Deep Learning (DL)

Lab4

Natural Language Processing (NLP)

Preprocessing

HOSSAM FAKHER
Computer vision
Engineer

What is Natural Language Processing(NLP) ?

- How to make language understandable to computers?

- NLP is deeply tied to the sequential nature of language.
- Sequences of words, phrases, or sentences are key to understanding meaning, context, and relationships between different parts of text.

- Then develop algorithms to operate on the input language:
    - CS
    - AI
    - Linguistics اللغويات

What are NLP tasks?

Encoder-Decoder pattern

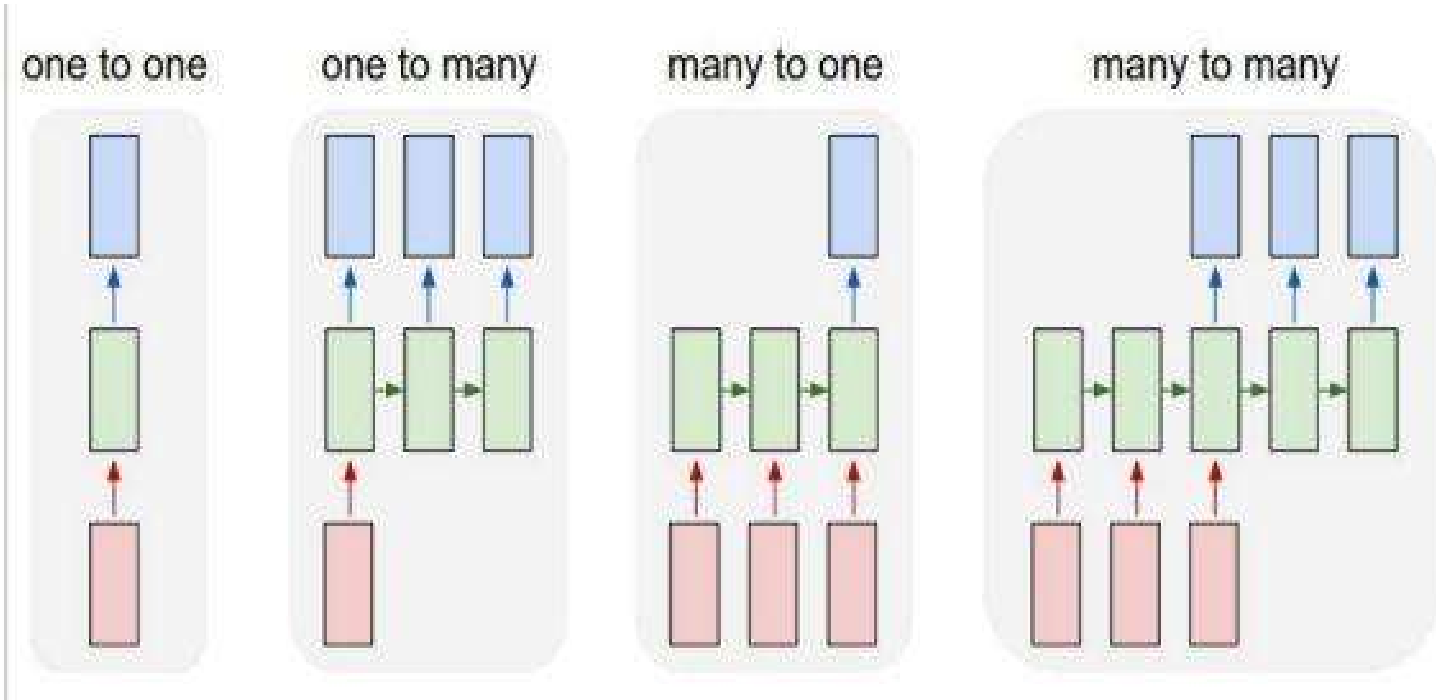


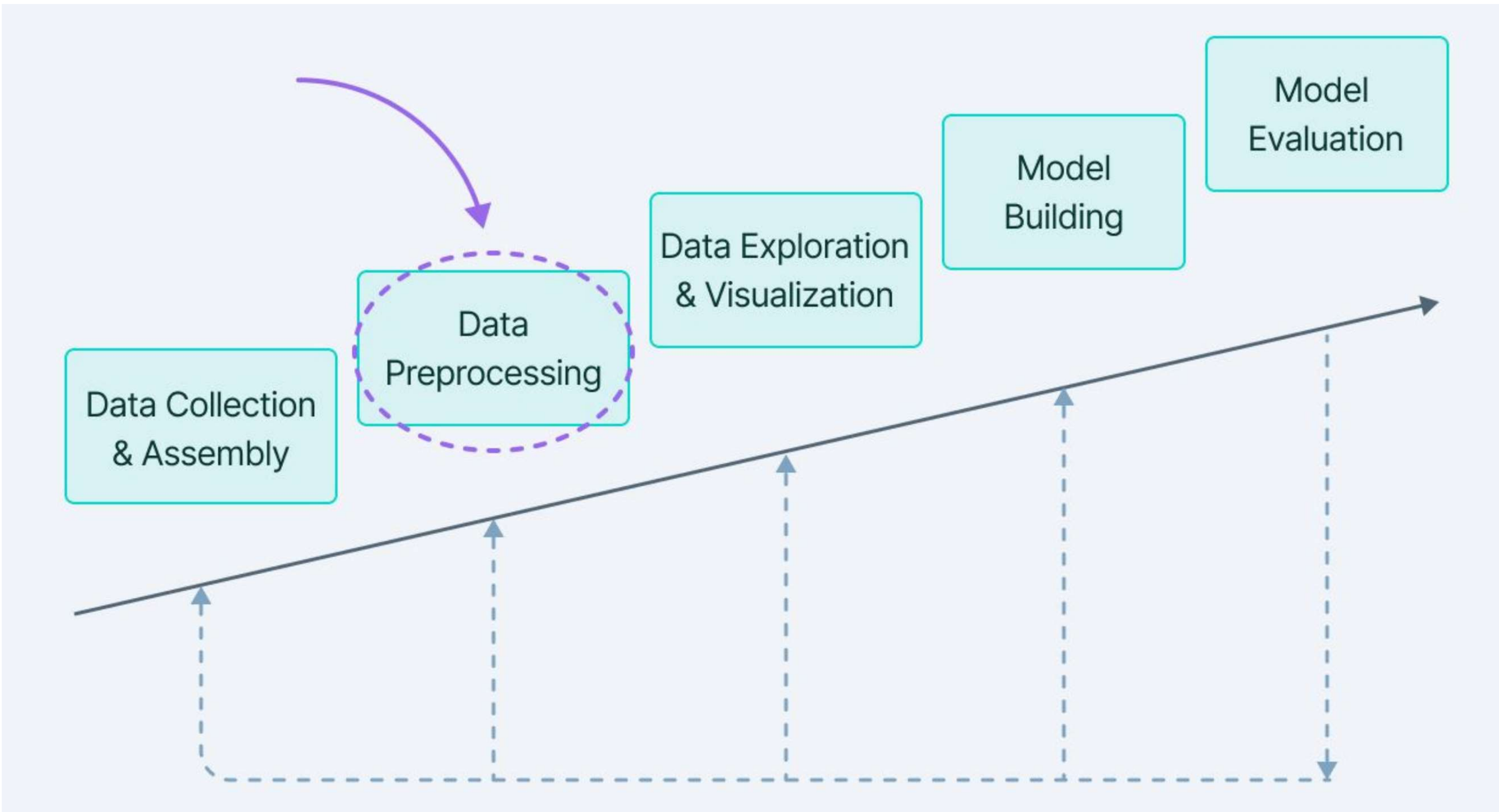
image classification | **image captioning** | **sentiment analysis** | **Machine Translation**

