Natural Language Processing (NLP)

Harsh Choudhary

Natural Language Processing (NLP)

Natural Language Processing (NLP) is a branch of Artificial Intelligence (AI) that helps machines to understand and process human languages either in text or audio form. It is used across a variety of applications from speech recognition to language translation and text summarization.

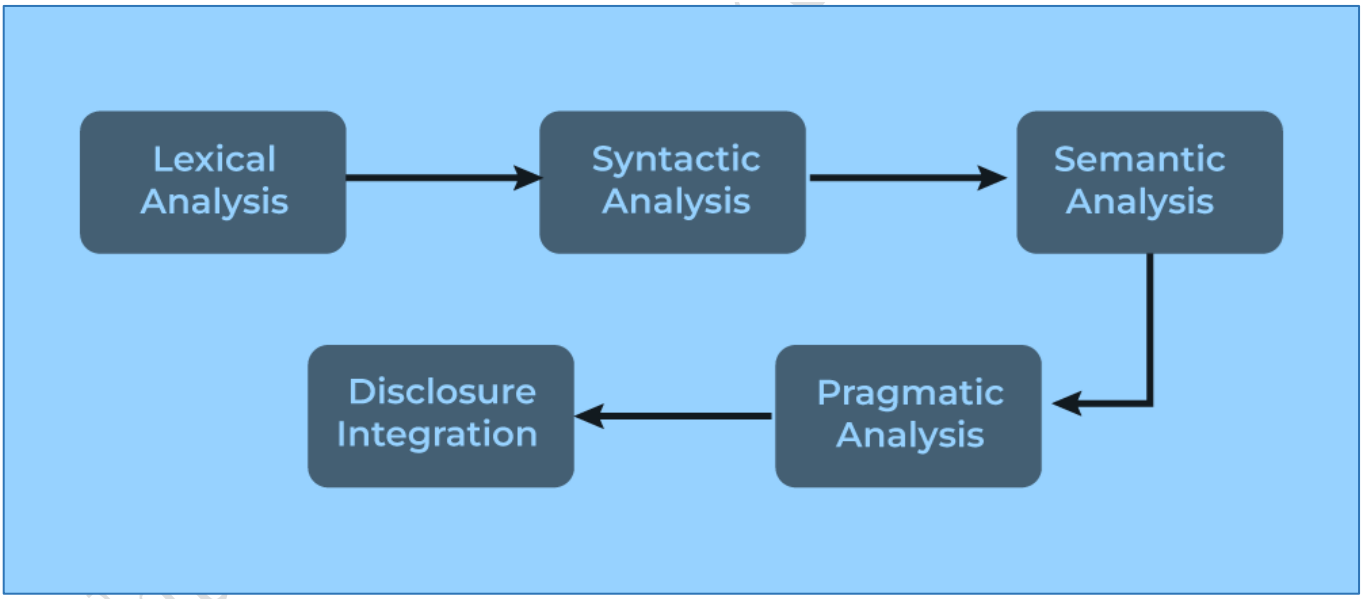
Natural Language Processing can be categorized into two components:

1. Natural Language Understanding: It involves interpreting the meaning of the text.
2. Natural Language Generation: It involves generating human-like text based on processed data.

- Natural Language Understanding
- Natural Language Generation

Phases of Natural Language Processing

It involves a series of phases that work together to process and interpret language with each phase contributing to understanding its structure and meaning.



Libraries for NLP

Some of natural language processing libraries include:

- NLTK (Natural Language Toolkit)
- spaCy
- TextBlob
- Transformers (by Hugging Face)
- Gensim
- NLP Libraries in Python.

NLTK - NLP

Natural Language Processing (NLP) plays an important role in enabling machines to understand and generate human language. Natural Language Toolkit (NLTK) stands out as one of the most widely used libraries. It provides a combination linguistic resources, including text processing libraries and pre-trained models, which makes it ideal for both academic research and practical applications.

NLTK is a Python's API library and it can perform a variety of operations on textual data such as classification, tokenization, stemming, tagging, semantic reasoning, etc.

Installation

NLTK can be installed simply using pip or by running the following code.

