NLP NOTES

NLP Day 1 :

Natural Language Processing (NLP) is a subfield of linguistics, artificial intelligence (AI), deep learning (DL), and computer science.
Natural language is the ordinary language that has evolved naturally in humans through use and repetition, without conscious planning or premeditation. The need for NLP arose from the desire to replace communication with machines, which were previously communicating in machine (binary) language.

APPLICATIONS:
1. **Language and Text Processing**

- Chatbots and Virtual Assistants (like Siri): These tools use NLP to understand spoken or typed commands. They parse your language, identify your intent ("Find a restaurant nearby"), and generate a human-like response.
- Text Generation: This includes features like autocompletion and autocorrection. When you're typing, the system uses an NLP model to predict the next word or phrase you're likely to use, helping you write faster and with fewer errors.
- Grammar Correction (like Grammarly): These services use NLP to analyze the syntax and grammar of your text. They can identify and suggest corrections for spelling, punctuation, and grammatical mistakes.

2. Information Retrieval and Organization

- Contextual Advertisements: These systems use NLP to understand the content of a web page or a user's search query. By identifying keywords and topics, they can serve ads that are highly relevant to what the user is interested in.
- Collecting Relevant Information from Files: This is often called Named Entity Recognition (NER). NLP models are used to scan documents and automatically extract specific pieces of information, such as names of people, places, dates, or organizations.
- Knowledge Graphs: These are databases that store information in a network of entities and their relationships. NLP is used to read large amounts of text and automatically extract these relationships to build and expand the knowledge graph.

3. Classification and Sentiment

- Spam Classifiers: These systems analyze the text content of emails to determine if they are spam or legitimate. They look for patterns, keywords, and phrases commonly associated with junk mail.
- Text/Document Classification: This is a broad category where NLP is used to categorize documents into predefined groups. For example, sorting news articles into categories like "Sports," "Politics," or "Technology."
- Sentiment Analysis: NLP models are used to determine the emotional tone or opinion expressed in a piece of text. For example, analyzing customer reviews on Amazon to see if the sentiment is positive, negative, or neutral.

4. Advanced Functions

- Tagging Parts of Speech: This is a fundamental NLP task where a system identifies the part of speech for each word in a sentence (e.g., noun, verb, adjective). This is a crucial step for more complex analysis like parsing.
- Language Translation: Services like Google Translate use sophisticated NLP models to translate text from one human language to another. The model processes the input text, understands its meaning, and then generates an equivalent output in the target language.
- Text Summarization (like Inshorts): These systems read a long piece of text and automatically generate a shorter, coherent summary that captures the main ideas.
- Speech-to-Text Conversion: This is a technology that takes spoken audio and transcribes it into written text. This is a crucial component of virtual assistants, call center analytics, and dictation software.

NLP Approaches:
Heuristic Approach: This approach uses simple rules and patterns.

Regular Expressions: Finding and matching specific patterns in text (e.g., finding and removing HTML tags from content).

WordNet: A type of lexical database where words are linked to each other in terms of their meaning, creating a graph or tree structure.

Open Mind Common Sense: Storing basic factual information in a database to answer common questions.

Machine Learning (ML) Approach:

Advantage over Heuristics: The output is generated based on input data and its connections, not by general rules.

How it works: Textual data is converted into numbers (a process known as vectorization), which is then used as input for an ML model.

Used Models: Naive Bayes, Logistic Regression, Support Vector Machines (SVM), Latent Dirichlet Allocation (LDA), Hidden Markov Models.

Deep Learning (DL) Approach:

Advantages:

In ML, we don't always care about sequential information. In DL, we maintain this sequential information.

In DL, we don't need to generate features manually, as is often the case with ML.

Used Architectures: Recurrent Neural Networks (RNN), Long Short-Term Memory (LSTM), Gated Recurrent Units (GRU), Convolutional Neural Networks (CNN), and Transformers.

Why NLP is Difficult:
Ambiguity: Sentences can have more than one meaning.

Example 1: "I saw a boy on the beach with my goggles."

Interpretation 1: I saw the boy on the beach while I was using my goggles.

Interpretation 2: I saw the boy on the beach where he was wearing my goggles.

Example 2: "I never ate a cake like that before."

This could mean the cake was either very good or very bad.

Contextual Words: The same word can have different meanings based on the context of the sentence.

Example: "I ran to the milk store because we ran out of milk." Here, "ran" has two different meanings.

Colloquialisms and Slangs:

"Piece of cake" (slang for an easy task).

"Pulling your leg" (slang for teasing or joking).

NLP DAY 2 : NLP PIPELINE

The NLP pipeline is the series of steps followed to build an end-to-end NLP software.

Step 1: Data Acquisition :

Collecting data for the project.

1 Data with us (internal sources):

Data is available over the table (in simple files).

Data is in the company database (ask a data engineer).

If you have a small amount of data, you can use techniques like data augmentation, bigram flipping (swapping two consecutive words), back-translation (translating a query to another language and back), or adding noise to the query.

2 Data not with us (external sources):

Use public datasets (e.g., from Kaggle).

Web scraping (collecting data from websites using a tool like Beautiful Soup).

Use an API (e.g., from rapidapi.com) to collect data.

Collect data from PDFs or images (using OCR).

Collect data from audio/video (using speech-to-text).

3 Data not present (creation):

Manually collect data from users through forms and classify them.

Generate synthetic data to supplement a small dataset.

Step 2: Text Preparation :

Preparing the text for future stages.

Basic Text Cleaning:

Removing HTML tags.

Removing emojis.

Converting shorthand to full form.

Correcting typos.

Basic Preprocessing:

Sentence Tokenization and Word Tokenization (splitting text into sentences and words).

Stop Word Removal (removing common words like "the," "a," "is").

Lemmatization and Stemming (reducing words to their base or root form).

Lower casing and language detection.

Advanced Preprocessing:

Parsing (analyzing the grammatical structure of a sentence).

Parts of Speech (POS) Tagging (labeling each word in a sentence with its part of speech, like noun, verb, adjective).

Coreference Resolution (finding all expressions that refer to the same entity in a text, e.g., identifying that "he," "him," and "the man" all refer to the same person).

Explanation of Unexplained Terms
Heuristic Approach: A heuristic is a practical, rule-of-thumb method for solving a problem, which may not always be perfect but is fast and simple.

WordNet: A large lexical database of English where nouns, verbs, adjectives, and adverbs are grouped into sets of cognitive synonyms.

Open Mind Common Sense: A project that collects common-sense knowledge from people all over the world to build a large-scale database of facts.

Vectorization: The process of converting text into a numerical vector that a machine learning model can understand.

LDA (Latent Dirichlet Allocation): A topic modeling algorithm used to discover the abstract "topics" that occur in a collection of documents.

RNN, LSTM, GRU: Types of neural networks designed to process sequential data, where the output depends on previous inputs in the sequence. LSTMs and GRUs are more advanced versions of RNNs that solve the vanishing gradient problem.

CNN, Transformers:

CNN (Convolutional Neural Networks): Originally used for images, CNNs can also be used for NLP tasks by applying filters to sequences of words.

Transformers: A modern neural network architecture that uses self-attention mechanisms to weigh the importance of different words in a sequence. They are the foundation of state-of-the-art LLMs.

Lemmatization vs. Stemming:

Stemming: A crude heuristic process that chops off the end of a word to get its root (e.g., "running" → "run").

Lemmatization: A more sophisticated linguistic process that uses a vocabulary and morphological analysis to get the base form of a word, known as a lemma (e.g., "better" → "good").

STEP 3 : FEATURE ENGINEERING :

converting our text to numbers

1. in ml we need to preprocess out input data then we need to create the feature ourself based on our knowledge over that domain
2. in dl we need to preprocess the input data but we don't need to create feature our dl model dose it by itself

advantages / disadvantages:

in ml we are doing feature creation by ourself but not in dl so in dl we don't require any domain knowledge to create any feature
in dl the main disadv over ml is we cannot predict what is the main feature that was reason for models accuracy but in ml we can have the
ability to justify which feature was the reason for model accuracy through metrics like correlation , covariance

STEP 4 : MODELLING :

in this step we have exact /perfect data we train the model on data set and then we do evaluation

step a : modelling:

1 heuristic method
2 ml approaches
3 dl approaches
4 cloud api

based on amount of data and complexity we decide which modelling approach is selected

if we less data we can use heuristic approach
if we have more data use ml if we have still more we use DL
if solution is already present then use cloud solution

step b : evaluation:

1 internal evaluation
2 external evaluation

STEEP 5 deployment:

3 stages :
a deployment
b monitoring
c update

NLP DAY 3 : TEXT PREPROCESSING

Preprocessing converts raw text (noisy, inconsistent) into a cleaner, more consistent form that models can learn from. The exact steps depend on task (IR, classification, NER, translation, language modeling) — never blindly apply every step. Always decide based on downstream model requirements.