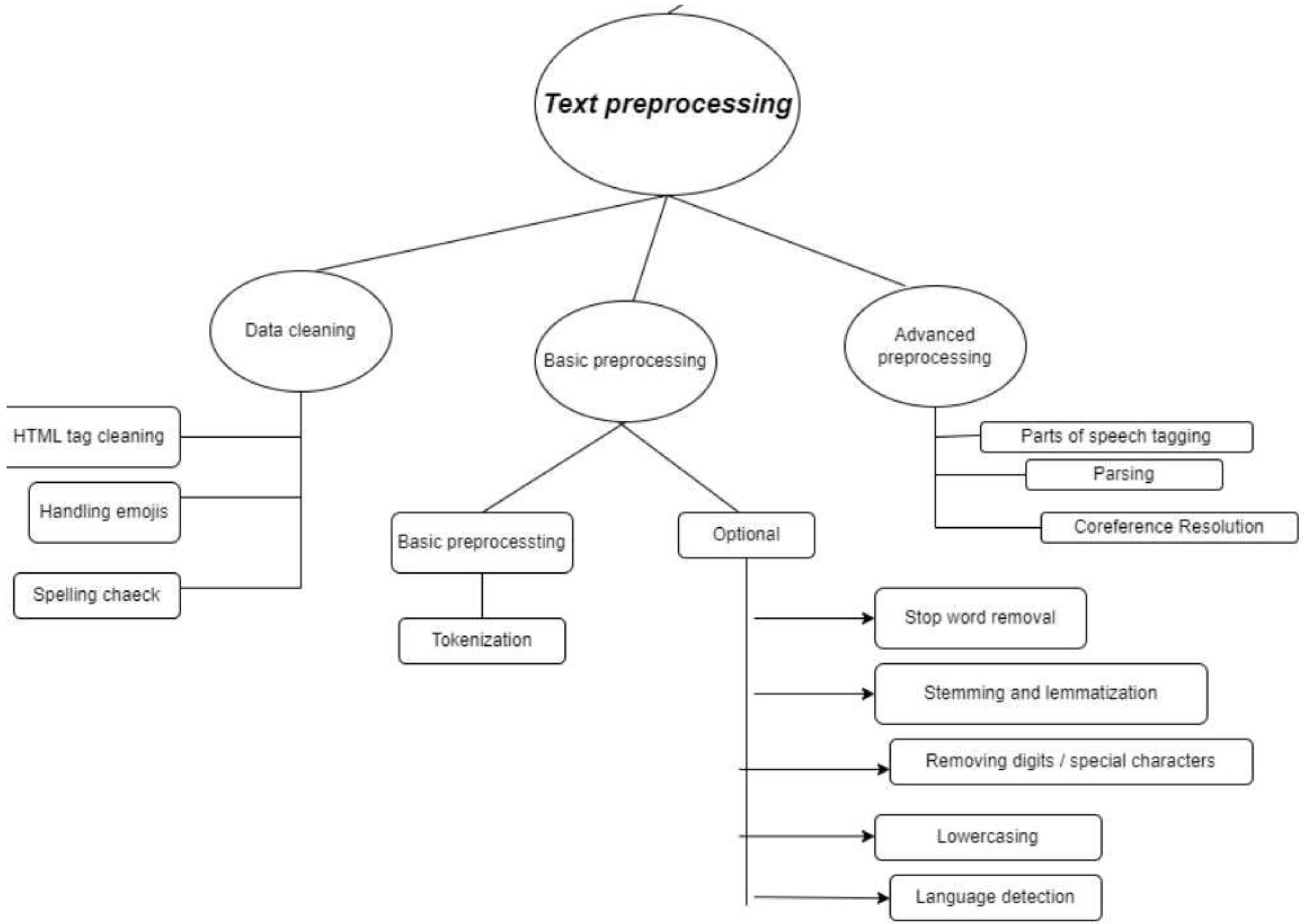
Why NLP is hard?

Human Language is difficult

1. Symbolic: Uses symbols to convey meaning. الرموز
2. Implicit Meaning: Includes indirect cues (e.g., sarcasm). سخرية
3. Multiple Encodings: Same meaning can have different expressions (gestures, emoticons, speech). نفس المعنى
4. Sparse Vocabulary: Huge vocabulary set. كلمات كثيرة
5. Diverse: Encompasses multiple languages, dialects, and accents. لغات ولهجات مختلفة

Cycle of Natural Language Processing




Text preprocessing
Data cleaning
Basic preprocessing
Advanced preprocessing
HTML tag cleaning
Handling emojis
Spelling chaeck
Basic preprocessting
Tokenization
Optional
Stop word removal
Stemming and lemmatization
Removing digits / special characters
Lowercasing
Language detection
Parts of speech tagging
Parsing
Coreference Resolution

Data Cleaning

```python
from tqdm import tqdm
# Initialize tqdm with pandas
tqdm.pandas()
```

```
100%|██████████████████████████████████████████████████| 10000/10000
```

