# Natural Language Processing(NLP):

Its a branch of AI that focuses on the interaction between computers and humans using natural language. The goal is to enable machines to understand, interpret, & generate human language in a way that its both meaningful & useful. Its a blend of lingusitic, computer science, & ML, maing capable of processing and analyzing large amount of natural language data.

---

**Key Components:**

- Tokenization
- Part of Speech
- Name Entity Recognition(NER)
- Sentiment Analysis
- Stemming and Lemmatization
- Stop words removal
- Dependency
- Topic Modeling

**Common Applications:**

- Chatbot & Virtual Assistance
- Sentiment Analysis
- Text Summarization
- Speech Recognition
- Spam Detection
- Predictive text

**Popular NLP Libraries:**

- NLTK
- SpaCy
- TextBlob
- Genism
- Transformers

**Techniques used in NLP**

- Bag of Words(BoW)
- TF-IDF(Term Frequency-Inverse Document Frequency)
- Word Embedding
- Recurrent Neural Network(RNN)
- Transformers

**NLP Framework:**

- NLTK
- SpaCy
- Gensim

---

# Components of NLP

- **Natural Language Understanding(NLU)**
  - tokens, bigram, trigram, ngram, stem, porter stemmer, lancaster, stemmer, snowball stemmer, lemmatization, stopword, POS, NER.
- **Natural Language Generation(NLG)**
  - WordCloud + Matplotlib (Multiple Visualization)

```
# Download NLTK to your local system

import os
import nltk
nltk.download('all')
```

Show hidden output

# 1. Tokenization

Tokenization in NLP is the process of breaking down a text into smaller units called tokens. Tokens can be words, characters, or subwords, depending on the specific task and approach. Tokenization is an essential step in text preprocessing for NLP tasks because it allows algorithms to parse and understand the input text by converting it into manageable pieces.

**Types of Tokeniation**

1. Word Tokenization
2. Sentence Tokenization
3. Character Tokenization
4. Subword Tokenization

**Why Tokenization is Important**

1. Prepares text for analysis: Tokenization converts unstructured text data into a structured format that algorithms can process.
2. Maintains context: Proper tokenization helps preserve the meaning of the text by ensuring punctuation and special characters are handled correctly.
3. Enables feature extraction: Tokens are often used as features in NLP models for tasks such as text classification, sentiment analysis, and more.

**Challenges in Tokenization**

1. Ambiguity: Tokenizers must handle cases where word boundaries are not clear (e.g., "I'm" could be tokenized as ["I", "'m"]).
2. Language Differences: Tokenization varies widely between languages. For example, Chinese and Japanese use no spaces between words, requiring more complex tokenization logic.
3. Punctuation: Deciding how to handle punctuation (e.g., splitting "U.S." or "don't" correctly).
4. Compound Words: Languages with compound words (e.g., German) present unique challenges for word tokenization.

```
AI = "Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever

AI
```

```
'Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s. '
```

**Word Tokenizer**

- Splits text into individual words.
- It uses spaces and punctation as delimiters to identify words. However it generally removes punctuation marks and treats them as separate tokens.

```
from nltk.tokenize import word_tokenize

word_tokens=word_tokenize(AI)
print(word_tokens)
print('\n')
print("Length of Words got tokenized", len(word_tokens))
```

```
['Lorem', 'Ipsum', 'is', 'simply', 'dummy', 'text', 'of', 'the', 'printing', 'and', 'typesetting', 'industry', '.', 'Lorem', 'Ipsum', 'ha


Length of Words got tokenized 28
```

**Sentence Tokenizer**

- Splits text into individual sentences.
- It identifies sentence boundaries using punctuation marks like periods (.), question marks (?), and exclamation marks (!). It also considers capitalization and abbreviations to avoid incorrect splits.

```
from nltk.tokenize import sent_tokenize

sent_tokens=sent_tokenize(AI)
print(sent_tokens,'\n')
print("Length of Sentences got tokenized", len(sent_tokens))
```

```
['Lorem Ipsum is simply dummy text of the printing and typesetting industry.', "Lorem Ipsum has been the industry's standard dummy text ev

Length of Sentences got tokenized 2
```

**Character Tokenizer**

- Splits text into individual characters.
- It treats each character as a separate token, include spaces and punctuation.

```
Ch_text="Hello World!"

ch_tokens=list(Ch_text)
print(ch_tokens)

print('\n')
print("Length of characters got tokenized", len(ch_tokens))
```

```
['H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd', '!']


    Length of characters got tokenized 12
```

**Whitespace Tokenizer**

- Splits text based on whitespaces(spaces, tabs, and newlines).
- It treats any sequence of whitespace character as a delimiter. Punctuation is included in the tokens if its attached to a wword.

```
from nltk.tokenize import WhitespaceTokenizer

ws_tokens=WhitespaceTokenizer().tokenize(AI)

print(ws_tokens,'\n')
print("Length of words got tokenized", len(ws_tokens))
```

```
['Lorem', 'Ipsum', 'is', 'simply', 'dummy', 'text', 'of', 'the', 'printing', 'and', 'typesetting', 'industry.', 'Lorem', 'Ipsum', 'has',

Length of words got tokenized 25
```

**WordPunct Tokenizer**

- Splits text into words and punctuation separately.
- It breaks the text into tokens where words are separated by punctuation, including punctuation marks as separate tokens.

```
# WordPunct tokenization

S='Good apple cost is $3.88.'
from nltk.tokenize import wordpunct_tokenize
wp=wordpunct_tokenize(S)

wp
```

```
['Good', 'apple', 'cost', 'is', '$', '3', '.', '88', '.']
```

| Tokenizer Type | Description | Example Output |
|---|---|---|
| **Word Tokenizer** | Splits by words (spaces & punctuation) | `['Hello', ',', 'world', '!']` |
| **Sentence Tokenizer** | Splits by sentences | `['Hello world.', 'How are you?']` |
| **Character Tokenizer** | Splits into individual characters | `['H', 'e', 'l', 'l', 'o', '!']` |
| **Whitespace Tokenizer** | Splits by whitespace | `['Hello,', 'world!', 'How', 'are', 'you?']` |
| **WordPunct Tokenizer** | Splits by words & punctuation separately | `['Hello', ',', 'world', '!', 'How', "'", 's', 'going', '?']` |

**Ngrams**

```
# Unigram

uni=word_tokenize(AI)
uni
```

```
['Lorem',
 'Ipsum',
 'is',
 'simply',
 'dummy',
 'text',
 'of',
 'the',
 'printing',
 'and',
 'typesetting',
 'industry',
 '.',
 'Lorem',
 'Ipsum',
 'has',
 'been',
 'the',
 'industry',
 "'s",
 'standard',
 'dummy',
 'text',
 'ever',
 'since',
 'the',
 '1500s',
 '.']
```

```
# Bigram

from nltk.util import bigrams

bigr=list(bigrams(uni))
bigr
```

```
[('Lorem', 'Ipsum'),
 ('Ipsum', 'is'),
 ('is', 'simply'),
 ('simply', 'dummy'),
 ('dummy', 'text'),
 ('text', 'of'),
 ('of', 'the'),
 ('the', 'printing'),
 ('printing', 'and'),
 ('and', 'typesetting'),
 ('typesetting', 'industry'),
 ('industry', '.'),
 ('.', 'Lorem'),
 ('Lorem', 'Ipsum'),
 ('Ipsum', 'has'),
 ('has', 'been'),
 ('been', 'the'),
 ('the', 'industry'),
 ('industry', "'s"),
 ("'s", 'standard'),
 ('standard', 'dummy'),
 ('dummy', 'text'),
 ('text', 'ever'),
 ('ever', 'since'),
 ('since', 'the'),
 ('the', '1500s'),
 ('1500s', '.')]
```

```
# Trigram

from nltk.util import trigrams

trigr=list(trigrams(uni))
trigr
```

```
[('Lorem', 'Ipsum', 'is'),
 ('Ipsum', 'is', 'simply'),
 ('is', 'simply', 'dummy'),
 ('simply', 'dummy', 'text'),
 ('dummy', 'text', 'of'),
 ('text', 'of', 'the'),
 ('of', 'the', 'printing'),
 ('the', 'printing', 'and'),
 ('printing', 'and', 'typesetting'),
 ('and', 'typesetting', 'industry'),
 ('typesetting', 'industry', '.'),
 ('industry', '.', 'Lorem'),
 ('.', 'Lorem', 'Ipsum'),
 ('Lorem', 'Ipsum', 'has'),
 ('Ipsum', 'has', 'been'),
 ('has', 'been', 'the'),
 ('been', 'the', 'industry'),
```

```
('the', 'industry', "'s"),
('industry', "'s", 'standard'),
("'s", 'standard', 'dummy'),
('standard', 'dummy', 'text'),
('dummy', 'text', 'ever'),
('text', 'ever', 'since'),
('ever', 'since', 'the'),
('since', 'the', '1500s'),
('the', '1500s', '.')]
```

```
# ngrams 6

from nltk.util import ngrams

sixgr=list(ngrams(uni,6))
sixgr
```

```
[('Lorem', 'Ipsum', 'is', 'simply', 'dummy', 'text'),
 ('Ipsum', 'is', 'simply', 'dummy', 'text', 'of'),
 ('is', 'simply', 'dummy', 'text', 'of', 'the'),
 ('simply', 'dummy', 'text', 'of', 'the', 'printing'),
 ('dummy', 'text', 'of', 'the', 'printing', 'and'),
 ('text', 'of', 'the', 'printing', 'and', 'typesetting'),
 ('of', 'the', 'printing', 'and', 'typesetting', 'industry'),
 ('the', 'printing', 'and', 'typesetting', 'industry', '.'),
 ('printing', 'and', 'typesetting', 'industry', '.', 'Lorem'),
 ('and', 'typesetting', 'industry', '.', 'Lorem', 'Ipsum'),
 ('typesetting', 'industry', '.', 'Lorem', 'Ipsum', 'has'),
 ('industry', '.', 'Lorem', 'Ipsum', 'has', 'been'),
 ('.', 'Lorem', 'Ipsum', 'has', 'been', 'the'),
 ('Lorem', 'Ipsum', 'has', 'been', 'the', 'industry'),
 ('Ipsum', 'has', 'been', 'the', 'industry', "'s"),
 ('has', 'been', 'the', 'industry', "'s", 'standard'),
 ('been', 'the', 'industry', "'s", 'standard', 'dummy'),
 ('the', 'industry', "'s", 'standard', 'dummy', 'text'),
 ('industry', "'s", 'standard', 'dummy', 'text', 'ever'),
 ("'s", 'standard', 'dummy', 'text', 'ever', 'since'),
 ('standard', 'dummy', 'text', 'ever', 'since', 'the'),
 ('dummy', 'text', 'ever', 'since', 'the', '1500s'),
 ('text', 'ever', 'since', 'the', '1500s', '.')]
```