! pip install nltk

Accessing Additional Resources: For the usage of additional resources such as recourses of languages other than English - we can run the following in a python script. It has to be done only once when you are running it for the first time in your system.

import nltk

nltk.download('all')

Now, having installed NLTK successfully in our system we can perform some basic operations on text data using NLTK.

1. Tokenization

Tokenization refers to break down the text into smaller units. It splits paragraphs into sentences and sentences into words. It is one of the initial steps of any NLP pipeline. Let us have a look at the two major kinds of tokenization that NLTK provides:

1.1 Word Tokenization

It involves breaking down the text into words.

"I study Machine Learning on tech." will be word-tokenized as:

['I', 'study', 'Machine', 'Learning', 'on', 'tech, '.'].

1.2 Sentence Tokenization

Natural Language Processing (NLP)

It involves breaking down the text into individual sentences.

"I study Machine Learning on tech. Currently, I'm studying NLP" will be sentence-tokenized as :

['I study Machine Learning on tech.', 'Currently, I'm studying NLP.']

In Python, both these tokenizations can be implemented in NLTK as follows:

```
# Tokenization using NLTK
from nltk import word_tokenize, sent_tokenize
sent = "tech is a great learning platform.\
It is one of the best for Computer Science students."
print(word_tokenize(sent))
print(sent_tokenize(sent))
```

2. Stemming and Lemmatization

When working with natural language, our focus is on understanding the intended meaning behind words. To achieve this, it is essential to reduce words to their root or base form. This process is known as **canonicalization**.

For example, words like "play", "plays", "played" and "playing" all refer to the same action and can therefore be mapped to the common base form "play."

There are two commonly used techniques for canonicalization: **stemming** and **lemmatization**.

2.1 Stemming

Stemming generates the base word from the given word by removing the affixes of the word. It has a set of pre-defined rules that guide the dropping of these affixes. It must be noted that stemmers might not always result in semantically meaningful base words. Stemmers are faster and computationally less expensive than lemmatizers.

In the following code, we will be stemming words using Porter Stemmer:

```
from nltk.stem import PorterStemmer

# create an object of class PorterStemmer
porter = PorterStemmer()
print(porter.stem("play"))
```

Natural Language Processing (NLP)

```
print(porter.stem("playing"))

print(porter.stem("plays"))

print(porter.stem("played"))
```

Output:

play
play
play
play

We can see that all the variations of the word 'play' have been reduced to the same word, 'play'. In this case, the output is a meaningful word 'play'. However, this is not always the case.

Let us take an example:

```
from nltk.stem import PorterStemmer

# create an object of class PorterStemmer

porter = PorterStemmer()

print(porter.stem("Communication"))
```

Output:

Commun

The stemmer reduces the word 'communication' to a base word 'commun' which is meaningless in itself.

2.2 Lemmatization

Lemmatization involves grouping together the inflected forms of the same word. This way, we can reach out to the base form of any word which will be meaningful in nature. The base from here is called the Lemma.

Lemmatizers are slower and computationally more expensive than stemmers.

Example: *'play', 'plays', 'played', and 'playing' have 'play' as the lemma.*

In Python, both these tokenizations can be implemented in NLTK as follows:

```
from nltk.stem import WordNetLemmatizer

# create an object of class WordNetLemmatizer

lemmatizer = WordNetLemmatizer()

print(lemmatizer.lemmatize("plays", 'v'))
```

Natural Language Processing (NLP)

```
print(lemmatizer.lemmatize("played", 'v'))

print(lemmatizer.lemmatize("play", 'v'))

print(lemmatizer.lemmatize("playing", 'v'))
```

Output:

play
play
play
play

- Note that in lemmatizers, we need to pass the Part of Speech of the word along with the word as a function argument.
- Also, lemmatizers always result in meaningful base words.

Let us take the same example as we took in the case for stemmers.

```
# create an object of class WordNetLemmatizer

lemmatizer = WordNetLemmatizer()

print(lemmatizer.lemmatize("Communication", 'v'))
```

Output:

Communication

3. Part of Speech Tagging

Part of Speech (POS) tagging refers to assigning each word of a sentence to its part of speech. It is significant as it helps to give a better syntactic overview of a sentence.