Remove html tags using Regular expressions

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

```
df['review']=df['review'].progress_apply(lambda x : remove_html_tags(x))
df.head()
```

```
text = "<div>Hello World</div> <p>Paragraph</p>"
cleaned_text = remove_html_tags(text)
print(cleaned_text)
```

```
Hello World Paragraph
```

Removing URLs

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

df['review']=df['review'].progress_apply(lambda x : remove_url(x))
df.head()
```

```
text = "Visit our site at http://example.com or https://secure.example.com for more information."
cleaned_text = remove_url(text)
print(cleaned_text)
```

```
Visit our site at  or  for more information.
```

Handling emojis

```
# pip install emoj
import emoji

df['review'] = df['review'].progress_apply(emoji.demojize)
df.head()
```

```
Original Text: I love pizza 🍕 and ice cream 🍦! Let's celebrate! 🎉
Converted Text: I love pizza :pizza: and ice cream :ice_cream:! Let's celebrate! :tada:
```

Remove digits

```
# Define a function to remove digits
def remove_digits(text):
    # Remove digits using regex
    return re.sub(r'\d+', '', text)

df['review'] = df['review'].progress_apply(lambda x: remove_digits(x))
df.head()
```

```
text = "I have 2 apples and 33 oranges."
cleaned_text = remove_digits(text)
print(cleaned_text)
```

```
I have  apples and  oranges.
```

Remove Punctuation

```
import string
```

```
exclude=string.punctuation
exclude
```

```
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

These are the punchuations in python

```
def remove_punc(text):
    for char in exclude:
        text=text.replace(char,'')
    return text

df['review']=df['review'].progress_apply(lambda x : remove_punc(x))
df.head()
```

Remove Punctuation

```
text = "Hello, world! How's it going? Let's meet at 5:00 p.m."
cleaned_text = remove_punc(text)
print(cleaned_text)
```

```
Hello world Hows it going Lets meet at 500 pm
```

Spelling correction

```
#!pip install textblob
from textblob import TextBlob
def check_spelling(text):
    textblb=TextBlob(text)
    return textblb.correct().string

df['review']=df['review'].progress_apply(lambda x : check_spelling(x))
df.head()
```

```
text = "I havv goood speling!"
corrected_text = correct_spelling(text)
print(corrected_text)
```

```
I have good spelling!
```

Removing StopWords

```
from nltk.corpus import stopwords
```

```
StopWords = stopwords.words("english")
StopWords
```

```
['i',
 'me',
 'my',
 'myself',
 'we',
 'our',
 'ours',
 'ourselves',
 'you',
 "you're",
 "you've",
 "you'll",
 "you'd",
 'your',
 'yours',
 'yourself',
```

```
def remove_stopwords(text):
    filtered_text = ' '.join(word for word in text.split() if word.lower() not in StopWords)
    return filtered_text

df['review']=df['review'].progress_apply(lambda x : remove_stopwords(x))
df.head()
```

```
text = "This is an example of a sentence with stop words."
cleaned_text = remove_stopwords(text)
print(cleaned_text)
```

```
example sentence stop words.
```

Stemming & Lemmatization

Stemming is the process of shortening words by removing suffixes, often resulting in non-precise words. It relies on simple rules, such as removing suffixes, without considering the meaning of the word.

Lemmatization is the process of converting a word to its correct linguistic base form, called the "lemma," considering the word's meaning and context.
Lemmatization usually requires advanced libraries and grammar rules.

Key Difference

- **Stemming** is faster but may yield imprecise roots.

- **Lemmatization** is more accurate and considers grammatical meaning, but it is slower.

```
from nltk.stem import PorterStemmer

ps=PorterStemmer()
def stem_words(text):
    return " ".join([ps.stem(word) for word in text.split()])
```

```
sample='running ran runner easily fairly'
stem_words(sample)
```

```
'run ran runner easili fairli'
```

```
#df['review'] = df['review'].progress_apply(stem_words)
#df.head()
```

```
from nltk.stem import WordNetLemmatizer

le=WordNetLemmatizer()
def lemm_words(text):
    return " ".join([le.lemmatize(word) for word in text.split()])
```

```
sample='running ran runner easily fairly'
lemm_words(sample)
```