## 2. Stemming

Process in NLP where words are reduced to their root form or base by removing suffixes and prefixes. The goal is to group words with the same root meaning together. This is particularly useful in tasks like text mining, search engines, and information retrieval to ensure that different forms of a word(eg., "running", "runs", "runner") are treated as the same base word ("run").

**How Stemming Works:**

Stemming algorithms use heuristic rules to chop off the ends of words. This process can sometimes produce stems that are not valid words in the language but still capture the core meaning.

**Common Stemming Algorithms:**

1. Porter Stemmer
2. Lancaster Stemmer
3. Snowball Stemmer

```
# Sample words to stem
words = ["running", "runner", "runs", "easily", "fairly"]
```

**1. Porter Stemmer:**

It uses a series of rules to reduce words to their stems. It is considered relatively conservative, meaning it usually produces stems that are still recognizable and readable.

Strength:

- Strikes a good balance between reducing words and maintaining readability.

```
from nltk.stem import PorterStemmer
PortStem=[]

for i in words:
  PortStem.append(PorterStemmer().stem(i))

print('Original Words:',words,'\n')
print('Stemmed Words:',PortStem)
```

```
Original Words: ['running', 'runner', 'runs', 'easily', 'fairly']

Stemmed Words: ['run', 'runner', 'run', 'easili', 'fairli']
```

**2. Lancaster Stemmer:**

Also known as the Paice-Husk Stemmer, it is a more aggressive stemming algorithm compared to Porter. It tends to produce shorter stems, often making the output less readable. It uses a more extensive set of rules.

Strength:

- Very aggressive in reducing words to their stems.

```
from nltk.stem import LancasterStemmer
LancStem=[]

for i in words:
  LancStem.append(LancasterStemmer().stem(i))

print('Original Words:',words,'\n')
print('Stemmed Words:',LancStem)
```

```
Original Words: ['running', 'runner', 'runs', 'easily', 'fairly']

Stemmed Words: ['run', 'run', 'run', 'easy', 'fair']
```

**3. Snowball Stemmer:**

Also known as the Porter2 Stemmer, it is an improved version of the original Porter Stemmer. It provides better stemming and supports multiple languages (e.g., English, French, German, etc.). It is more flexible and refined than the Porter Stemmer.

Strength:

- Balanced and more linguistically accurate than Porter, with support for multiple languages.

```
from nltk.stem import SnowballStemmer
SnowStem=[]

for i in words:
  SnowStem.append(SnowballStemmer('english').stem(i))

print('Original Words:',words,'\n')
print('Stemmed Words:',SnowStem)
```

```
Original Words: ['running', 'runner', 'runs', 'easily', 'fairly']

Stemmed Words: ['run', 'runner', 'run', 'easili', 'fair']
```

**Stemming vs. Lemmatization**

While stemming cuts off word endings using a set of rules, lemmatization is a more advanced technique that reduces words to their base or dictionary form (lemma) using linguistic knowledge. For example, "running" would be lemmatized to "run" using its part of speech, while stemming would also produce "run" but might not work as accurately across various contexts.

## 3. Lemmatization

Text normalization technique in NLP that reduces words to their base or dictionary form, called a lemma. This process ensures that different forms of a word are treated as the same, which is useful for tasks such as information retrieval, text mining, and machine learning. Unlike stemming, which blindly trims suffixes based on rules, lemmatization considers the meaning and context of the word and requires a part-of-speech (POS) tag to return accurate lemmas. For instance, the word “running” would be lemmatized to “run”, and “better” to “good” when provided with the right POS context.

**Types:**

1. Dictionary-Based
2. Rule-Based
3. Statistical
4. ML

**1. Dictionary-Based Lemmatization**

This method uses a pre-built dictionary or lexical database to look up the lemma of a word. It maps inflected forms of a word to their corresponding base forms. Words are matched against entries in the dictionary, and if found, they are replaced with their lemmas.

- Strengths: Simple and fast if the dictionary is comprehensive. Works well for common words.

- Weaknesses: Limited by the coverage of the dictionary. May not handle new or rare words.

```
from nltk.stem import WordNetLemmatizer
db_lemma=[]

for i in words:
  db_lemma.append(WordNetLemmatizer().lemmatize(i))

print('Original Words:',words,'\n')
print('Lemmatized Words:',db_lemma)
```

```
Original Words: ['running', 'runner', 'runs', 'easily', 'fairly']

Lemmatized Words: ['running', 'runner', 'run', 'easily', 'fairly']
```

**Note:** By default, WordNet Lemmatizer assumes the part of speech as a noun, which can affect the result. You can specify the POS to improve accuracy.

```
from nltk.stem import WordNetLemmatizer
from nltk.corpus import wordnet

db_pos_lemma=[]

# Sample words to lemmatize
pos_words = [("running", wordnet.VERB), ("better", wordnet.ADJ), ("geese", wordnet.NOUN)]

for i,pos in pos_words:
  db_pos_lemma.append(WordNetLemmatizer().lemmatize(i,pos))

print('Original Words:',pos_words,'\n')
print('Lemmatized Words:',db_pos_lemma)
```

```
Original Words: [('running', 'v'), ('better', 'a'), ('geese', 'n')]

Lemmatized Words: ['run', 'good', 'goose']
```

## 4. Stopwords

Common words in a language that are usually filtered out during text preprocessing in NLP tasks. These words do not contribute significant meaning to a sentence and are often removed to focus on the more informative parts of the text. Examples of stopwords in English include “the,” “is,” “in,” “and,” “a,” “an,” etc.

**Why Remove Stopwords?**

- Reduces Noise: Stopwords often add little to no value in NLP tasks such as text classification, sentiment analysis, or keyword extraction.
- Improves Efficiency: By removing these words, text processing algorithms work on fewer words, improving processing time and resource usage.
- Reduces Dimensionality: It helps in reducing the feature space, which is beneficial for machine learning models.

```
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize

#Sample Text
text="This is an example sentence with some stopwords in it."

# Tokenize the text
words=word_tokenize(text)

# Get the list of English stopwords
stop_words=set(stopwords.words('english'))

# filter out the stopwords
filtered_words=[]
for w in words:
  if w not in stop_words:
    filtered_words.append(w)

print('Original Text:',text,'\n')
print('Filtered Text:',filtered_words)
```

```
Original Text: This is an example sentence with some stopwords in it.

Filtered Text: ['This', 'example', 'sentence', 'stopwords', '.']
```

```
# Length of stopwords of different language

print('Chinese:',len(stopwords.words('chinese')))
print('English:',len(stopwords.words('english')))
print('French:',len(stopwords.words('french')))
print('German:',len(stopwords.words('german')))
```

```
Chinese: 841
English: 179
French: 157
German: 232
```

## 5. Part of Speech (POS)

POS stands for Part of Speech. POS tagging (also known as grammatical tagging or word-category disambiguation) is the process of assigning each word in a text to a particular part of speech, such as nouns, verbs, adjectives, adverbs, etc., based on its definition and context within a sentence.

| Tag | Part of Speech | Description | Example |
|---|---|---|---|
| **NN** | Noun | Common noun | dog, city, car |
| **NNS** | Noun (Plural) | Plural noun | dogs, cities, cars |
| **NNP** | Proper Noun | Singular proper noun | John, India, Microsoft |
| **NNPS** | Proper Noun (Plural) | Plural proper noun | Americans, Australians |
| **VB** | Verb (Base Form) | Base form of a verb | run, eat, be |
| **VBD** | Verb (Past Tense) | Past tense of a verb | ran, ate, was |
| **VBG** | Verb (Gerund) | Verb ending in -ing | running, eating |
| **VBN** | Verb (Past Participle) | Past participle form | run, eaten, been |
| **VBP** | Verb (Present) | Non-3rd person singular present verb | run, eat |
| **VBZ** | Verb (3rd Person) | 3rd person singular present verb | runs, eats |
| **JJ** | Adjective | Describes a noun | big, fast, beautiful |
| **RB** | Adverb | Modifies a verb, adjective, or other adverb | quickly, very, well |
| **PRP** | Pronoun | Personal pronoun | he, she, it |
| **DT** | Determiner | Introduces a noun | the, a, an |
| **IN** | Preposition | Shows relationship between nouns | in, on, at |
| **CC** | Conjunction | Connects words, phrases, or clauses | and, but, or |
| **UH** | Interjection | Exclamatory word | wow, ouch, hey |

```
S='Arjun is a good rider when it comes to hourse riding'
S_t=word_tokenize(S)
S_t
```

```
['Arjun',
 'is',
 'a',
 'good',
 'rider',
 'when',
 'it',
 'comes',
 'to',
 'hourse',
 'riding']
```

```
for i in S_t:
  print(nltk.pos_tag([i]))
```

```
print("\n")
```

```
[('Arjun', 'NN')]
[('is', 'VBZ')]
[('a', 'DT')]
[('good', 'JJ')]
[('rider', 'NN')]
[('when', 'WRB')]
[('it', 'PRP')]
[('comes', 'VBZ')]
[('to', 'TO')]
[('hourse', 'NN')]
[('riding', 'VBG')]
```

## 6. Chunking

Chunking, also known as shallow parsing, is a Natural Language Processing (NLP) technique used to extract phrases (or "chunks") from a sentence. While POS tagging labels each word in a sentence with its part of speech, chunking goes a step further by grouping these tags into meaningful phrases, such as noun phrases (NP), verb phrases (VP), and prepositional phrases (PP).