Recommended generic pipeline (order + rationale)

1. Lowercasing (and casefolding)

- lower casing is done so that 2 same word one start with upper and other with lower case do not represent the 2 different words during tokenization of the input string

```
text = text.casefold()  # robust Unicode-aware lowercasing
# or
text = text.lower()  # simple ASCII lowercasing
```

2. Removing HTML tags

- Why: web-scraped text contains <script>, <a>, etc., which confuse models.
- Safe ways:
  - BeautifulSoup: reliable — BeautifulSoup(html, "html.parser").get_text()

3. Removing / replacing URLs

- Why: URLs are noisy and many of the tokens are useless; but sometimes domain or path matters.
- Strategy:
  - Replace with placeholder: __URL__ or <URL>.
  - If domain info matters (e.g., news), extract domain separately.
- Regex:

```
text = re.sub(r'https?://\S+|www\.\S+', '<URL>', text)
```

4. Removing punctuation

- Why: punctuation increases tokens; sometimes not informative.
- Caveats:
  - Don’t blindly strip: apostrophes (negation), hyphens (word compounding), emoticons, abbreviations (U.S.A.), decimal points in numbers.
  - For sentiment tasks, punctuation like !!! or ?!! matters — consider keeping or encoding.
- Approach: either remove by regex or remove except preserved characters:

```
import string
table = str.maketrans('', '', string.punctuation)
text = text.translate(table)  # removes standard ASCII punctuation
```

5. Chat word treatment (slang / abbreviations)

- What: expand chat abbreviations and shorthand to full forms.
- Example map: {'gn': 'good night', 'asap': 'as soon as possible', 'brb': 'be right back'}
- Approach:
  - Maintain a dictionary for expansions.
  - Apply token-level replacement before tokenization.
  - Watch out for ambiguous expansions; prefer context or manual curation for domain-specific slang (gaming, medical).
- Tip: also expand contractions (don't → do not) — libraries exist (contractions package).

6. Spelling correction

- Why: fix typos that harm vocabulary and downstream models.
- Libraries & approaches:
  - Rule / edit-distance: pyspellchecker (Peter Norvig style), symspellpy (fast, dictionary-based).
  - Probabilistic: TextBlob's .correct().
  - Context-aware: transformer-based spell checkers (BERT seq2seq) — better but heavier.
- Caveats:
  - Corrections can introduce errors (especially for proper nouns, domain terms, code words).
  - For short text / social media, spell correction may remove intended stylizations.
- Recommendation: use only when typos are common and vocabulary cleaning matters; for classification often unnecessary if using robust embeddings.

7. Stop-word removal

- What: remove frequent functional words (the, is, and) to reduce noise.
- Use:
  - NLTK, spaCy provide default lists — but customize heavily.
  - Keep negations (not, no) for sentiment — remove carefully.
  - For topic modeling, stopwords help; for language models or NER they may harm.
- Implement:

```
from nltk.corpus import stopwords
stopset = set(stopwords.words('english')) - {'not','no'}  # keep negations
tokens = [t for t in tokens if t not in stopset]
```

8. Handling emojis

Two options:

1. Remove emojis (if irrelevant).
2. Replace emojis with text labels (preserve sentiment/meaning).

- Libraries: emoji (Python) to demojize: emoji.demojize("I ❤️ NLP") → "I :red_heart: NLP".
- Approach:
  - Use emoji.demojize() to convert to tokens like :smile: and then map to words ':smile:' -> 'smile' or phrases "smiling_face".
  - For short messages, emojis carry strong sentiment — prefer to convert rather than drop.
- Emoticons: convert :-) and :(to happy / sad manually or via regex mapping.

9. Tokenization

- Types:
  - Sentence tokenization: split text into sentences (nltk.sent_tokenize, spaCy).
  - Word tokenization: whitespace, regex, or language-specific tokenizers.
  - Subword tokenization: BPE, WordPiece, SentencePiece (used by transformer models) — reduces OOVs.
- Challenges:
  - Prefixes: $10, #hashtag, @user.
  - Suffixes: 10km, 50%.
  - Ambiguity: same word with different meanings (word sense) — tokenization alone can't solve.
  - Languages without spaces: Chinese/Japanese require segmentation (jieba, MeCab).
- Libraries:
  - spaCy: high-quality tokenization + POS + NER; customizable rules.
  - NLTK, tokenizers (Hugging Face) for subword tokenization.
- Example (spaCy):

```
import spacy
nlp = spacy.load("en_core_web_sm")
doc = nlp("This is a sentence.")
tokens = [tok.text for tok in doc]
```

10. Stemming

- What: crude heuristic chopping of word endings (e.g., running → run, happiness → happi).
- Algorithms: Porter, Snowball (Porter2), Lancaster.
- Pros: fast, reduces vocab.
- Cons: may produce non-word stems; over-stemming can merge unrelated words.
- When: IR or when loss of morphological correctness is acceptable.
- Python (NLTK):

```
from nltk.stem import PorterStemmer
stemmer = PorterStemmer()
stemmer.stem('running')  # 'run'
```

11. Lemmatization

- What: map word to base/dictionary form using vocabulary & POS (e.g., better → good, running → run).
- Requires: POS tags for best results (WordNet lemmatizer).
- Pros: returns real words; preserves meaning better than stemming.
- Cons: slower, requires POS tagger or morphological analyzer.
- Libraries:
  - NLTK WordNetLemmatizer (needs POS tag)
  - spaCy: fast and returns lemmas (no separate POS step needed — integrated).
- Example (NLTK):

```
from nltk.stem import WordNetLemmatizer
lemmatizer = WordNetLemmatizer()
lemmatizer.lemmatize('running', pos='v')  # 'run'
```

A**dditional important topics (often missing)**

Contraction expansion

- Expand I'm → I am, they're → they are. Use contractions library or custom rules. Expand before removing apostrophes.

Unicode normalization & accent removal

- Normalize forms (NFC / NFKC) and optionally remove accents (unicodedata.normalize('NFKD', s) + remove combining characters) or use unidecode for transliteration.

HTML entities & BOM

- Use html.unescape() and ftfy.fix_text() to repair mojibake.

Number / date / units normalization

- Replace numbers with <NUM> or normalize representation (1,000 → 1000, 10k → 10000). Dates → <DATE>. Units (kg, km) often normalized separately.

Hashtags / Mentions

- For social data:
  - Replace @user with <USER>.
  - Split hashtags with camelCase: #NewDelhiTrip → new delhi trip (use wordsegment or heuristic).

Language detection & multilingual pipelines

- Detect language (langdetect, fastText) before preprocessing; some steps differ per language (stemming/lemmatization not language-agnostic).

Subword tokenization (BPE / WordPiece / SentencePiece)

- Modern transformer models require their tokenizer (Hugging Face tokenizers) — do not apply classic tokenization before these, use the model tokenizer.

Handling OOV & rare words

- Options: map to <UNK>, use subword tokenization, or replace rare tokens with categories.

Preserve Named Entities

- If NER matters (e.g., in QA), mark or protect entities from lowercasing or stemming; better to extract entities first.

Negation handling

- Transform not good → not_good or keep negations to retain sentiment polarity. Removing stopwords blindly will remove not.

Data deduplication & cleaning

- Remove near-duplicates, empty lines, or repeated characters (e.g., sooooo good — convert repeated chars to soo good or mark emphasis).

Fast, scalable preprocessing

- Use spaCy's nlp.pipe() for batch processing, or vectorized pandas operations. For massive corpora, use streaming and chunking.

Pitfalls & gotchas

- Over-cleaning: can remove signals (negation, emphasis, capitalization).
- Spell-correction hazards: can change domain terms & names.
- Order matters: expanding contractions after removing apostrophes will fail.
- Cross-language content: English-only tokenizers fail on code-switched text.
- Emoji/Emoticon loss: dropping them loses strong sentiment cues.
- Model mismatch: preprocessing must match the assumptions of the downstream model (uncased vs cased BERT).

Final short cheat-sheet (what to run and when)

- html.unescape() + BeautifulSoup → HTML removal
- unicodedata.normalize('NFKC') → Unicode normalization
- re.sub() for URLs/emails → replace with placeholders
- contractions.fix() → expand don't → do not
- emoji.demojize() → convert emoji to readable tokens
- .casefold() → robust lowercasing (if applicable)
- spaCy pipeline → tokenization + POS + lemma
- Optional: symspellpy / TextBlob → spelling fixes
- Stopwords removal → after tokenization/lemmatization (keep negations!)

lemmetization v/s stemming :

🔹 Stemming
- Definition: A rule-based, heuristic process that chops off prefixes/suffixes to reduce a word to its root form.
- Output: The result may not be a valid dictionary word.
- Algorithm examples: Porter Stemmer, Snowball Stemmer, Lancaster Stemmer.
- Speed: Fast, since it only applies string rules (no dictionary lookup).
- Accuracy: Lower — can over-stem (merge unrelated words) or under-stem.