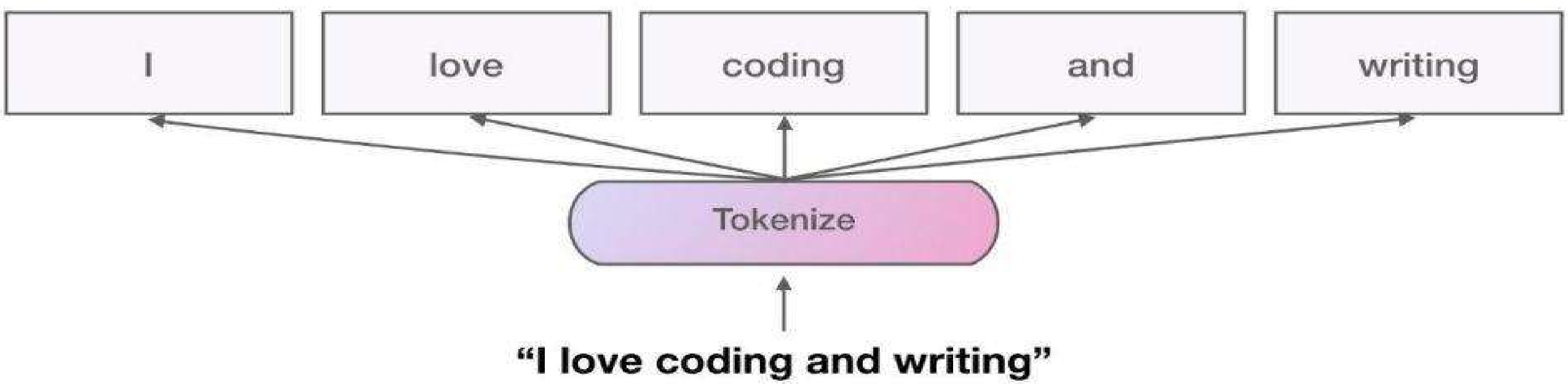
```
'running ran runner easily fairly'
```

```
#df['review'] = df['review'].progress_apply(lemm_words)
#df.head()
```

```
Original Words: ['running', 'ran', 'runner', 'easily', 'fairly']
After Stemming: ['run', 'ran', 'runner', 'easili', 'fairli']
After Lemmatization: ['run', 'run', 'runner', 'easily', 'fairly']
```

Tokenization

- **Tokenization** Splitting the text into individual words or tokens. For example, "Hello world!" becomes ["Hello", "world", "!"].
- The final Goal of Tokenization is : Creating Vocabulary
  - Python's split() function
  - Regular Expressions (RegEx)
  - NLTK library
  - SpaCy library
  - Gensim library



Tokenization

```
#NLTK
from nltk.tokenize import word_tokenize
```

```
X_train['Tokenization'] = X_train['review'].progress_apply(word_tokenize)
X_train.head()
```

```
# Texts to Sequences
from tensorflow import keras
tokenizer = keras.preprocessing.text.Tokenizer()
tokenizer.fit_on_texts(data['text_nonStopwords'])
data['text_sequences'] = tokenizer.texts_to_sequences(data['text_nonStopwords'])
```

Part-of-Speech (POS)

- **Part-of-Speech (POS) Tagging** is the process of identifying the grammatical category of each word in a text, such as noun, verb, adjective, etc.
- POS tagging is crucial in Natural Language Processing (NLP) as it helps understand the structure and meaning of sentences.
- Each word in a sentence is assigned a tag representing its part of speech, which provides information about the word's role in the sentence.

```
import nltk

# Download NLTK data if not already installed
nltk.download('averaged_perceptron_tagger')
nltk.download('punkt')

# Example sentence
sentence = "The quick brown fox jumps over the lazy dog."

# Tokenize the sentence
words = nltk.word_tokenize(sentence)

# Apply POS tagging
pos_tags = nltk.pos_tag(words)

print("Word and POS tags:", pos_tags)
```

- POS tags might look like:
  - **The**: Determiner (DT)
  - **quick**: Adjective (JJ)
  - **brown**: Adjective (JJ)
  - **fox**: Noun (NN)
  - **jumps**: Verb (VBZ)
  - **over**: Preposition (IN)
  - **the**: Determiner (DT)
  - **lazy**: Adjective (JJ)
  - **dog**: Noun (NN)