Chunking helps in identifying and extracting sub-structures in sentences, making it useful for various NLP tasks like information extraction, named entity recognition, and syntactic analysis.

**How Chunking Works**

1. POS Tagging: The first step is to assign part-of-speech tags to each word in the sentence.
2. Chunk Patterns: Define patterns using regular expressions to specify which POS tags form a particular chunk.
3. Chunking: Apply these patterns to the tagged words to extract chunks.

**Two key concept in chunking**

1. Syntatic Analysis: It helps to divide the text with grammatic structure like noun phrase, verb phrase.
2. Information Extraction: Relevant information extraction from the text.

**Chunking in LLM**

- It is a process to processing of input text in smaller and manageble segments, these segments called chunks. This process helps to improve efficiency.
- The chunking in NLP done by manually where in LLM its done by transformers.

```
from nltk import ne_chunk

Ch_text='The US president stays in the WHITEHOUSE'

Ch_t=word_tokenize(Ch_text)
Ch_t
```

```
['The', 'US', 'president', 'stays', 'in', 'the', 'WHITEHOUSE']
```

```
Ch_tag=nltk.pos_tag(Ch_t)
Ch_tag
```

```
[('The', 'DT'),
 ('US', 'NNP'),
 ('president', 'NN'),
 ('stays', 'NNS'),
 ('in', 'IN'),
 ('the', 'DT'),
 ('WHITEHOUSE', 'NNP')]
```

```
Ch_ner=ne_chunk(Ch_tag)
print(Ch_ner)
```

```
(S
  The/DT
  (GSP US/NNP)
  president/NN
  stays/NNS
  in/IN
  the/DT
  (ORGANIZATION WHITEHOUSE/NNP))
```

## 7. Natural Entity Recognition (NER)

Its a NLP technique used to identify and classify entities in a given text into predefined categories. These entities typically represent real-word objects like names, locations, organizations, dates, and more.

```
import nltk
from nltk import word_tokenize, pos_tag,ne_chunk
from nltk.tree import Tree

text="Elon Musk is the CEO of SpaceX, which is headquartered in Hawthorne, California."

def extract_named_entities(text):
  #Tokenize the text
  words=word_tokenize(text)

  #Perform POS tagging
  words_pos=pos_tag(words)

  #Perform NER
  NER_entities_tree=ne_chunk(words_pos)

  #Extract named entities
  named_entities=[]
  for subtree in NER_entities_tree:
    if isinstance(subtree, Tree):
      entity_name=' '.join([token for token, pos in subtree.leaves()])
      entity_type=subtree.label()
      named_entities.append((entity_name, entity_type))
  return named_entities


# Extract and display named entities
entities=extract_named_entities(text)
for entity, entity_type in entities:
  print(f"{entity}: {entity_type}")
```

```
Elon: PERSON
Musk: ORGANIZATION
CEO of SpaceX: ORGANIZATION
Hawthorne: GPE
California: GPE
```

**Explanation:**

1. Tokenization:
    - Splits the text into individual words.
2. POS Tagging:
    - Assigns parts of speech (e.g., noun, verb) to each word.
3. Chunking:
    - Groups words into chunks and assigns labels to identify named entities.
4. Extracting Entities:
    - Extracts entities and their types from the parsed tree.

**Notes:**

- PERSON: A person's name.
- ORGANIZATION: A company or group.
- GPE: Geopolitical entity (e.g., cities, countries).

## 8. Natural Language Generation (Word Cloud)

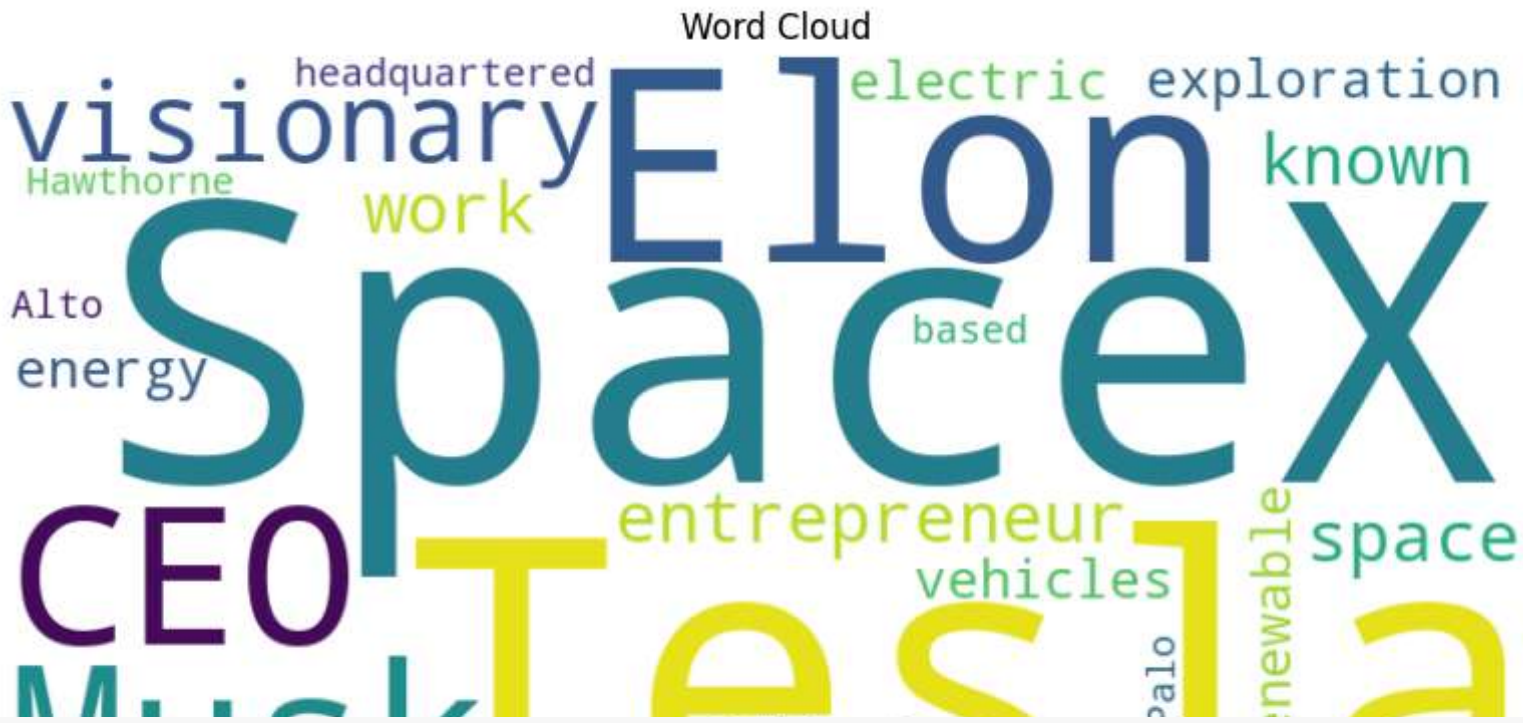
Visual representation of text data, where the size of each word indicates its frequency or importance within the text. Word clouds are often used in NLP for visualizing the most prominent words in a document or a set of documents. This helps in understanding the main themes or topics quickly.

```
from wordcloud import WordCloud
import matplotlib.pyplot as plt

# Sample Text
text = """
Elon Musk is the CEO of SpaceX and Tesla. He is a visionary entrepreneur known for his work on space exploration,
electric vehicles, and renewable energy. SpaceX is headquartered in Hawthorne, California, and Tesla is based in Palo Alto.
"""
```

```
#Create a WordCloud object
wordcloud=WordCloud(width=800, height=400, background_color='white', colormap='viridis').generate(text)

#Plot the wordcloud
plt.figure(figsize=(10,5))
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis('off')
plt.title('Word Cloud')
plt.show()
```