👉 Examples:
- running → run ✅
- happiness → happi ❌ (not a valid word)
- studies → studi ❌

🔹 Lemmatization
- Definition: A linguistic, vocabulary-based process that reduces a word to its base/dictionary form (lemma), considering morphology and part-of-speech (POS).
- Output: Always a valid dictionary word.
- Libraries: WordNet Lemmatizer (NLTK), spaCy.
- Speed: Slower, since it needs POS tagging and dictionary lookups.
- Accuracy: Higher — preserves true base word.

👉 Examples:
- running (verb) → run ✅
- happiness → happy ✅
- studies (noun) → study ✅

POS TAGGING IN LAMMETIZATION :
POS info means Part-Of-Speech information — i.e., knowing whether a word is used as a noun, verb, adjective, adverb, etc. in a sentence.

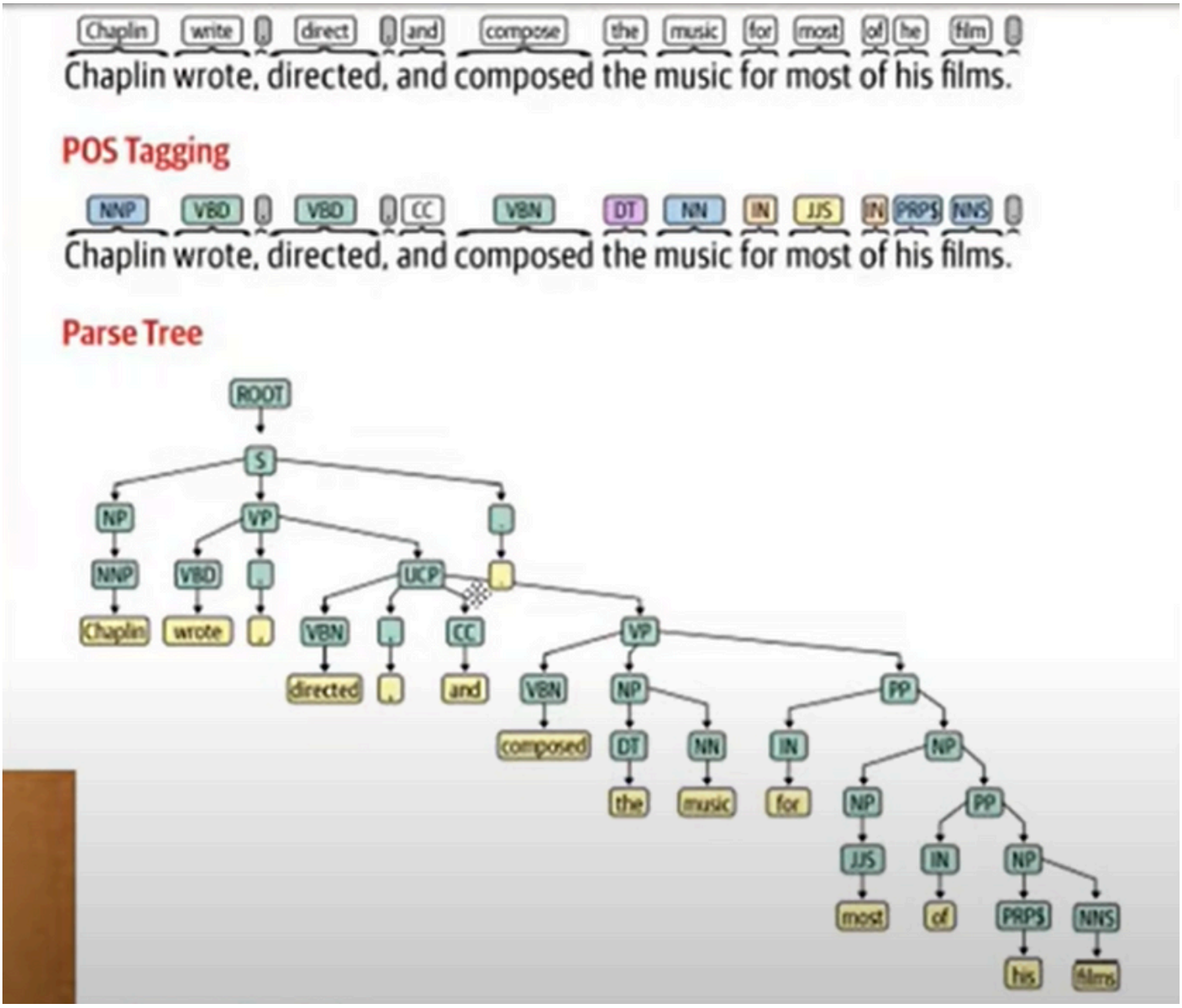
🔹 Why is POS info important?
- Many words in English change meaning depending on their role.
- Lemmatization needs POS tags to return the correct base (lemma).
- Without POS info, a word may not be reduced correctly.

🔹 Example

Take the word "flies":

- As a noun: "The flies are buzzing."
- Lemma = "fly" (insect)
- As a verb: "She flies to Delhi every week."
- Lemma = "fly" (action of flying)

So same surface word "flies" → different lemmas depending on POS.



DAY 4 NLP : TEXT PREPRESENTATION , WORD2WECK:

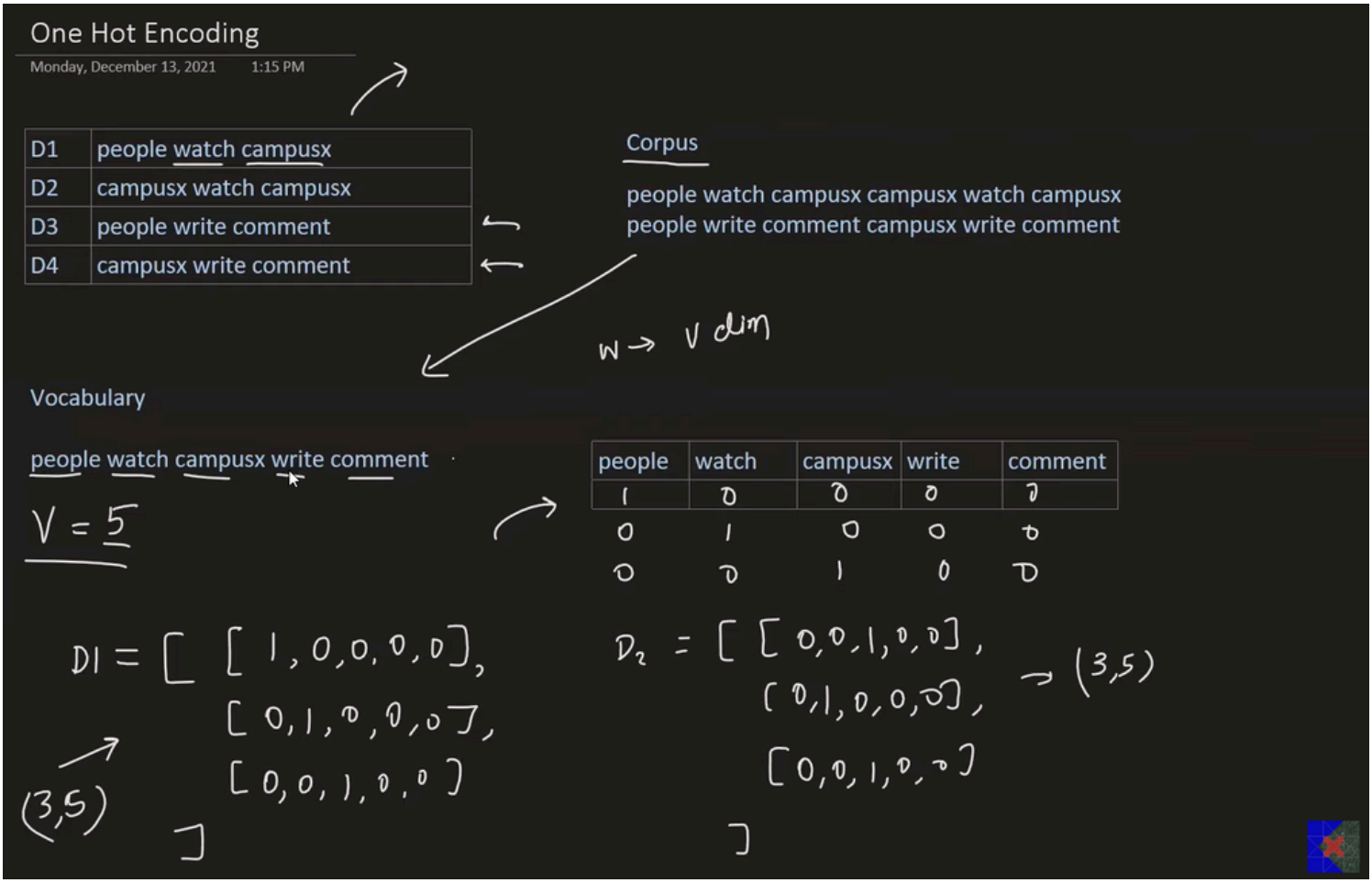
definition of word, document , vocabulary and corpus:

1. "NLP is fun"
2. "I love NLP"
3. "Python is powerful"

word is basic unit of language
document is set of words like sentence ,query
vocabulary : set of all unique words
corpus is set of all document

- Words = ["NLP", "is", "fun", "I", "love", "Python", "powerful"]
- Documents = Sentence 1, Sentence 2, Sentence 3
- Vocabulary = {"NLP", "is", "fun", "I", "love", "Python", "powerful"} (7 unique words)
- Corpus = ["NLP is fun", "I love NLP", "Python is powerful"]

ONE HOT ENCODING :
converting my documents in the form of numbers/ vector form



ADV: its easy to understand and can be coded easily

DISADV: if my corpus has 1000000 words and in my vocabulary has 30000 unique words then after **OHE** every word in my document is represented in form of vector of dimension of 30000

1 due to this sparse array is created for every word and overfitting occurs

2 in ml all my input must have same size to train the the model if i use document of different length then my model wont be trained

3 **OOV problem :** during testing if a new word comes like “HELLO” which is not present in our vocabulary then its not possible to represent “HELLO” interms of 0 and 1 as HELLO was not present in our vocabulary

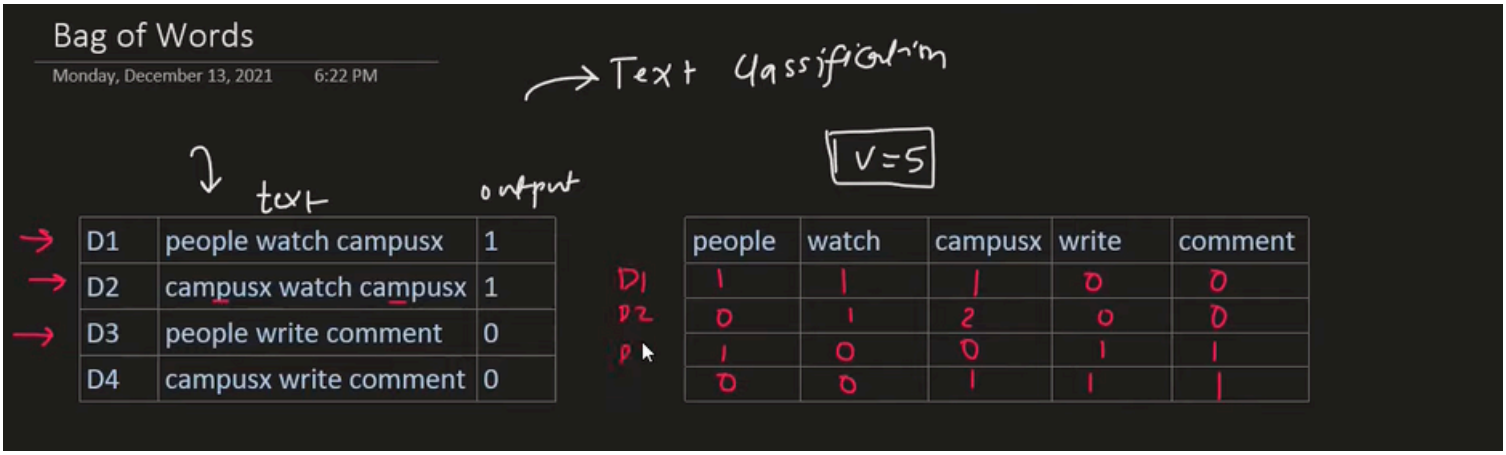
4. **Loss of Semantic Meaning**

- OHE treats all words as equally distant from each other.
- Example: Vocabulary = [run, walk, bottle]
  - run = [1,0,0]
  - walk = [0,1,0]
  - bottle = [0,0,1]
- In vector space:
  - Distance(run, walk) = Distance(run, bottle) = Distance(walk, bottle) = √2
- But in reality:
  - "run" and "walk" are semantically similar (both actions).
  - "bottle" is unrelated.
- 👉 OHE fails to capture word similarity/relationships

due to these adv we never use OHE for representing of document in numbers

BAG OF WORDS:

In Bag of Words, a document is represented as a vector of word counts (or frequencies).



in bag of words the order of the words dose not matters

when we represent all our document in vector form its placed in a dimensional space and the similarity between the sentence is measured based on the angle created between those 2 vectors in that n dimensional space

this can be implemented in sklearn

from sklearn.feature_extraction.text import countvectorizer

Advantages of Bag of Words (BoW) over One-Hot Encoding (OHE) :

Reduced Dimensionality per Document

- OHE: Every word is represented as a vector of size = |Vocabulary|.
  - Example: Vocabulary = 30,000 words → each word = 30,000-dimensional vector.
  - For a document of 1,000 words → size = 30,000 × 1,000 = 30M dimensions 😵.
- BoW: Each document is represented as a single vector of size = |Vocabulary|.
  - Same document → just 30,000 dimensions, not multiplied by document length.
  - ✅ Much more compact than OHE.

Handles Documents of Any Length

- OHE: Representation size depends on the number of words (document length).
- BoW: Representation size = |Vocabulary| (fixed for all documents).
- ✅ Works seamlessly for short or long documents.

Simpler & Faster to Train

- With BoW, every document is a fixed-length vector.
- No need for padding/truncating like in OHE.
- ✅ Easier and faster for ML models to process.

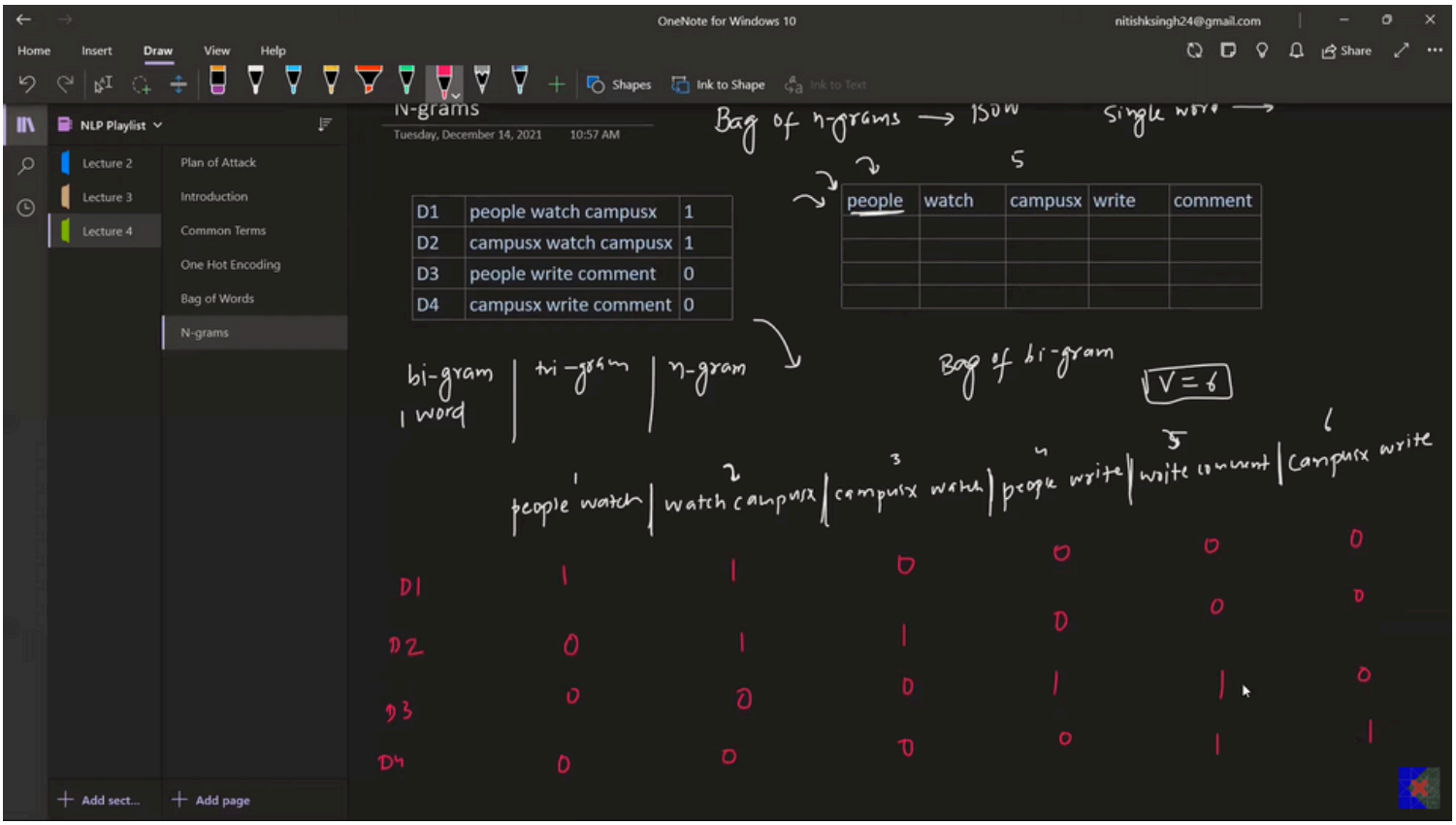
Disadvantages OF BOW :