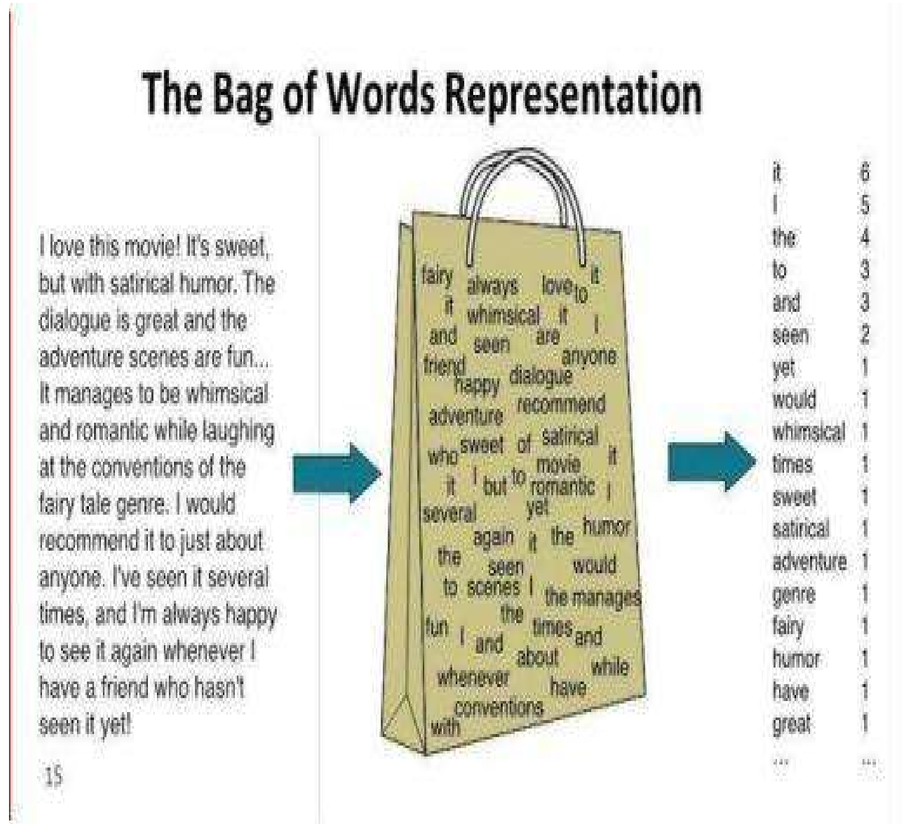
Explanation of Some POS Tags

- **NN**: Noun (e.g., "dog", "fox")
- **VB**: Verb (e.g., "jump", "run")
- **JJ**: Adjective (e.g., "quick", "brown")
- **DT**: Determiner (e.g., "the", "a")
- **IN**: Preposition (e.g., "over", "in")

Bag of Words (BoW)

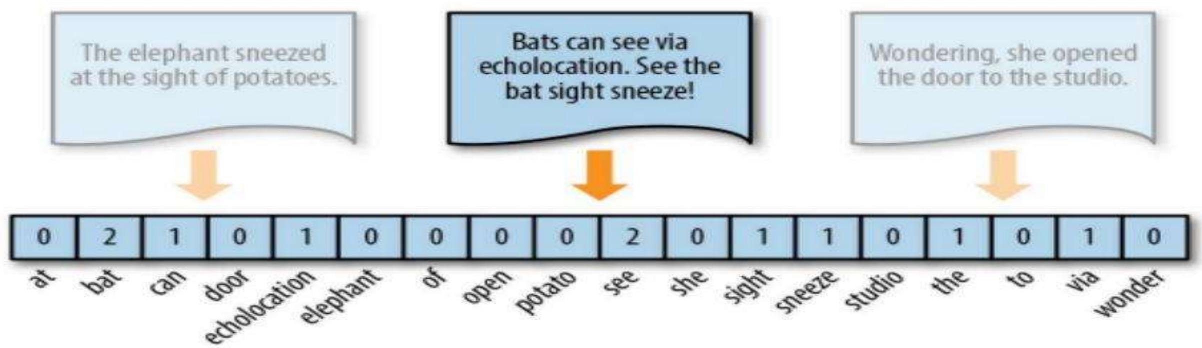
Bag of Words (BoW) is represented as a collection of words without considering grammar, order, or structure. This method focuses on the presence and frequency of words, making it a simple yet effective approach for converting text into numerical features for machine learning models.