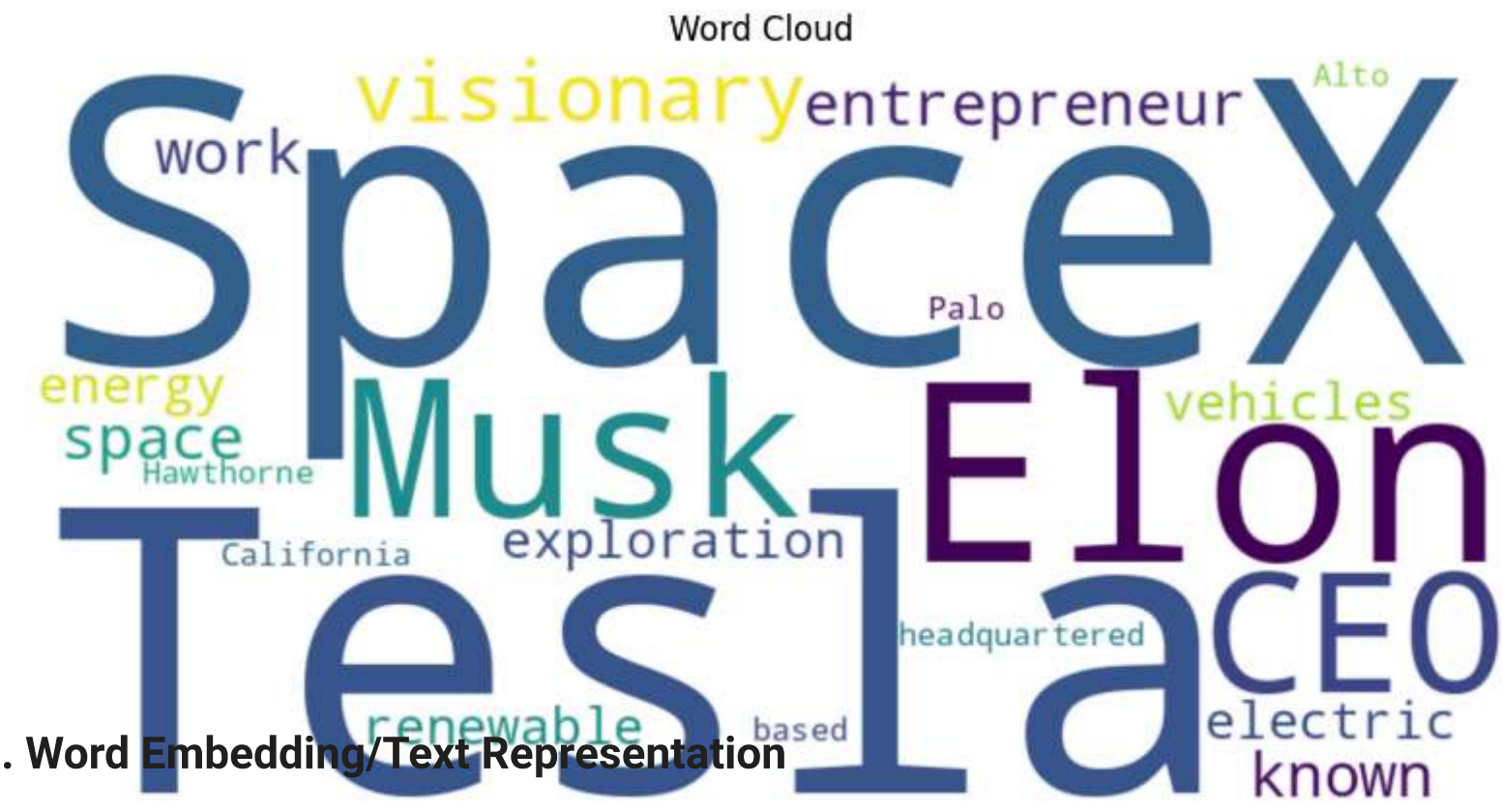



```
from wordcloud import WordCloud
import matplotlib.pyplot as plt

# Sample Text
text = """
Elon Musk is the CEO of SpaceX and Tesla. He is a visionary entrepreneur known for his work on space exploration,
electric vehicles, and renewable energy. SpaceX is headquartered in Hawthorne, California, and Tesla is based in Palo Alto.
"""

#Create a WordCloud object
wordcloud=WordCloud(width=800, height=400, background_color='white', colormap='viridis').generate(text)

#Plot the wordcloud
plt.figure(figsize=(10,5))
plt.imshow(wordcloud, interpolation='quadric')
plt.axis('off')
plt.title('Word Cloud')
plt.show()
```




## 9. Word Embedding/Text Representation

In Natural Language Processing (NLP), text representation is a crucial step where raw text is transformed into numerical formats so that algorithms can process and analyze it. This is a technique to represent words in a numeric form that models semantic relationships between them.

Three popular text representation techniques are:

1. Bag of Words (BoW)
2. Term Frequency-Inverse Document Frequency (TF-IDF)
3. Word2Vec

**Text Preprocessing**

It is required in most NLP tasks for cleaning and structuring raw text data for better analysis. These preprocessing steps are commonly used in NLP tasks such as text classification, sentiment analysis, and topic modeling to make the text data more usable by machine learning models.

- Improves Model Performances
- Standardization
- Reduces Dimensionality
- Ensures Meaningful Features
- Contextual Relevance

```
text="Natural Language Processing with Python is powerful. Python is popular for NLP tasks. Word clouds help in visualization."
text
```

```
'Natural Language Processing with Python is powerful. Python is popular for NLP tasks. Word clouds help in visualization.'
```

```
import nltk
import re
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
from nltk.stem import PorterStemmer
from nltk.tokenize import sent_tokenize


ps=PorterStemmer()
lemmatizer=WordNetLemmatizer()
sentences=nltk.sent_tokenize(text)


# Create the empty list name as corpus
corpus=[]

for i in range(len(sentences)):
  review=re.sub('[^a-zA-Z]',' ',sentences[i]) #Removes Non-Alphabetic Characters
  review=review.lower()  #Lowercasing
  review=review.split()  #Splitting into individual Words
  review=[lemmatizer.lemmatize(word) for word in review if word not in set(stopwords.words('english'))] #gets base from each word and stopwwo
  review=' '.join(review) #joining the words into a sentence
  corpus.append(review)


print(corpus)
```

```
['natural language processing python powerful', 'python popular nlp task', 'word cloud help visualization']
```

**1. Bag of Words(BoW)**

This model is one of the simplest text representation techniques. It treats text as a 'bag' of individual words, ignoring grammar and word order, but keeping track of word frequencies.

BoW is vectorization which converted text to 0 or 1

```
Document: ["I Love NLP","NLP is fun"]
Vocabulary: ["I","Love", "NLP", "is","fun"]

BoW Vectors:
["I","Love", "NLP", "is","fun"]
[1, 1, 1, 0, 0] ==> "I Love NLP"
[0, 0, 1, 1, 1] ==> "NLP is fun"
```

```
from sklearn.feature_extraction.text import CountVectorizer

# BoW vectorizer
vectorizer = CountVectorizer()
X = vectorizer.fit_transform(corpus)

# Display feature names and vector representation
print(vectorizer.get_feature_names_out())
print(X.toarray())
```

```
['cloud' 'help' 'language' 'natural' 'nlp' 'popular' 'powerful'
 'processing' 'python' 'task' 'visualization' 'word']
[[0 0 1 1 0 0 1 1 1 0 0 0]
```

```
[0 0 0 0 1 1 0 0 1 1 0 0]
[1 1 0 0 0 0 0 0 0 0 1 1]]
```

```
# Display the BoW matrix

import pandas as pd
df_bow=pd.DataFrame(X.toarray(), columns=vectorizer.get_feature_names_out())
print(df_bow)
```

```
   cloud  help  language  natural  nlp  popular  powerful  processing  python  \
0      0     0         1        1    0        0         1           1       1
1      0     0         0        0    1        1         0           0       1
2      1     1         0        0    0        0         0           0       0

   task  visualization  word
0     0              0     0
1     1              0     0
2     0              1     1
```

**2. TF-IDF (Term Frequency - Inverse Document Frequency)**

This is an extension of BoW of words model. It measures the importance of a word in a document relative to its frequency across the entire corpus. It helps to down-weight common words and highlight rare, important words.

**How It Works:**

- **Term Frequency (TF)**: Measures how often a word appears in a document.

$$\text{TF}(w) = \frac{\text{Number of times word } w \text{ appears in the document}}{\text{Total number of words in the document}}$$

- **Inverse Document Frequency (IDF)**: Measures how important a word is across the entire corpus.

$$\text{IDF}(w) = \log\left(\frac{\text{Total number of documents}}{\text{Number of documents containing word } w}\right)$$

- **TF-IDF Calculation**:

$$\text{TF-IDF}(w, D) = \text{TF}(w, D) \times \text{IDF}(w)$$

**Example:** Given three documents:

1. "I love machine learning"
2. "I love deep learning"
3. "Machine learning is fun"

We compute TF for each word and its IDF based on the corpus. For example, the word "love" appears in documents 1 and 2, so its IDF would be lower compared to "machine" or "deep," which are less frequent across the corpus.

```
from sklearn.feature_extraction.text import TfidfVectorizer

tf=TfidfVectorizer()
X_tfidf=tf.fit_transform(corpus).toarray()
print(X_tfidf)
```

```
[[0.         0.         0.46735098 0.46735098 0.         0.
  0.46735098 0.46735098 0.35543247 0.         0.         0.        ]
 [0.         0.         0.         0.         0.52863461 0.52863461
  0.         0.         0.40204024 0.52863461 0.         0.        ]
 [0.5        0.5        0.         0.         0.         0.
  0.         0.         0.         0.         0.5        0.5       ]]
```

```
# Display the TF-IDF matrix

import pandas as pd
df_tfidf=pd.DataFrame(X_tfidf,columns=tf.get_feature_names_out())
print(df_tfidf)
```

```
   cloud  help  language   natural       nlp   popular  powerful  processing  \
0    0.0   0.0  0.467351  0.467351  0.000000  0.000000  0.467351    0.467351
1    0.0   0.0  0.000000  0.000000  0.528635  0.528635  0.000000    0.000000
2    0.5   0.5  0.000000  0.000000  0.000000  0.000000  0.000000    0.000000
```

| | python | task | visualization | word |
|---|---|---|---|---|
| 0 | 0.355432 | 0.000000 | 0.0 | 0.0 |
| 1 | 0.402040 | 0.528635 | 0.0 | 0.0 |
| 2 | 0.000000 | 0.000000 | 0.5 | 0.5 |

**3. Word2Vec** Its a neural network-based algorithm that generates dense, continuous word vectors capturing the semantic meaning of words.