1. OOV Problem (Out-of-Vocabulary)
   - If a new word appears during testing (e.g., "HELLO" not in vocabulary), it cannot be represented.
2. Sparsity
   - Even though it's smaller than OHE, most documents use only a small subset of the vocabulary.
   - Example: In a vocabulary of 30,000 words, a short document may only use 100 words → most entries are 0.
   - Leads to sparse vectors, inefficient storage, and harder ML training.
3. No Semantics
   - "run" and "walk" are represented as unrelated, even though they are similar.
   - Word order is ignored → "dog bites man" = "man bites dog".

3 BAG OF N-GRAMS , BI-GRAMS , TRI-GRAMS :

in OHE and BOW we dont care about the order of words present in our document due to this we loss the real meaning of our doccument
here instead of taking individual word we take group of words at a time in our table to create the base of our table

Instead of taking single words (unigrams), we take groups of words.
n-gram = sequence of n consecutive words.
Examples:
Unigrams (n=1): ["I", "love", "dogs"]
Bigrams (n=2): ["I love", "love dogs"]
Trigrams (n=3): ["I love dogs"]



✅ Advantages of N-grams :

1. Captures Word Order / Context
   - Preserves local context and meaning.
   - "New York" is different from "New" + "York".
2. Improves Performance in NLP tasks
   - Useful for text classification, sentiment analysis, speech recognition, etc.
3. Better Semantic Understanding
   - "not good" vs "good" → unigram would fail, but bigram captures negativity.

❌ Disadvantages of N-grams :

1. Vocabulary Explosion
   - For vocabulary size = V:
     - Unigrams = V
     - Bigrams ≈ V^2
     - Trigrams ≈ V^3 😵
   - This increases memory usage & sparsity.

- Data Sparsity Problem
- Many n-grams may never appear in training but could appear in testing.
- Short-Range Context Only
- Bi-grams or tri-grams capture only local word dependencies, not long-range context (solved later by embeddings & transformers).

4 **TF - IDF :**

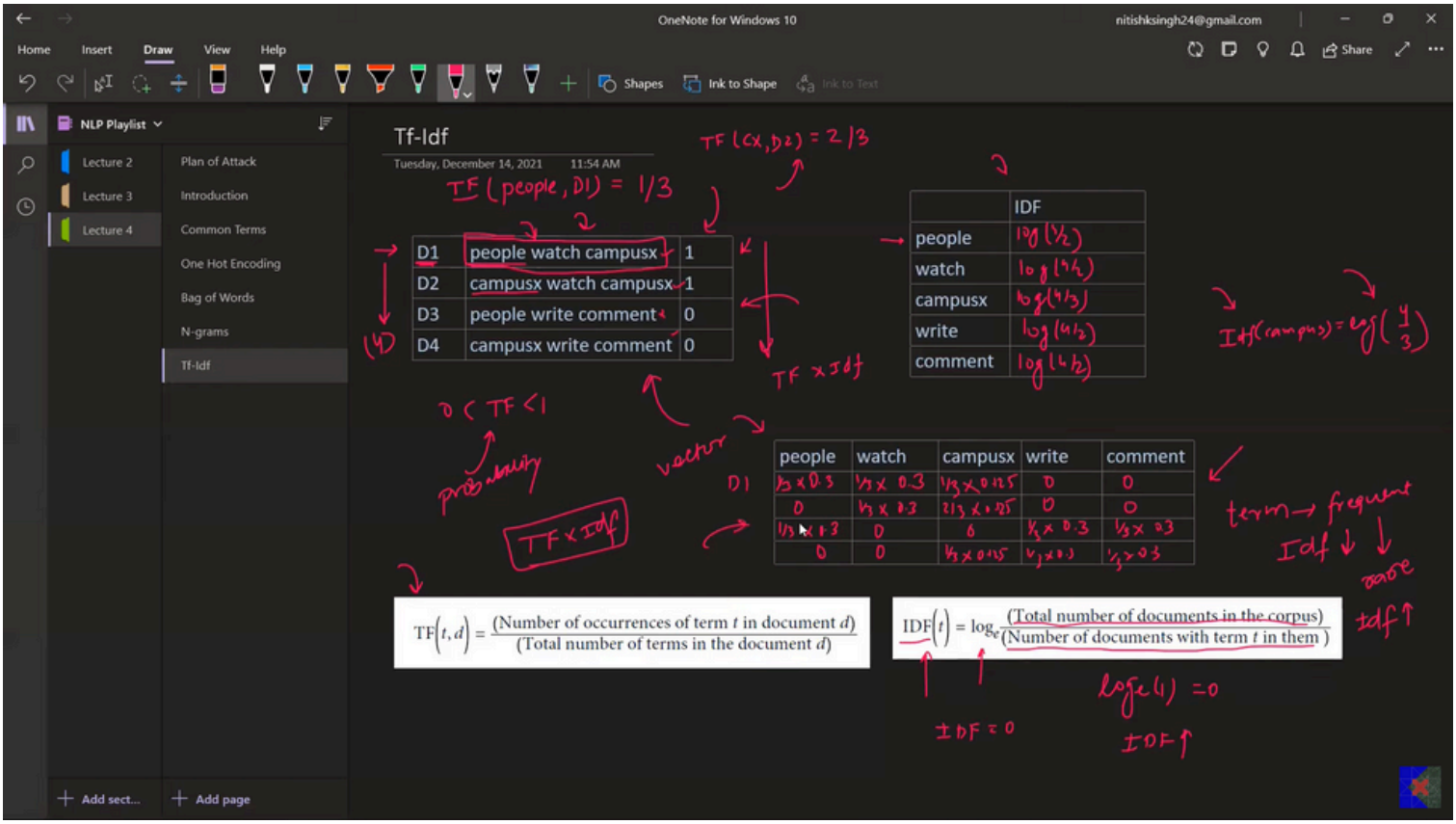
the scoring for a particular word is not depends on frequency it depends of TF and IDF basically we give more score to that word which is present more in a particular doc but rarely present in corpus

TF(t,d) = (no of occurrence of term t in doc d) /
(total no of term in doc d)

IDF(t) =log((total no of doc in corpus)/
(no of doc with term t)) + **1**

1 is added if term t present in all doc then log (n/n) becomes 0 due to this we miss TF contribution so we add 1

TF-IDF(t,d)=TF(t,d)×IDF(t)



need of log in IDF term ??

- Without log:
  - rare word present only in 1 out of 10000 doc then IDF for it will be (10000/1) = 10000
  - common word present in 5000 doc so IDF for it will be 10000/5000 =2
  - there is a large difference b/w to IDF score due to this its not comparable
  - with log :
  - rare = log(10000)=4
  - common log(2) = 0.3
  - now there is difference but its in a comparable range

🟦 TF–IDF: Advantages & Disadvantages

✅ Advantages

1. Highlights Important Words
   - Gives more weight to words that are frequent in a document but rare in the corpus.
   - Helps focus on meaningful keywords instead of common words like "the" or "is".
2. Better than BoW
   - Unlike BoW, TF-IDF considers both term frequency and word rarity, making it more informative.
3. Fixed-Length Representation
   - Vector size = vocabulary size → easy to feed into ML models.
4. Widely Used in Information Retrieval
   - Search engines, document ranking, and text classification often use TF-IDF.

❌ Disadvantages

1. Ignores Word Order
   - "not good" and "good not" are treated the same → context lost.
2. Sparse Vectors
   - For large vocabularies, most entries are zero → inefficient storage.
3. No Semantic Meaning
   - Synonyms like "car" and "automobile" are treated as unrelated words.
4. Static Weights
   - Once TF-IDF is computed from a corpus, it doesn’t adapt dynamically to new context.
5. Sensitive to Corpus Size
   - Rare words in a small corpus may get exaggerated importance

5 TF-IDF WITH N-GRAMS:

Using TF-IDF with N-grams

🔹 Why Mix TF-IDF and N-grams?