- **Tokenization**: Break down each document into individual words (tokens).
- **Vocabulary Creation**: Create a set of unique words (vocabulary) across all documents in the dataset.
- **Vectorization**: Represent each document as a vector of word counts based on the vocabulary.



Vectorization

1. Vectorizing : The process that we use to convert text to a form that Python and a machine learning model can understand
2. It is defined as the process of encoding text as integers to create feature vectors
3. Build vocab
4. Register the index of the word from the vocab
5. Not in vocab? □ store as UNK
6. Need to pad? □ encode as PAD = 0



There are many vectorization techniques, we will focus on the three widely used vectorization techniques:

- Count vectorization
- N-Grams.
- Term frequency - inverse document frequency (TF-IDF)

1- Count vectorization- Document term matrix

- This means, if a particular word appears many times in spam or ham message ,then the particular word has a high predictive power of determining if the message is a spam or ham .

- NLP is interesting , NLP is good
- Don’t like NLP
- good subject

NLP	is	interesting	Don't	like	good	subject
2	2	1	0	0	1	0
1	0	0	1	1	0	0
0	0	0	0	0	1	0

1- Count vectorization- Document term matrix

```
from sklearn.feature_extraction.text import CountVectorizer
```

```
vectorizer = CountVectorizer(
    lowercase=True,                    # Converts all characte
    stop_words=None,                   # You can provide a lis
    max_features=10000,                # all tokens will be in
    vocabulary=None,                   # a vocabulary will be
    binary=False                       # If False, the count c
)

# Transform the text data into a document-term matrix
X_tr = vectorizer.fit_transform(X_train['review'])
X_te = vectorizer.transform(X_test['review'])
```

```
print(vectorizer.vocabulary_)
```

1- Count vectorization- Document term matrix

```
vocab = vectorizer.get_feature_names_out()
vocab
```

```
array(['aaron', 'abandon', 'abandoned', ...,
```

```
vocab.shape
```

```
(10000,)
```

```
X_train1 = pd.DataFrame(X_tr.toarray(), columns=vocab)

X_train1.head()
```

```
X_test1 = pd.DataFrame(X_te.toarray(), columns=vocab)

X_test1.head()
```

2- N-gram vectorizing

- The n-grams process creates a document-term matrix like we saw before. Now we still have one row per text message and we still have counts that occupy the individual cells but instead of the columns representing single terms ,here ;all combinations of adjacent words of length and in your text.

As an example, let's use the string Natural Language Processing.

Natural Language Processing

Unigrams:	Natural, Language, Processing
Bigrams:	Natural Language, Language Processing
Trigrams:	Natural Language Processing

2.1 N-gram vectorizing(1,2)

```
from sklearn.feature_extraction.text import CountVectorizer
```

```
vectorizer = CountVectorizer(
    lowercase=True,                    # Converts all characte
    stop_words=None,                   # You can provide a lis
    max_features= None,                # all tokens will be in
    ngram_range=(1, 2),                # Use unigrams (1 word)
    vocabulary=None,                   # a vocabulary will be
    binary=False                       # If False, the count o
)

# Transform the text data into a document-term matrix
X_tr = vectorizer.fit_transform(X_train['review'])
X_te = vectorizer.transform(X_test['review'])
```

2.2 N-gram vectorizing(2,2)

```
from sklearn.feature_extraction.text import CountVectorizer
```

```
vectorizer = CountVectorizer(
    lowercase=True,                    # Converts all character
    stop_words=None,                   # You can provide a list
    max_features= None,                # all tokens will be inc
    ngram_range=(2, 2),
    vocabulary=None,                   # a vocabulary will be c
    binary=False                       # If False, the count of
)

# Transform the text data into a document-term matrix
X_tr = vectorizer.fit_transform(X_train['review'])
X_te = vectorizer.transform(X_test['review'])
```