Two Main Approaches of Word2Vec:

Skip-gram Model: Predicts the context words (neighbors) for a given target word. Continuous Bag of Words (CBOW): Predicts the target word based on the context words (surrounding words).

```
from gensim.models import Word2Vec
from nltk.corpus import stopwords
import re
```

```
Str="""I have three visions for India. In 3000 years of our history, people from all over
               the world have come and invaded us, captured our lands, conquered our minds.
               From Alexander onwards, the Greeks, the Turks, the Moguls, the Portuguese, the British,
               the French, the Dutch, all of them came and looted us, took over what was ours.
               Yet we have not done this to any other nation. We have not conquered anyone.
               We have not grabbed their land, their culture,
               their history and tried to enforce our way of life on them.
               Why? Because we respect the freedom of others.That is why my
               first vision is that of freedom. I believe that India got its first vision of
               this in 1857, when we started the War of Independence. It is this freedom that
               we must protect and nurture and build on. If we are not free, no one will respect us.
               My second vision for India's development. For fifty years we have been a developing nation.
               It is time we see ourselves as a developed nation. We are among the top 5 nations of the world
               in terms of GDP. We have a 10 percent growth rate in most areas. Our poverty levels are falling.
               Our achievements are being globally recognised today. Yet we lack the self-confidence to
               see ourselves as a developed nation, self-reliant and self-assured. Isn't this incorrect?
               I have a third vision. India must stand up to the world. Because I believe that unless India
               stands up to the world, no one will respect us. Only strength respects strength. We must be
               strong not only as a military power but also as an economic power. Both must go hand-in-hand.
               My good fortune was to have worked with three great minds. Dr. Vikram Sarabhai of the Dept. of
               space, Professor Satish Dhawan, who succeeded him and Dr. Brahm Prakash, father of nuclear material.
               I was lucky to have worked with all three of them closely and consider this the great opportunity of my life.
               I see four milestones in my career"""

# text Preprocessing the data
text=re.sub(r'\[[0-9]*\]',' ',Str)
text=re.sub(r'\s+',' ',text)
text=text.lower()
text=re.sub(r'\d',' ',text)
text=re.sub(r'\s+',' ',text)

# preparing the dataset
sentences=nltk.sent_tokenize(text)
sentences = [nltk.word_tokenize(sentence) for sentence in sentences]
```

```
for i in range(len(sentences)):
  sentences[i]=[word for word in sentences[i] if word not in stopwords.words('english')]

# Training the Word2Vec model
model=Word2Vec(sentences,min_count=1)

# Get vocabulary using the updated method
words=list(model.wv.index_to_key)
print("\nVocabulary:", words)
```

```
Vocabulary: ['.', ',', 'india', 'vision', 'must', 'nation', 'world', 'us', 'three', 'freedom', 'respect', 'see', 'first', 'power', 'yet',
```

```
# Get vector for a specific word
vector = model.wv['freedom']
print("\nVector for 'freedom':\n", vector)
```

```
Vector for 'freedom':
 [-0.0051631  -0.0066453  -0.00776184  0.0083275  -0.00195818 -0.00692628
 -0.00412722  0.00519563 -0.00291064 -0.00380083  0.00159657 -0.002827
 -0.00159934  0.00109188 -0.00293189  0.00851129  0.0039173  -0.00997951
  0.00624404 -0.00680946  0.00080995  0.00440757 -0.00506883 -0.00214917
  0.00807772 -0.00424843 -0.00769278  0.00928459 -0.00216417 -0.00472591
  0.00860444  0.00427192  0.00435006  0.00925565 -0.00846837  0.00528312
  0.00204817  0.00421336  0.00171719  0.00446049  0.00453381  0.00609029
 -0.00319895 -0.00455255 -0.00042677  0.00255552 -0.0033018   0.00607402
  0.00416903  0.00780428  0.00263079  0.00808929 -0.00134183  0.0080396
```

```
 0.00372594 -0.00798969 -0.00391808 -0.00244056  0.00485149 -0.00087969
-0.00283986  0.00783192  0.00931493 -0.00159713 -0.00516179 -0.00465352
-0.00484487 -0.00957492  0.00137493 -0.00421809  0.00249247  0.00560655
-0.00401112 -0.00959241  0.00158418 -0.00670505  0.00250431 -0.00379514
 0.00709942  0.00063034  0.00353738 -0.00271629 -0.00176523  0.00768152
 0.00140953 -0.00584416 -0.00778533  0.00125514  0.0064905   0.005566
-0.00891621  0.00861838  0.00400781  0.00746545  0.00979657 -0.00725363
-0.00907492  0.00585616  0.00939498  0.00346861]
```

```
# Find similar words using the updated method
similar=model.wv.most_similar('freedom')
print("\nMost similar words to 'freedom':\n", similar)
```

```
Most similar words to 'freedom':
 [('unless', 0.2513527274131775), ('moguls', 0.2480589896440506), ('.', 0.2167753130197525), ('visions', 0.21363484859466553), ('developm
```

**Note:**

- When the Data is Small- We can go for Gensim Model
- When the Data is Huge- We can use RNN(Recurrent Neural Network)

**1. Continuous Bag of Words (CBOW)**

CBOW predicts a target word given a context of surrounding words. For instance, in the sentence "The cat sits on the mat," if you take the context words "The","cat", "on", "the," and "mat," CBOW would predict the target word "sits."

-Mechanism

- Input: Surrounding Words
- Output: The predicted target words
- The model averages the vectors of the context words to generate a prediction for the target word. Example: Given the context words ("The," "cat," "on," "the," "mat"), CBOW would try to predict the word "sits."

-Strengths:

- Works well with smaller datasets
- Tends to capture the global context of the sentence.

**2. Skip-gram**

Skip-gram does the opposite of CBOW; it uses a target word to predict its surrounding context words. For example, given the target word "sits," it would try to predict the context words around it.

-Mechanism

- Input: A target word.
- Output: The predicted context words.
- The model generates predictions for multiple context words based on the input target word. Example: Given the target word "sits," Skip-gram might predict the context words ("The," "cat," "on," "the," "mat").

-Strengths:

- More effective for larger datasets.
- Captures more information about the relationships between words, making it useful for understanding semantics.