1. Capture Word Importance (TF-IDF)
   - TF-IDF highlights words or word sequences that are frequent in a document but rare in the corpus.
2. Capture Local Context (N-grams)
   - N-grams capture word order and short-range dependencies.
   - Example: "not good" as a bigram → distinguishes sentiment better than unigram "not" + "good".
3. Combined Benefit
   - TF-IDF assigns weights to n-grams based on importance.
   - Rare but meaningful n-grams get higher score → improves text classification, sentiment analysis, and information retrieval.

🔹 **Example**

Sentence: "I do not like this movie"

- Unigrams → "I", "do", "not", "like", "this", "movie"
- Bigrams → "I do", "do not", "not like", "like this", "this movie"
- TF-IDF on unigrams → "not" may get low importance (common word)
- TF-IDF on bigrams → "not like" → rare and important → higher weight → captures negative sentiment

🔹 **Advantages of TF-IDF + N-grams**

1. Captures word importance + local context
2. Improves performance in text classification & sentiment analysis
3. Preserves phrases or meaningful word combinations

🔹 **Disadvantages**

1. Vocabulary Explosion → N-grams increase size exponentially → sparse vectors
2. Computationally Expensive → especially for large corpora
3. Still Limited Context → Only captures short-range relationships (for long context → embeddings or transformers are needed)

OHE → BoW → N-grams → TF-IDF → TF-IDF+N-grams → Embeddings

6 CREATING THE CUSTOM FEATURE :

based on the problem statement we create the custom feature

Creating Custom Features in NLP

- Beyond standard text representations (OHE, BoW, TF-IDF, embeddings), you can design your own features from the text.
- These features depend on the problem statement (what you are trying to predict).
- The goal is to extract information that makes the model more accurate.

🔹 Examples of Custom Features

1. **Lexical Features (Surface-level text features)**
   - Word Count → total words in the document
   - Character Count → length of text
   - Average Word Length
   - Number of Unique Words
   - Punctuation Count → e.g., "!" in reviews can indicate strong sentiment
   - Uppercase Ratio → "THIS MOVIE IS GREAT" = shouting/strong opinion
2. **Syntactic Features (Structure-related**)
   - POS Tag Distribution → how many nouns, verbs, adjectives in the text
   - Parse Tree Depth (complexity of sentence)
3. S**emantic Features (Meaning-related)**
   - Sentiment Score → from libraries like TextBlob or VADER
   - Subjectivity Score → is the text opinionated or factual?
   - Keyword Presence → e.g., in spam detection: words like "FREE", "WINNER", "OFFER"
4. **Domain-Specific Features**
   - In spam classification → number of links, number of digits, suspicious words
   - In hotel reviews → presence of words like "clean", "dirty", "service", "location"
   - In finance → count of positive vs negative financial terms (profit, loss, growth)

🔹 **Why Create Custom Features?**

1. Boost Model Performance → Sometimes raw TF-IDF/embeddings miss simple cues (like exclamation marks in sentiment).
2. Domain Knowledge → Custom features let you encode expert insights into the model.
3.

DAY 5 : WORD EMBEDDING AND WORD2vec :

- **WORD EMBEDDING :**A word embedding is a way of representing words as dense vectors of real numbers in a continuous vector space.
- Words with similar meanings or usage end up being close together in this space.

TYPES :
1 frequency based :
BOW
TF-IDF

2 prediction based :
WORD2vec

in word2vec we can easily capture semantic meaning of words
size of vectors will be small so speed increases
its dense non zero vector due to this no sparse array → no overfitting

types in word2vec:
1 CBOW: continuous Bag of words
on higher view OBOW Is as same as BERT models as we trying to predict masked word using unmasked
here for a give query/sentence we try to predict the center word based on surrounding words present in our query

let us assume we have a vocabulary of 10000 words
since its a neural network we have 2 layers
1st hidden layer consist of 300 neurons and 2nd layer consist of 10000 words basically vocabulary size
window size refers to no of words to consider before and after the center word
“lion is king of jungle” the center word in king i need to predict the word king using other 4 words

STEPS:
window size = 2
1 do ONE HOT ENCODING to words [lion , is , of , jungle]
lion → [0, 0, 1, 0, ..., 0] (1 × 10,000)
is → [0, 1, 0, 0, ..., 0]
of → [0, 0, 0, 1, ..., 0]
jungle → [0, 0, 0, 0, ..., 1]

2 pass it to shallow neural network where there is 300 neuron in 1st hidden layer
so that word which is in (1,10000) is converted into (1,300) form

3 repeat this step to all 4 words

4 aggregation :
here add all 4 vector and take avg of it resulting a new vector formed by aggregation of all4 vector

5 pass this new vector into 2nd layer consisting of 10000 neurons with soft max as its activation function

6 this results a new vector of dimension of (1,10000) that represents the predicted middle word

Component Shape / Size :
One-hot input : 1 × 10,000
embedding matrix : 10000* 300
Embedding vector : 1 × 300
Aggregated vector : 1 × 300
Output weights W′ : 300 × 10,000
Output logits : 1 × 10,000
Soft max output : 1 × 10,000

```mathematica
Context Words: "lion", "is", "of", "jungle"
      |
      ▼
One-Hot Encoding (1 x 10,000 each)
      |
      ▼
Embedding Lookup (W: 10,000 x 300)
      |
      ▼
Embeddings e1, e2, e3, e4 (each 1 x 300)
      |
      ▼
Aggregation (Average)
      h = (e1 + e2 + e3 + e4)/4  → 1 x 300
      |
      ▼
Hidden → Output Projection (W': 300 x 10,000)
      |
      ▼
Output logits u = h x W' → 1 x 10,000
      |
      ▼
Softmax Activation
      ŷ = softmax(u) → 1 x 10,000
      |
      ▼
Cross-Entropy Loss with True Center Word ("king") → Backpropagation
      |
      ▼
Update Weights W (embedding) and W' (output)
```

2 SKIP -GRAM :

in skip gram we predict the surrounding words using the central word

“lion is king of jungle “

step 1 :
center word : king , do OHE for king ->[0,0,....1,0,0]

step 2:
multiply this vector with embedding matrix of size 10000*300 resulting in a (1,300) embedding vector of king

step 3 : this embedding vector is then multiplied with weighted matrix of dimension (300*10000) resulting in a (1,10000) dimension vector

Step 4 (refined explanation)

- After multiplying the embedding (1 × 300) with the output weight matrix (300 × 10000), you get a score vector (1 × 10000) for all vocabulary words.
- Apply softmax → probabilities for all words.
- Compare the predicted probabilities with the actual context words in your window.
  - E.g., for "king" with window = 2 → context words = [“is”, “of”]
- Compute cross-entropy loss and use backpropagation to update:
  - Embedding matrix (W) → adjusts the dense vector for "king"
  - Output weight matrix (W') → adjusts scores for predicting context words
- Over many training steps, the model learns embeddings so that "king" predicts "is" and "of" with high probability.

DAY 6 : TEXT CLASSIFICATION :

its supervised ml task where we do classification

types of classifications :
1 binary classification (dog or cat),(email spam classifier),(sentiment classifiers in ecom)
2 multiclass classification(iris classification)
3multilabled (one input can be classified into multiple classes)
ex lion can be classified as animal, carnivorous , predator

TEXT CLASSIFICATION PIPELINE :
1 data acquisition
2 text preprocessing
3 text vectorization
4 modelling(ml(naive base , DT)/dl model(RNN(LSTM) , CNN, BERT)
5 deployment

Approaches of modelling :
1heuristic approach
2 APIs
3Machine learning models
4Deeplearning models

1 **ml model to predict the disease :**

https://github.com/adithyaprabhu007/NLP_notes_codes/blob/main/disease_prediction_logistic_regression.ipynb

In this model we used a pretrained Word2Vec model.
So in this model there are no 2 hidden layers as we studied previously.
Our architecture is a weighted matrix of size 100 × 7 so that when we multiply the embedding with this matrix, it forms a 1 × 7 dimension array. A softmax function is applied over this array and prediction is done based on the max value.