Example

Let's see how NLTK's POS tagger will tag this sentence.

```
from nltk import pos_tag

from nltk import word_tokenize

tokenized_text = word_tokenize(text)

tags = tokens_tag = pos_tag(tokenized_text)

tags
```

Output:

POS output

4. Named Entity Recognition (NER)

Natural Language Processing (NLP)

Named Entity Recognition (NER) is another important task in Natural Language Processing (NLP) and NLTK provides built-in capabilities to perform it. NER involves identifying and classifying key information in a text such as names of people, places , organizations and more. It's an important step for information extraction and understanding the meaning of text at a deeper level.

Example: "Barack Obama was born in Hawaii in 1961."
Let's see how NLTK's NER module identifies entities in this sentence.

```
from nltk import word_tokenize, pos_tag, ne_chunk


# Download the required resource for NER

nltk.download('maxent_ne_chunker_tab')

nltk.download('words') # This resource is also needed for the chunker


# Sample text

text = "Barack Obama was born in Hawaii in 1961."


# Tokenize and POS tag the sentence

tokens = word_tokenize(text)

tags = pos_tag(tokens)


# Apply Named Entity Recognition

entities = ne_chunk(tags)

print(entities)
```

Output:

Natural Language Processing (NLP)

```
[nltk_data] Downloading package maxent_ne_chunker_tab to
[nltk_data]     /root/nltk_data...
[nltk_data]   Unzipping chunkers/maxent_ne_chunker_tab.zip.
[nltk_data] Downloading package words to /root/nltk_data...
[nltk_data]   Unzipping corpora/words.zip.
(S
  (PERSON Barack/NNP)
  (PERSON Obama/NNP)
  was/VBD
  born/VBN
  in/IN
  (GPE Hawaii/NNP)
  in/IN
  1961/CD
  ./.)
```

POS using NER

In conclusion, the Natural Language Toolkit (NLTK) works as a powerful Python library that has a wide range of tools for Natural Language Processing (NLP). From fundamental tasks like text pre-processing to more advanced operations such as semantic reasoning, NLTK provides an API that aids to all these needs of language-related tasks.

Tokenization Using Spacy

Tokenization is one of the first steps in Natural Language Processing (NLP) where we break down text into smaller units or "tokens." These tokens can be words, punctuation marks or special characters making it easier for algorithms to process and analyze the text. SpaCy is one of the most widely used libraries for NLP tasks which provides an efficient way to perform tokenization.

Let's consider a sentence: "I love natural language processing!"

After tokenization: ["I", "love", "natural", "language", "processing", "!"]

This breakdown helps us understand the individual components of the text, making it easier for algorithms to process. Tokenization is important for further tasks like text classification, sentiment analysis and more.

Key Features of SpaCy Tokenizer

1. **Efficient Tokenization**: SpaCy's tokenizer is built for speed and efficiency, capable of handling large volumes of text quickly without compromising accuracy.

Natural Language Processing

2. **Handles Punctuation**: Unlike many basic tokenizers, it treats punctuation marks as separate tokens which is important for tasks like sentence segmentation and parsing.
3. **Language-Specific Tokenization**: It's tokenization is adjusted for different languages, taking into account language-specific rules like contractions, abbreviations and compound words.
4. **Whitespace and Special Character Handling**: The tokenizer properly manages spaces, newlines and special characters, ensuring that tokens like URLs, hashtags and email addresses are correctly identified.
5. **Customizable**: It allows us to customize the tokenizer by adding special rules for tokenization, giving us more control over how the text is split into tokens.

Implementation of Tokenization Using SpaCy

Here, we'll see how to implement tokenization using SpaCy.

1. Blank Model Tokenization

Here, we are using SpaCy's blank model (**spacy.blank("en")**) which initializes a minimal pipeline without pre-trained components like part-of-speech tagging or named entity recognition. This example shows a basic tokenization functionality.

```
import spacy

nlp = spacy.blank("en")


doc = nlp("techis a one stop\
learning destination for tech")


for token in doc:
    print(token)
```

Output:

Tokens

2. Displaying the Pipeline Components

We use the pre-trained **en_core_web_sm** model which includes various components for NLP tasks. After loading the model, we can display the available components in the pipeline.