**Conclusion:**

Both CBOW and Skip-gram are effective methods for learning word embeddings, and the choice between them often depends on the size of the training dataset and the specific application. CBOW tends to be faster and is suitable for smaller datasets, while Skip-gram excels in capturing relationships in larger datasets.

```
import gensim
from gensim.models import Word2Vec

# Sample sentences for training

sentences=[["I", "love", "natural", "language", "processing"],
    ["Word2Vec", "is", "a", "great", "tool"],
    ["Machine", "learning", "is", "fun"],
    ["Natural", "language", "processing", "is", "awesome"]]

# CBOW Model
cbow_model=Word2Vec(sentences,vector_size=100,window=2,min_count=1,sg=0)

# Skip-gram Model
skipgram_model=Word2Vec(sentences,vector_size=100,window=2,min_count=1,sg=1)

# Getting the vector for a word
word="language"
```

```
cbow_vector=cbow_model.wv[word]
skipgram_vector=skipgram_model.wv[word]

print("CBOW Vector for '{}':\n{}".format(word,cbow_vector))
print("\nSkip-gram Vector for '{}':\n{}".format(word,skipgram_vector))

# Finding similar words
print("\n")
cbow_similar_words=cbow_model.wv.most_similar(word,topn=5)
print(f"CBOW- Words similar to '{word}':",cbow_similar_words)
print("\n")
skipgram_similar_words=skipgram_model.wv.most_similar(word,topn=5)
print(f"Skip-gram- Words similar to '{word}':",skipgram_similar_words)
```

```
CBOW Vector for 'language':
[ 9.4794443e-05  3.0776660e-03 -6.8129268e-03 -1.3756783e-03
  7.6698321e-03  7.3483307e-03 -3.6729362e-03  2.6408839e-03
 -8.3165076e-03  6.2072724e-03 -4.6391813e-03 -3.1636052e-03
  9.3106655e-03  8.7376230e-04  7.4904198e-03 -6.0752141e-03
  5.1592872e-03  9.9243205e-03 -8.4574828e-03 -5.1340456e-03
 -7.0650815e-03 -4.8629697e-03 -3.7796097e-03 -8.5361497e-03
  7.9556443e-03 -4.8439130e-03  8.4241610e-03  5.2615325e-03
 -6.5502375e-03  3.9581223e-03  5.4700365e-03 -7.4268035e-03
 -7.4072029e-03 -2.4764745e-03 -8.6256117e-03 -1.5829162e-03
 -4.0474746e-04  3.3000517e-03  1.4428297e-03 -8.8208629e-04
 -5.5940356e-03  1.7293066e-03 -8.9629035e-04  6.7937491e-03
  3.9739395e-03  4.5298305e-03  1.4351519e-03 -2.7006667e-03
 -4.3665408e-03 -1.0332628e-03  1.4375091e-03 -2.6469158e-03
 -7.0722066e-03 -7.8058685e-03 -9.1226082e-03 -5.9341355e-03
 -1.8468037e-03 -4.3235817e-03 -6.4619821e-03 -3.7178723e-03
  4.2904112e-03 -3.7397402e-03  8.3768284e-03  1.5343785e-03
 -7.2409823e-03  9.4339680e-03  7.6326625e-03  5.4943082e-03
 -6.8490817e-03  5.8238246e-03  4.0079155e-03  5.1836823e-03
  4.2568049e-03  1.9397212e-03 -3.1705969e-03  8.3537176e-03
  9.6112443e-03  3.7936033e-03 -2.8369424e-03  6.7305832e-06
  1.2181988e-03 -8.4593873e-03 -8.2249697e-03 -2.3308117e-04
  1.2385092e-03 -5.7431920e-03 -4.7247363e-03 -7.3465765e-03
  8.3276192e-03  1.2043064e-04 -4.5089805e-03  5.7007410e-03
  9.1796070e-03 -4.1010864e-03  7.9633193e-03  5.3759255e-03
  5.8792117e-03  5.1329390e-04  8.2118409e-03 -7.0186048e-03]

Skip-gram Vector for 'language':
[ 9.42478000e-05  3.07589723e-03 -6.81467308e-03 -1.37446506e-03
  7.66968913e-03  7.34756188e-03 -3.67422565e-03  2.64107878e-03
 -8.31711013e-03  6.20709499e-03 -4.63969819e-03 -3.16430302e-03
  9.31042060e-03  8.73924117e-04  7.48968776e-03 -6.07334916e-03
  5.15946327e-03  9.92259663e-03 -8.45542643e-03 -5.13521628e-03
 -7.06540514e-03 -4.86227963e-03 -3.78140691e-03 -8.53707641e-03
  7.95734860e-03 -4.84467065e-03  8.42242502e-03  5.26282424e-03
 -6.55034464e-03  3.95740150e-03  5.47204912e-03 -7.42573338e-03
 -7.40648946e-03 -2.47416925e-03 -8.62730667e-03 -1.58222113e-03
 -4.05228348e-04  3.30146588e-03  1.44360668e-03 -8.81455548e-04
 -5.59283607e-03  1.72997033e-03 -8.96777317e-04  6.79324893e-03
  3.97348078e-03  4.53017978e-03  1.43477262e-03 -2.69988249e-03
 -4.36586002e-03 -1.03212311e-03  1.43821072e-03 -2.64558522e-03
 -7.07180146e-03 -7.80386291e-03 -9.12202150e-03 -5.93545521e-03
 -1.84791500e-03 -4.32417588e-03 -6.46110671e-03 -3.71831306e-03
  4.28991020e-03 -3.73849226e-03  8.37847777e-03  1.53453019e-03
 -7.24213943e-03  9.43322573e-03  7.63178896e-03  5.49173821e-03
 -6.84854668e-03  5.82335703e-03  4.00784146e-03  5.18454844e-03
  4.25614510e-03  1.93787215e-03 -3.17041040e-03  8.35264847e-03
  9.61161498e-03  3.79254599e-03 -2.83515733e-03  7.22470668e-06
  1.21920742e-03 -8.46034940e-03 -8.22578371e-03 -2.32077524e-04
  1.23913167e-03 -5.74427750e-03 -4.72703110e-03 -7.34633533e-03
  8.32915492e-03  1.21630226e-04 -4.51042084e-03  5.70245273e-03
  9.18024499e-03 -4.09950921e-03  7.96421152e-03  5.37427003e-03
  5.87765872e-03  5.15150023e-04  8.21374636e-03 -7.01800501e-03]


CBOW- Words similar to 'language': [('tool', 0.1991048902273178), ('Word2Vec', 0.1727149933576584), ('Natural', 0.170233353972435), ('l
```

## Comparison of BoW, TF-IDF, and Word2Vec

| Aspect | BoW | TF-IDF | Word2Vec |
|---|---|---|---|
| **Representation** | Sparse vector (frequency count) | Sparse vector (weighted frequency) | Dense vector (continuous) |
| **Context** | Ignores context and word order | Ignores context, but weights frequency | Captures context and semantic meaning |
| **Dimensionality** | High (based on vocabulary size) | High (based on vocabulary size) | Low-dimensional, fixed-length vectors |
| **Advantages** | Simple, easy to implement | Reduces importance of common words | Captures semantic relationships |
| **Disadvantages** | Sparse, high-dimensional, ignores meaning | Sparse, still ignores order/context | Requires large corpus, computationally expensive |

# 11. spaCy

It is one of the most popular NLP libraries in Python. Its designed for efficient and easy to use NLP tasks, making it an excellent choice for building production-ready applications. Unlike other NLP libraries like NLTK, which focus on research and prototyping, spaCy is optimized for real-world use cases, providing pre-trained models and fast processing capabilities.

```
# Loading the spaCy Model

import spacy
nlp = spacy.load("en_core_web_sm")
```

```
AI = "Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever

#Process text
doc=nlp(AI)
doc
```

```
Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s.
```

### 1. Tokenization

```
tokens=[]

for i in doc:
  tokens.append(i.text)

print(tokens)
```

```
['Lorem', 'Ipsum', 'is', 'simply', 'dummy', 'text', 'of', 'the', 'printing', 'and', 'typesetting', 'industry', '.', 'Lorem', 'Ipsum', 'ha
```

### 2. Part-of-Speech (POS) Tagging

```
token_pos=[]

for i in doc:
  print(f"Token: {i.text}, POS: {i.pos_}, Tag: {i.tag_}")

print(token_pos)
```

```
Token: Lorem, POS: PROPN, Tag: NNP
Token: Ipsum, POS: PROPN, Tag: NNP
Token: is, POS: AUX, Tag: VBZ
Token: simply, POS: ADV, Tag: RB
Token: dummy, POS: ADJ, Tag: JJ
Token: text, POS: NOUN, Tag: NN
Token: of, POS: ADP, Tag: IN
Token: the, POS: DET, Tag: DT
```

```
Token: printing, POS: NOUN, Tag: NN
Token: and, POS: CCONJ, Tag: CC
Token: typesetting, POS: VERB, Tag: VBG
Token: industry, POS: NOUN, Tag: NN
Token: ., POS: PUNCT, Tag: .
Token: Lorem, POS: PROPN, Tag: NNP
Token: Ipsum, POS: PROPN, Tag: NNP
Token: has, POS: AUX, Tag: VBZ
Token: been, POS: AUX, Tag: VBN
Token: the, POS: DET, Tag: DT
Token: industry, POS: NOUN, Tag: NN
Token: 's, POS: PART, Tag: POS
Token: standard, POS: ADJ, Tag: JJ
Token: dummy, POS: ADJ, Tag: JJ
Token: text, POS: NOUN, Tag: NN
Token: ever, POS: ADV, Tag: RB
Token: since, POS: SCONJ, Tag: IN
Token: the, POS: DET, Tag: DT
Token: 1500s, POS: NOUN, Tag: NNS
Token: ., POS: PUNCT, Tag: .
[]
```

```
print("Tokens-->","Lemma-->","POS-->","POS Tagging-->","Dependency-->","shape-->","IsAlpha-->","StopWords",)
for i in doc:
  print(i.text,"-->",i.lemma_,"-->",i.pos_,"-->",i.tag_,"-->",i.dep_,"-->",i.shape_,"-->",i.is_alpha,"-->",i.is_stop)
```

```
Tokens--> Lemma--> POS--> POS Tagging--> Dependency--> shape--> IsAlpha--> StopWords
Lorem --> Lorem --> PROPN --> NNP --> compound --> Xxxxx --> True --> False
Ipsum --> Ipsum --> PROPN --> NNP --> nsubj --> Xxxxx --> True --> False
is --> be --> AUX --> VBZ --> ROOT --> xx --> True --> True
simply --> simply --> ADV --> RB --> advmod --> xxxx --> True --> False
dummy --> dummy --> ADJ --> JJ --> amod --> xxxx --> True --> False
text --> text --> NOUN --> NN --> attr --> xxxx --> True --> False
of --> of --> ADP --> IN --> prep --> xx --> True --> True
the --> the --> DET --> DT --> det --> xxx --> True --> True
printing --> printing --> NOUN --> NN --> nmod --> xxxx --> True --> False
and --> and --> CCONJ --> CC --> cc --> xxx --> True --> True
typesetting --> typeset --> VERB --> VBG --> conj --> xxxx --> True --> False
industry --> industry --> NOUN --> NN --> pobj --> xxxx --> True --> False
. --> . --> PUNCT --> . --> punct --> . --> False --> False
Lorem --> Lorem --> PROPN --> NNP --> compound --> Xxxxx --> True --> False
Ipsum --> Ipsum --> PROPN --> NNP --> nsubj --> Xxxxx --> True --> False
has --> have --> AUX --> VBZ --> aux --> xxx --> True --> True
been --> be --> AUX --> VBN --> ROOT --> xxxx --> True --> True
the --> the --> DET --> DT --> det --> xxx --> True --> True
industry --> industry --> NOUN --> NN --> poss --> xxxx --> True --> False
's --> 's --> PART --> POS --> case --> 'x --> False --> True
standard --> standard --> ADJ --> JJ --> amod --> xxxx --> True --> False
dummy --> dummy --> ADJ --> JJ --> amod --> xxxx --> True --> False
text --> text --> NOUN --> NN --> attr --> xxxx --> True --> False
ever --> ever --> ADV --> RB --> advmod --> xxxx --> True --> True
since --> since --> SCONJ --> IN --> prep --> xxxx --> True --> True
the --> the --> DET --> DT --> det --> xxx --> True --> True
1500s --> 1500 --> NOUN --> NNS --> pobj --> ddddx --> False --> False
. --> . --> PUNCT --> . --> punct --> . --> False --> False
```

### 3. Named Entity Recognition (NER):

```
for entity in doc.ents:
  print(entity.text,"--->",entity.label_)
```

```
Lorem Ipsum ---> PERSON
Lorem Ipsum ---> PERSON
the 1500s ---> DATE
```