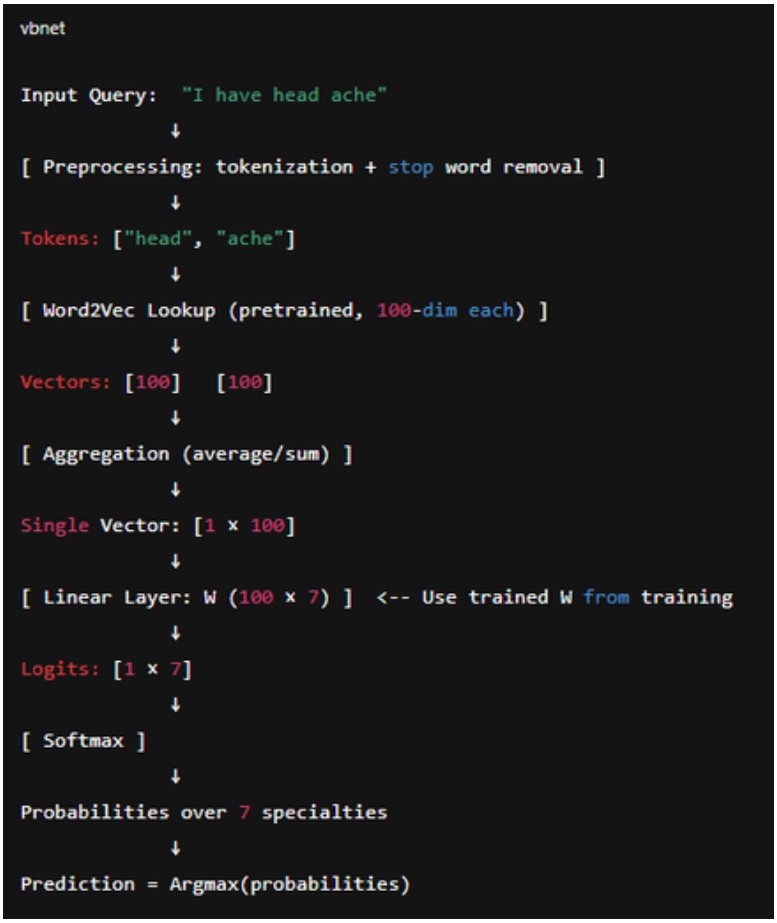
- 7 refers to 7 different specialties
- 100 refers to embedding vector dimension

The weighted matrix is updated using backpropagation and gradient descent, with the help of categorical cross entropy.

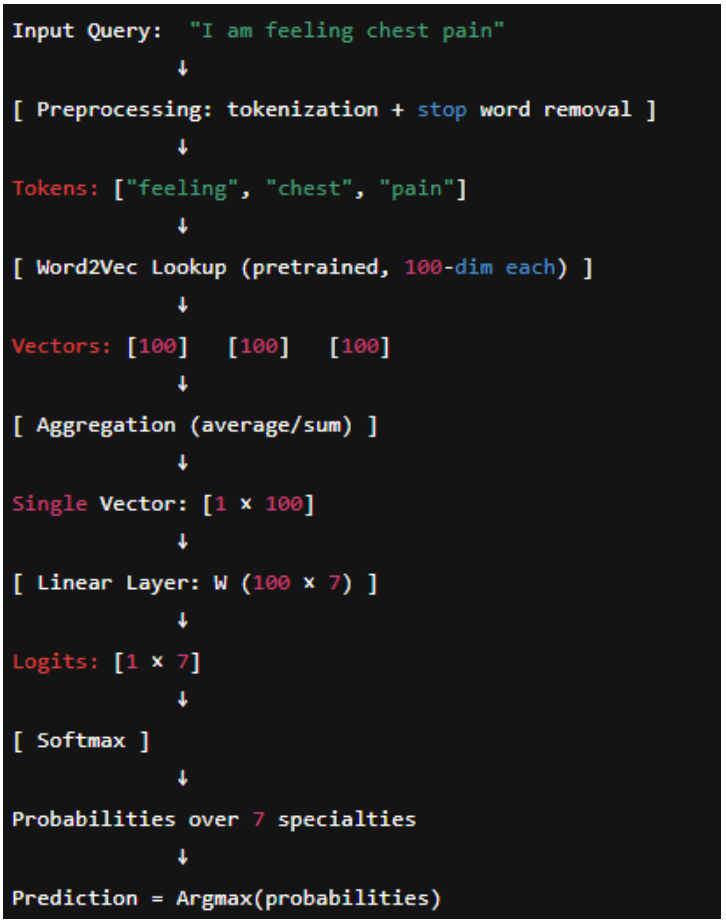
Steps :

1. If my vocabulary has 10,000 words, the Word2Vec pretrained model gives meaningful vector embeddings to all these words at the start of training whenever required.
2. During training of my weights, if my query is “I am feeling chest pain”, after tokenization and stop word removal there are only “feeling”, “chest”, “pain”. Our model gives vector embeddings for these words.
3. Aggregation of all 3 embeddings is done and multiplied with the weighted matrix. This results in a [1,7] array. After applying the softmax function, the best one is selected based on the softmax score as prediction. The prediction is then converted to OHE along with the actual value, and loss is calculated to train my weight matrix.
4. During testing, if the query is “I have head ache”, the same process of tokenization and stop word removal is done, resulting in [head, ache]. Word2Vec returns the vector embeddings for these 2 words and the aggregate vector embedding is calculated. This is multiplied with the weight matrix, and softmax is applied over the [1,7] dimension array, and the best one is selected as the result.
5. there is no problem of OOV as my word2vec gives embedding of any word at any time given that the word must present in GOOGLE NEWS vocabulary on which it was trained

TRAINING



TESTING



2 disease prediction with out using any pretrained model to get vector embedding of words:

If I don't use a pretrained model to get embeddings, I need to train my own embeddings from scratch.

- While training, I need a large number of sentences to capture meaningful word representations.

Steps:

1. Text preprocessing – tokenize, remove stop words, clean text.
2. Apply CBOW (Continuous Bag of Words) – this helps the model understand semantic relations between words and learn embeddings for all words in the vocabulary.

When training my own model to get embeddings, the weighted matrix must be of size X × 300, where:

- X = vocabulary size
- 300 = embedding dimension (hyperparameter)

Initially, all weights are randomly initialized using some initialization technique.

- This weighted matrix multiplication results in a vector embedding of size [1 × 300] for a given word.
- This embedding is then passed to the next layer with a weight matrix of size [300 × X], resulting in a [1 × X] vector.
- From here, we apply soft max → backpropagation → weight update for both matrices, repeatedly, until:
  - The embeddings and weight matrices are fine-tuned
  - The loss is minimized completely

Notes / Clarifications:

- The process you described is essentially CBOW (or Skip-gram) training for Word2Vec.
- The [1 × 300] embedding is learned during training, not fixed.
- Both weight matrices are trainable parameters.
- The "next layer" weight matrix [300 × X] corresponds to predicting target/context words.

DAY-7 : PARTS OF SPEECH TAGGING:

tagging / assigning the related parts of speech for every word present in the document
this is a advanced preprocessing step

HIDDEN MARKOV MODEL :
HMM is a probabilistic model where we observe data indirectly, and we use statistical methods to infer the hidden process behind it.

how pos tagging done in HMM:
we train our model on manually POS tagged sentences in beginning

step1: manual pos tagging on training sets:
1 nithish loves campusx
noun verb noun

2 can nithish google campusx
model noun verb noun

3 will ankita google campusx
model noun verb noun

4 ankita loves will
noun verb noun

5 will loves google
noun verb noun

step2: calculating emission probablity:

word	noun	model	verb
nithish	2	0	0
loves	0	0	2
campusx	3	0	0
google	1	0	2
will	2	1	0
ankita	2	0	0
can	0	1	0

emission probability : calculate the probability of word of being noun , model , verb

word	noun	model	verb
nithish	2/10	0/2	0/5
loves	0/10	0/2	3/5
campusx	3/10	0/2	0/5
google	1/10	0/2	2/5
will	2/10	1/2	0/5
ankita	2/10	0/2	0/5
can	0/10	1/2	0/5

this mean probability of a word is noun, model, verb

ex:

will : has probability of 2/10 of being noun and 1/2as model ans 0/5 as a verb

transition probability : what is the probability next word is noun given current is verb , probability next word is model given current is verb .....

step: at the start and end of each sentence add start and end

$$P(w_{t+1} \mid w_t) = \frac{\text{Count}(w_t, w_{t+1})}{\text{Count}(w_t)}$$

	noun	model	verb	end
start	3	2	0	0
noun	0	0	5	5
model	2	0	0	0
verb	5	0	0	0

probability	noun	model	verb	end
start	3/5	2/5	0	0
noun	0	0	5/10	5/10
model	2/2	0	0	0
verb	5/5	0	0	0

the above table says probability of a noun word after start is 3/5 and getting a verb is 2/5

need of pos tagging :

- **Understanding meaning (NLP preprocessing)**
- Helps the model know if “book” is a noun (“a book”) or a verb (“to book a ticket”).
- Disambiguates words based on context.