2.3 N-gram vectorizing(1,3)

```
from sklearn.feature_extraction.text import CountVectorizer
```

```
vectorizer = CountVectorizer(
    lowercase=True,                     # Converts all character
    stop_words=None,                    # You can provide a list
    max_features= None,                 # all tokens will be inc
    ngram_range=(1, 3),                 # Use unigrams (1 word)
    vocabulary=None,                    # a vocabulary will be c
    binary=False                        # If False, the count of
)

# Transform the text data into a document-term matrix
X_tr = vectorizer.fit_transform(X_train['review'])
X_te = vectorizer.transform(X_test['review'])
```

3. Term frequency - inverse document frequency (TF-IDF)

- TF-IDF creates a document term matrix, where there's still one row per text message and the columns still represent single unique terms.
- But instead of the cells representing the count, the cells represent a weighting that's meant to identify how important a word is to an individual text message.
- **weighting = TF*IDF**

Term Frequency (TF): This measures how frequently a term appears in a document. It is often calculated as:

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

Inverse Document Frequency (IDF): This measures how important a term is across the entire corpus. It helps to down weight common words that appear in many documents. IDF is calculated as:

$$\mathrm{IDF}(t) = \log\left(\frac{\text{Total number of documents}}{\text{Number of documents containing term } t}\right)$$

TF-IDF Calculation: The TF-IDF score for a term is the product of its TF and IDF scores:

$$\text{TF-IDF}(t,d) = \mathrm{TF}(t,d) \times \mathrm{IDF}(t)$$

3. TF-IDF- How to Compute:

Documents

D1: "the cat sat on the mat“ || D2: "the dog sat on the log“ || D3: "the cat chased the dog"

Step 1: Calculate Term Frequencies (TF)

Let's calculate the TF for the term "cat" in each document.

- **TF("cat", d1):**
  - Occurrences = 1
  - Total words = 6
  - TF("cat", d1) = $\frac{1}{6} = 0.167$
- **TF("cat", d2):**
  - Occurrences = 0
  - Total words = 6
  - TF("cat", d2) = $\frac{0}{6} = 0$
- **TF("cat", d3):**
  - Occurrences = 1
  - Total words = 5
  - TF("cat", d3) = $\frac{1}{5} = 0.2$

3. TF-IDF- How to Compute:

Documents

D1: "the cat sat on the mat“ || D2: "the dog sat on the log“ || D3: "the cat chased the dog"

Step 2: Calculate Inverse Document Frequency (IDF)

We already calculated the **IDF** for "cat" previously:

- Total documents (N) = 3
- Documents containing "cat" = 2 (d1 and d3)

$$\mathrm{IDF}(\text{"cat"}) = \log\left(\frac{3}{2}\right) \approx 0.176$$

3. TF-IDF- How to Compute:

$$\text{IDF}(\text{"cat"}) = \log\left(\frac{3}{2}\right) \approx 0.176$$

Step 3: Calculate TF-IDF

Now, let's recalculate the **TF-IDF** for the term "cat" in each document.

- **TF-IDF("cat", d1)**:

$$\text{TF-IDF}(\text{"cat"}, \text{d1}) = 0.167 \times 0.176 = 0.0294$$

- **TF-IDF("cat", d2)**:

$$\text{TF-IDF}(\text{"cat"}, \text{d2}) = 0 \times 0.176 = 0$$

- **TF-IDF("cat", d3)**:

$$\text{TF-IDF}(\text{"cat"}, \text{d3}) = 0.2 \times 0.176 = 0.0352$$

3. Term frequency - inverse document frequency (TF-IDF)

```
from sklearn.feature_extraction.text import TfidfVectorizer
```

```
tfidf_vectorizer = TfidfVectorizer(lowercase=True,
                                    tokenizer=None,
                                    stop_words=None,
                                    ngram_range=(1, 1),
                                    max_features=None,
                                    vocabulary=None,
                                    binary=False)

# Transform the text data into a document-term matrix
X_tr = tfidf_vectorizer.fit_transform(X_train['review'])
X_te = tfidf_vectorizer.transform(X_test['review'])
```

```
vocab = tfidf_vectorizer.get_feature_names_out()
vocab
```

```
array(['aaa', 'aaaarrgh', 'aaahthe', ..., 'über', 'überwoman',
       'ünfaithful'], dtype=object)
```

```
X_train2 = pd.DataFrame(X_tr.toarray(), columns=vocab)

X_train2.head()
```

```
X_test2 = pd.DataFrame(X_te.toarray(), columns=vocab)

X_test2.head()
```

Thanks