### 4. Word Vectors and Similarity:

supports word vectors (word embeddings) to measure similarity between words or documents.

```
# Load word vectors
word1 = nlp("Industry")
word2 = nlp("Standard")

# Check similarity
similarity = word1.similarity(word2)
print(f"Similarity between 'Industry' and 'Standard': {similarity}")
```

```
Similarity between 'Industry' and 'Standard': 0.6585106777519069
<ipython-input-42-28f11adacada>:6: UserWarning: [W007] The model you're using has no word vectors loaded, so the result of the Doc.simila
  similarity = word1.similarity(word2)
```

### 5. Word Frequency:

```
from string import punctuation
from spacy.lang.en.stop_words import STOP_WORDS

stopwords=list(STOP_WORDS)

word_freq={}

for word in doc:
  if word.text.lower() not in stopwords:
    if word.text.lower() not in punctuation:
      if word.text not in word_freq.keys():
        word_freq[word.text]=1
      else:
        word_freq[word.text]+=1

print(word_freq)
print(len(word_freq))
```

```
{'Lorem': 2, 'Ipsum': 2, 'simply': 1, 'dummy': 2, 'text': 2, 'printing': 1, 'typesetting': 1, 'industry': 2, 'standard': 1, '1500s': 1}
10
```

```
# To get normalized/weighted frequencies you should divide all freq with length of word count

for i in word_freq.keys():
  word_freq[i]=word_freq[i]/len(word_freq)

print(word_freq)
```

```
{'Lorem': 0.02, 'Ipsum': 0.02, 'simply': 0.01, 'dummy': 0.02, 'text': 0.02, 'printing': 0.01, 'typesetting': 0.01, 'industry': 0.02, 'sta
```

**Advantages of spaCy**

- Fast and Efficient: Optimized for production use.
- Pre-trained Models: Comes with ready-to-use models for multiple languages.
- Easy Integration: Can be easily integrated into data pipelines.
- Extensible: Supports custom models, pipelines, and extensions.

**Use Cases**

- Information Extraction: Extracting structured data from text (e.g., names, dates, entities).
- Sentiment Analysis: Analyzing the sentiment of text (positive, negative, neutral).
- Chatbots: Building conversational AI systems.
- Text Classification: Categorizing text into predefined categories.
- Document Similarity: Finding similarities between documents.

## 11. Gensim in NLP

Gensim is an open-source Python library widely used for topic modeling, document similarity, and word embedding in Natural Language Processing (NLP). It is efficient for processing large text corpora and performing tasks like topic modeling, similarity retrieval, and vectorization.

**Key Features**

- Efficient Algorithm
- Document Similarity
- Topic Modeling
- Word Embedding
- Corpus Streaming

**Popular Models:**

1. Word2Vec
2. FastText
3. Doc2Vec
4. Latent Dirichlet Allocation(LDA)

**When to use Gensim:**

- Large Scale Text Processing
- Topic Modelling
- Word Embeddings
- Document Similarity
- Out of Vocabulary Word Handling

**Advantages of Gensim:**

- Scalability
- Efficiency
- Comprehensive
- Pre-trained models

**1. Word2Vec:**

```
from gensim.models import Word2Vec

# Sample Corpus
sentences = [["i", "love", "machine", "learning"], ["deep", "learning", "is", "fun"]]

# Train Word2Vec model
model = Word2Vec(sentences, vector_size=100, window=5, min_count=1, workers=4)

# Get vector for a word
print(model.wv['love'])

# Find most similar words
print(model.wv.most_similar('learning'))
```

```
[-8.7274825e-03  2.1301615e-03 -8.7354420e-04 -9.3190884e-03
 -9.4281426e-03 -1.4107180e-03  4.4324086e-03  3.7040710e-03
 -6.4986930e-03 -6.8730675e-03 -4.9994122e-03 -2.2868442e-03
 -7.2502876e-03 -9.6033178e-03 -2.7436293e-03 -8.3628409e-03
 -6.0388758e-03 -5.6709289e-03 -2.3441375e-03 -1.7069972e-03
 -8.9569986e-03 -7.3519943e-04  8.1525063e-03  7.6904297e-03
 -7.2061159e-03 -3.6668312e-03  3.1185520e-03 -9.5707225e-03
  1.4764392e-03  6.5244664e-03  5.7464195e-03 -8.7630618e-03
 -4.5171441e-03 -8.1401607e-03  4.5956374e-05  9.2636338e-03
  5.9733056e-03  5.0673080e-03  5.0610625e-03 -3.2429171e-03
  9.5521836e-03 -7.3564244e-03 -7.2703874e-03 -2.2653891e-03
 -7.7856064e-04 -3.2161034e-03 -5.9258583e-04  7.4888230e-03
 -6.9751858e-04 -1.6249407e-03  2.7443992e-03 -8.3591007e-03
  7.8558037e-03  8.5361041e-03 -9.5840869e-03  2.4462664e-03
  9.9049713e-03 -7.6658037e-03 -6.9669187e-03 -7.7365171e-03
  8.3959233e-03 -6.8133592e-04  9.1444086e-03 -8.1582209e-03
  3.7430846e-03  2.6350426e-03  7.4271322e-04  2.3276759e-03
 -7.4690939e-03 -9.3583735e-03  2.3545765e-03  6.1484552e-03
  7.9856887e-03  5.7358947e-03 -7.7733636e-04  8.3061643e-03
 -9.3363142e-03  3.4061326e-03  2.6675343e-04  3.8572443e-03
  7.3857834e-03 -6.7251669e-03  5.5844807e-03 -9.5222248e-03
 -8.0445886e-04 -8.6887367e-03 -5.0986730e-03  9.2892265e-03
 -1.8582619e-03  2.9144264e-03  9.0712793e-03  8.9381328e-03
 -8.2084350e-03 -3.0123137e-03  9.8866057e-03  5.1044310e-03
 -1.5880871e-03 -8.6920215e-03  2.9615164e-03 -6.6758976e-03]
[('i', 0.016134681180119514), ('fun', -0.01083916611969471), ('machine', -0.02775035798549652), ('is', -0.05234673246741295), ('love', -0
```

**2. FastText**

It improves upon Word2Vec by representing each word as a bag of charcter n-gram. This allows it to handle out-of-vocabulary words better and learn more robust embeddings for rare or misspelled words.

```
from gensim.models import FastText

# Sample corpus
sentences = [["i", "love", "machine", "learning"], ["deep", "learning", "is", "fun"]]

# Train FastText model
model = FastText(sentences, vector_size=100, window=5, min_count=1, workers=4)

# Get vector for a word
print(model.wv['learning'])

# Find most similar words
print(model.wv.most_similar('learning'))
```

```
[ 1.08934077e-03 -5.32758095e-05 -9.47115652e-04  3.95029463e-04
 -1.97012443e-03 -4.83391195e-04 -5.70225879e-04 -6.75312069e-04
  1.33851462e-03  3.70645837e-04  9.34705313e-05  3.36847617e-04
 -1.10260618e-03  1.28237996e-03  4.40919917e-04 -1.53581623e-03
 -1.58885110e-03 -1.55063241e-03  3.61978346e-05 -2.19325186e-03
 -1.65212096e-03 -2.02007266e-03  1.52149866e-03 -1.18939875e-04
 -5.72251854e-04 -1.01570692e-03  1.78208298e-04 -9.08576476e-04
 -5.18821296e-04 -1.44369295e-03 -1.99734210e-03  1.61605922e-03
  9.45105101e-04 -1.97916073e-04  6.17624319e-05  9.92861576e-04
 -1.84913821e-04  2.15986060e-04 -1.39385066e-03 -1.23050751e-03
 -1.61930860e-03 -1.08920340e-03 -9.83674545e-04  6.95403141e-04
 -1.08810619e-03  2.05037446e-04 -2.50786048e-04  5.64446556e-04
```

```
 2.37473476e-04  7.82442861e-04  1.17882655e-03 -6.18749182e-04
-4.95790271e-04  1.66726543e-03  1.17137725e-03  9.29269416e-04
 6.02241125e-05  1.33268841e-04 -3.77347838e-04 -6.18015591e-04
 3.67764966e-04 -9.58980527e-04 -2.05716537e-03  1.23189075e-03
-1.39238837e-03 -7.69924314e-04  1.46265421e-03  8.81363114e-04
-4.78719565e-04 -3.46080837e-04  1.34837476e-03  4.69837280e-04
-9.80194542e-04 -1.38727110e-03 -2.19671041e-04 -4.38916468e-04
 1.83601445e-03  1.54251768e-03 -1.50243100e-03  8.23975308e-04
 1.99425593e-03  1.65304821e-03 -1.28263084e-03  1.91654661e-04
-1.21943001e-03  9.34042793e-04  1.50419527e-03 -6.19662867e-04
-6.40665705e-04  8.05283722e-04  1.35035533e-03  1.78457689e-04
 4.13762027e-04  3.14794888e-04  1.38592767e-03  1.45841751e-03
-5.72762860e-04  1.43580209e-03 -8.94677942e-04 -8.32054066e-04]
[('i', 0.017271071672439575), ('machine', -0.04402731731534004), ('is', -0.05588227137923241), ('fun', -0.08146974444389343), ('love', -0
```