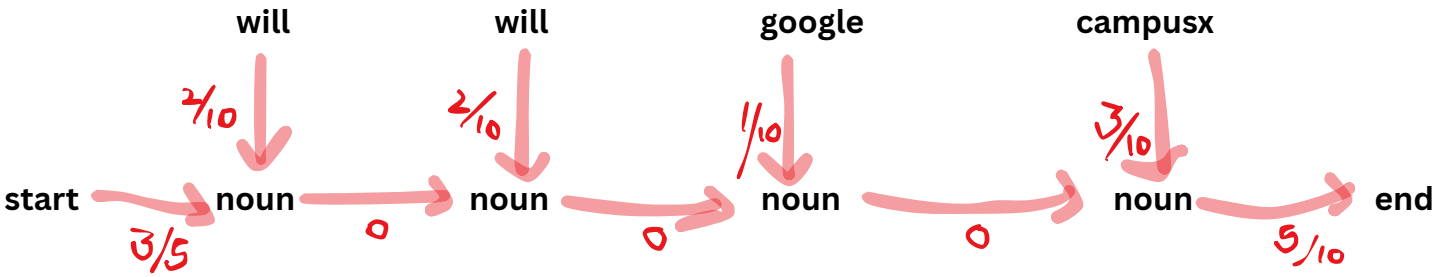
- **Feature for downstream tasks**
- Improves performance of tasks like Named Entity Recognition (NER), Question Answering, Machine Translation, Sentiment Analysis, etc.
- Example: knowing “Apple” is a proper noun vs “apple” a common noun helps in entity detection.

- **Linguistic structure**
- POS tags help identify subject, object, and relations in a sentence → useful in parsing and grammar-based models.

- **Statistical models (like HMM, Viterbi**)
- You model POS tagging as a sequence labeling problem.
- The words are observations; the tags (noun/verb/etc.) are the hidden states.
- Viterbi algorithm is used to find the most probable sequence of tags given the words.

pos tagging of new sentence form our trained model :

will will google campusx

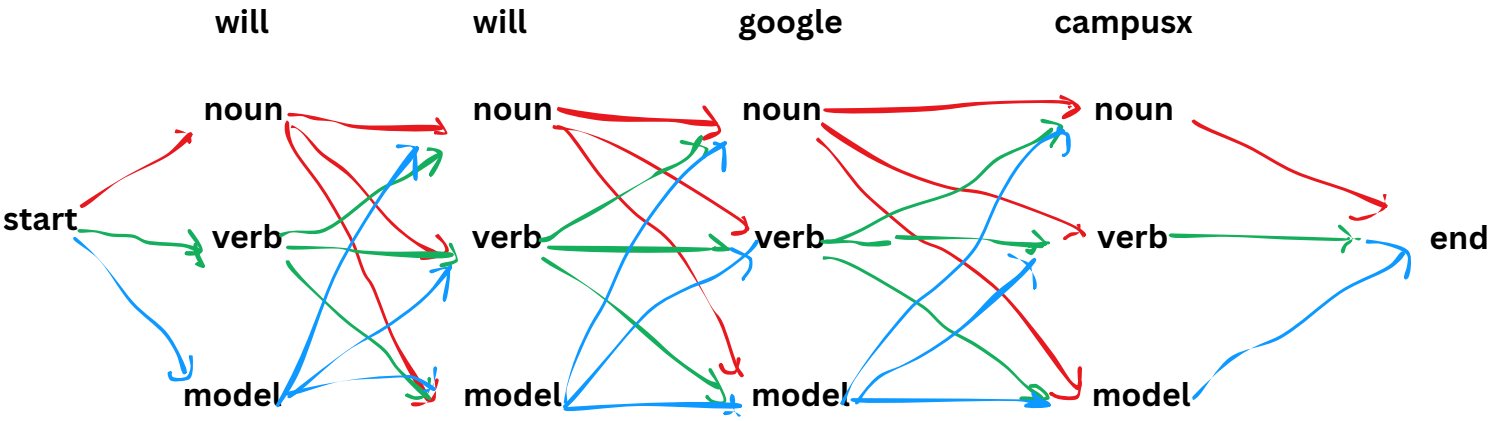


calculate product of probablity : 2/10 *.....* 5/10 of all 9 probability that is =0

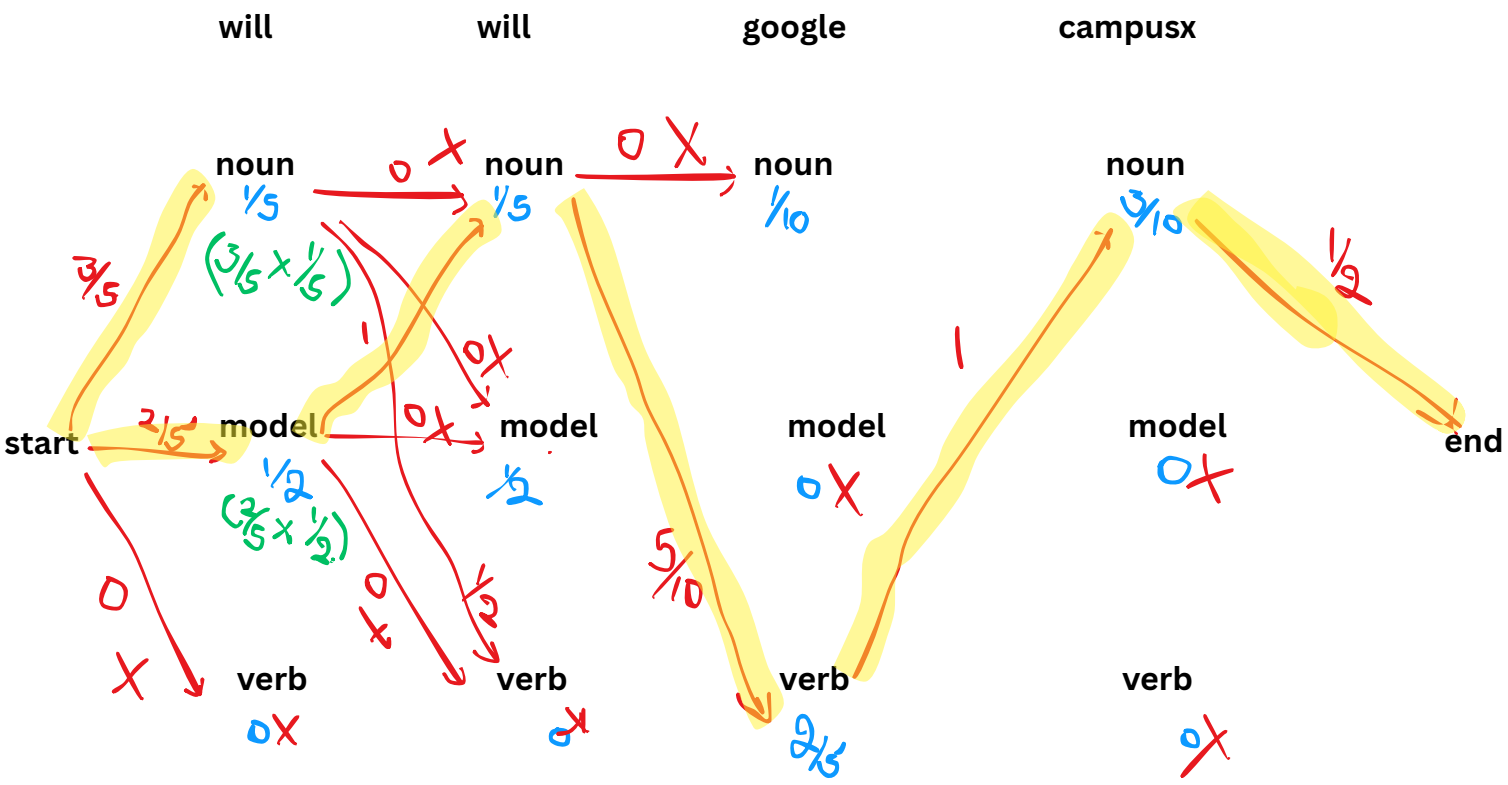
repeat this step for all possible combo and select max probability

probability of this combination is the max and non zero so this is the best POS tagging possible

total no of combination is n power m where n is no of POS and m is the no of words in my sentence



to reduce the time to compute the best possible POS tagging for the given sentence using all possible combination we use optimization technique known as **viterbi optimisation**



in viterbi algo we get the best path in the first run itself , we do backtrack once we reach end to build the the exact path that was traversed if we use greedy search we need to backtrack as there might be another best path

- **Viterbi Algorithm**
- It is dynamic programming.
- At every step it keeps not just the probability of being in a state but also a backpointer to the best previous state.
- By the time you reach the end, you already know the globally best path probability.
- Backtracking here is only for reconstructing the actual sequence of states, not for searching alternatives.
- So: the best path is guaranteed after the forward pass itself.

- **Greedy Search**
- At every step, it just picks the locally best option (the state with the highest immediate probability).
- This may lead to a suboptimal final path because a slightly worse choice earlier could allow for a much better path overall.
- If you backtrack in greedy search, you might discover a better path that was missed.