co-occur** in large text corpora
- Combines **global statistics** (like matrix factorization) with **local context** (like Word2Vec)

📌 Unlike Word2Vec (predictive), GloVe is more **count-based**.

BEST PRACTICES

To avoid Data leakage

STEP I; Data preprocessing

- Remove signs (using RE) re.sub('[a-zA-Z]',' ')
- Lowercase
- Tokenisation word_tokenize
- Lemmatization / Stemming (porter stemmer wordnet)
- Append corpus

STEP II; Train Test Split - To avoid Data leakage

y = get_dummies(labels)

from sklearn.feature_extraction.text import CountVectorizer, tfidfvectorizer

Hyperparameters, max_features, ngram_range

X_train = countvectorizer.fit_transform(X_train)

X_test = cv.transform(X_test).toarray

STEP III; Predictions from ML

model = sklearn.naive_bayes import MultinomialNB

we can also use Random forest classifier

Naive Bayes = Sparse Vectors

Random Forest = Dense Vectors.