**3. Doc2Vec** It is an extension of Word2Vec, but instead of learning word vectors, it learns vector representations for entire documents (or sentences). This is useful for tasks like document classification or document similarity.

```
from gensim.models import Doc2Vec
from gensim.models.doc2vec import TaggedDocument

# Sample corpus
documents = [TaggedDocument(words=["i", "love", "machine", "learning"], tags=["doc1"]),
             TaggedDocument(words=["deep", "learning", "is", "fun"], tags=["doc2"])]


# Train Doc2Vec model
model = Doc2Vec(documents, vector_size=100, window=5, min_count=1, workers=4)

# Get vector for a document
print(model.infer_vector(["machine", "learning"]))


# Find most similar documents
print(model.dv.most_similar('doc1'))
```

```
[ 3.3370054e-03  2.8402710e-03  4.1823182e-03  2.1049457e-03
  3.2470166e-03  2.9869764e-03  3.1769818e-03  1.2926805e-03
  1.8551910e-03 -4.0242216e-03  2.7256364e-03  2.4811071e-03
  8.4154727e-04  4.4918465e-03  3.6994116e-03 -2.8669066e-03
 -1.0254434e-03  4.6511830e-04  3.2875370e-03 -3.8270256e-03
  2.1618903e-04  4.7305776e-03  3.4854042e-03 -8.9604082e-04
  9.4270945e-04  3.7043530e-03  2.2603190e-03  4.4636875e-03
 -3.0255395e-03  7.5205386e-04  3.8147569e-03 -3.9616362e-03
  3.9851735e-03 -4.0818364e-04 -4.5556184e-03 -3.7670445e-03
  4.1214944e-04  2.5072491e-03  1.7392755e-04  2.1211147e-04
  1.1389733e-03 -8.7355258e-04 -4.8183547e-03  4.9089669e-04
 -3.1112444e-03  1.3741530e-03  2.7589530e-03  2.9194413e-04
 -2.6040627e-03 -4.2657610e-03 -3.6960682e-03  3.5613240e-04
 -3.0321599e-04 -1.5737286e-03  1.5244239e-03  2.7355016e-04
 -2.2260577e-04 -1.7401662e-03 -1.0108814e-03  3.8488461e-03
  3.4079670e-03  4.1966019e-03 -2.6735323e-03 -3.8674073e-03
 -2.4134410e-03  1.8342900e-03  3.8551111e-03  1.6476047e-03
  7.0666196e-04 -4.7972249e-03 -1.1707101e-03 -4.5369016e-03
  3.8601833e-03 -4.2042751e-03  2.0856215e-03  4.3359487e-03
 -4.1519450e-03 -7.8722536e-05  6.9830957e-04  3.9584260e-03
 -2.8490464e-03 -4.8608668e-03 -4.2147557e-03  3.1130391e-03
 -4.3163071e-03  5.7436823e-04 -1.4160520e-03 -4.4273697e-03
  2.4965697e-03  4.7113285e-03  1.5406454e-03 -2.7651023e-03
 -1.3581046e-03  3.2849717e-03  4.4084177e-03 -3.1067508e-03
 -6.4175576e-04  4.8202942e-03 -1.8224739e-03  2.0711338e-03]
[('doc2', 0.16280031204223633)]
```

**4. Latent Dirichlet Allocation (LDA):**

It is a topic modeling technique that is used to extract topics from a collection of documents. Gensim makes it easy to apply LDA to a corpus of text.

```
from gensim import corpora
from gensim.models import LdaModel

# Sample corpus
documents = ["I love machine learning", "Deep learning is fun", "I love artificial intelligence"]
texts=[[word for word in doc.split()] for doc in documents]

texts
```

```
[['I', 'love', 'machine', 'learning'],
 ['Deep', 'learning', 'is', 'fun'],
 ['I', 'love', 'artificial', 'intelligence']]
```

```
# Create a dictionary and corpus
dictionary = corpora.Dictionary(texts)
corpus = [dictionary.doc2bow(text) for text in texts]

# Train LDA model
lda_model = LdaModel(corpus, num_topics=2, id2word=dictionary, passes=10)

# Get topics
topics = lda_model.print_topics(num_words=3)
print(topics)
```

```
[(0, '0.180*"learning" + 0.174*"is" + 0.174*"fun"'), (1, '0.200*"I" + 0.200*"love" + 0.120*"artificial"')]
```

## 12. Transformers

Transformers have revolutionized Natural Language Processing (NLP) due to their parallelization capabilities, contextual understanding, and ability to handle long-range dependencies in text. These models leverage attention mechanisms to process input sequences in parallel, unlike previous models like RNNs and LSTMs, which process sequential data. The transformer architecture was introduced in the paper "Attention Is All You Need" by Vaswani et al. (2017), and since then, many powerful transformer-based models have been developed, including BERT, GPT, T5, RoBERTa, XLNet, and ALBERT. Below is a detailed look at each of these models:

**1. BERT(Bidirectional Encoder Representation from Transformers)**

- Released by: Google AI
- Model Type: Encoder-based
- Key Feature:
  - Bidirectional attention: It captures the context from both directions for better understanding.
  - Pre-training and Fine-Tuning.
  - Masked Language Model(MLM)
  - Next Sentence Prediction(NSP)
- Use Cases:
  - Question Answering
  - Named Entity Recognition
  - Text Classification
- Limitations:
  - Requires a large amount of computational resources, especially for pre-training.
  - BERT is an encoder-only model, meaning it is better suited for tasks like classification, NER, and sentence-level tasks rather than text generation.

**2. GPT (Generative Pre-trained Transformer)**

- Released by: OpenAI
- Model Type: Decoder-based
- Key Feature:
  - BAutoregressive Model: GPT is a decoder-only model that generates text word by word, conditioning on the previously generated words. It is typically used for text generation tasks.
  - Pre-training and Fine-tuning: GPT is first pre-trained on a large corpus of text data using a language modeling objective (predicting the next word in a sequence), then fine-tuned for specific tasks.
  - Transformer Decoder: GPT only uses the decoder part of the transformer architecture, unlike BERT, which uses the encoder.
  - Unidirectional attention: GPT processes text from left to right (unidirectional), which means it predicts the next word based on previous words, making it effective for generation tasks.
- Use Cases:
  - Text Generation
  - Language Modeling
  - Summarization
- Limitations:
  - GPT models can sometimes generate incoherent or irrelevant text, especially when they are not fine-tuned properly.
  - They are autoregressive, meaning they only generate text one token at a time and can be slow during inference.

**3. T5 (Text-to-Text Transfer Transformer)**

- Released by: Google Research
- Model Type: Encoder-decoder-based

- Key Feature:
    - Text-to-Text Framework: T5 treats every NLP task as a text-to-text problem. For example, it can translate a sentence, summarize a document, or classify text, all by feeding text inputs and generating text outputs.
    - Unified Model: T5 can perform a variety of NLP tasks by converting each task into a text generation problem (e.g., converting a classification task into a text label generation task).
    - Pre-training and Fine-tuning: T5 is pre-trained using a denoising objective, where parts of the input text are masked, and the model is trained to predict the missing parts.
    - Encoder-decoder Architecture: T5 uses both the encoder and decoder parts of the transformer to process the input and generate output.
- Use Cases:
    - Text Generation
    - Text Summarization
    - Text Translation
    - Question Answering
- Limitations:
    - T5 can be resource-intensive due to its encoder-decoder architecture.
    - Generating long sequences can be computationally expensive.

**4. RoBERTa (Robustly Optimized BERT Pretraining Approach)**

- Released by: Facebook AI
- Model Type: Encoder-based (like BERT)
- Key Feature:
    - Improved BERT: RoBERTa is a variant of BERT that improves its pretraining process by removing the Next Sentence Prediction (NSP) task and training on more data for longer periods.
    - Larger Batch Sizes: RoBERTa uses larger batch sizes and more training data compared to BERT.
    - Dynamic Masking: Unlike BERT's static masking strategy, RoBERTa uses dynamic masking, meaning the words selected for masking change every time a sentence is processed.
    - No NSP Task: RoBERTa does not use the Next Sentence Prediction task that was part of BERT's training process.
- Use Cases:
    - Text Classification
    - Named Entity Recognition
    - Question Answering
- Limitations:
    - RoBERTa is very computationally expensive, requiring powerful hardware for training.
    - Although it improves on BERT in several ways, it still shares some of the same limitations (e.g., lack of autoregressive generation capabilities).

**5. XLNet (Generalized Autoregressive Pretraining for Language Understanding)**

- Released by: Google AI and Carnegie Mellon University
- Model Type: Autoregressive (like GPT) and Autoencoder (like BERT)
- Key Feature:
    - Permutation-based Training: Unlike GPT (autoregressive) and BERT (autoencoding), XLNet uses a permutation-based training method, capturing both left-to-right and right-to-left context in training.
    - Better Long-range Dependencies: XLNet can capture long-range dependencies better than traditional autoregressive models like GPT and autoencoders like BERT.
    - Transformer Architecture: XLNet uses both the encoder and decoder parts of the transformer model, allowing it to model complex relationships between tokens.
- Use Cases:
    - Text Classification
    - Text Summarization
    - Question Answering
- Limitations:
    - XLNet can be more computationally expensive than models like BERT.
    - Its permutation-based training approach is more complex to implement compared to simple autoregressive models like GPT.

**6. ALBERT (A Lite BERT)**

- Released by: Google Research and Toyota Technological Institute
- Model Type: Encoder-based (like BERT)
- Key Feature:
    - Parameter Reduction: ALBERT reduces the number of parameters in BERT by sharing parameters across layers and factorizing the embedding matrix. This leads to a model that is lighter (smaller in size) and faster to train.
    - Sentence Order Prediction (SOP): ALBERT replaces the NSP task with a new objective, the Sentence Order Prediction task, which is designed to improve the model's understanding of sentence order and semantic coherence.
    - Lightweight: ALBERT is more efficient than BERT, providing a good trade-off between performance and resource requirements.
- Use Cases:
    - Text Classification
    - Named Entity Recognition
    - Question Answering