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

Natural Language Processing (NLP)

nlp.pipe_names

Output:

['tok2vec', 'tagger', 'parser', 'attribute_ruler', 'lemmatizer', 'ner']

3. Tokenization with Part-of-Speech Tagging and Lemmatization

Here we see how to process text, extract token information like part-of-speech (POS) tagging and obtain the lemmatized form of each token.

```
import spacy

nlp = spacy.load("en_core_web_sm")


doc = nlp("If you want to be an excellent programmer \

, be consistent to practice daily on GFG.")


for token in doc:
    print(token, " | ",
        spacy.explain(token.pos_),
        " | ", token.lemma_)
```

The **spacy.explain()** function provides a description of the POS tag for each token. The model performs tokenization, POS tagging and lemmatization automatically when we process the text with NLP.

Advantages of Using SpaCy for Tokenization

1. **Efficiency:** It is designed for fast processing, enabling quick tokenization even for large volumes of text which is useful in real-time or large-scale applications.
2. **High Accuracy:** Its pre-trained models like en_core_web_sm offer reliable and accurate tokenization along with other NLP tasks like part-of-speech tagging and named entity recognition.
3. **Ease of Use:** The intuitive and simple API makes SpaCy easy to use for tokenization, reducing the complexity of setting up tokenization tasks for both beginners and experts.
4. **Extensibility:** It allows easy integration with other NLP tasks such as lemmatization, dependency parsing and named entity recognition (NER) making it a useful library for building custom NLP pipelines.

Natural Language Processing (NLP)

Limitations of Using SpaCy for Tokenization

1. **Memory Consumption:** SpaCy's pre-trained models can be large and require a significant amount of memory which may not be ideal for memory-constrained environments.
2. **Dependency on Pre-trained Models:** While it provides great pre-trained models, these models might not perform as well on specialized or domain-specific text without additional fine-tuning.
3. **Limited Support for Some Languages:** While it supports many languages, its models may not be as robust or comprehensive for certain languages especially those with limited linguistic resources.

TextBlob.sentiment() method

Sentiment analysis helps us find the emotional tone of text whether it's positive, negative or neutral. The **TextBlob.sentiment()** method simplifies this task by providing two key components:

- **polarity:** which shows if the text is positive, negative or neutral
- **subjectivity:** which shows whether the text is more opinion-based or factual

Let's see a basic example:

```
from textblob import TextBlob


text = "GFG is a good company and always value their employees."

blob = TextBlob(text)

sentiment = blob.sentiment

print(sentiment)
```

Output:

Sentiment(polarity=0.7, subjectivity=0.6000000000000001)

- **Polarity:** 0.7 means positive sentiment.
- **Subjectivity:** 0.6 means it's more opinion-based.

Syntax:

TextBlob.sentiment

Return:

Natural Language Processing (NLP)

- **Polarity:** A score between -1 (negative) and 1 (positive) showing how positive or negative the text is.
- **Subjectivity:** A score between 0 (factual) and 1 (opinion-based) showing how subjective or objective the text is.

Lets see some more examples:

Example 1: Negative Sentiment

Here, we analyze a sentence that expresses a strong negative sentiment. The polarity score reflects the negative tone while the subjectivity shows opinion-based statement.

text = "I hate bugs in my code."

blob = TextBlob(text)

sentiment = blob.sentiment

print(sentiment)

Output:

Sentiment(polarity=-0.8, subjectivity=0.9)

- **Polarity:** -0.8 shows a negative sentiment.
- **Subjectivity:** 0.9 shows the text is highly opinion-based.

Example 2: Neutral Sentiment

In this example, we'll analyze a neutral sentence that conveys factual information without expressing any strong opinion or emotion. It will show how TextBlob classifies a sentence with no sentiment bias.

text = "The sun rises in the east."

blob = TextBlob(text)

sentiment = blob.sentiment

print(sentiment)

Output:

Sentiment(polarity=0.0, subjectivity=0.0)

- **Polarity:** 0.0 means neutral.
- **Subjectivity:** 0.0 shows it's factual.

Example 3: Mixed Sentiment

Natural Language Processing (NLP)

Here the sentence presents a neutral sentiment but is more opinion-based than factual. The polarity score remains neutral while the subjectivity score reflects an opinion or preference.

```
text = "I enjoy coding, but debugging can be frustrating."

blob = TextBlob(text)

sentiment = blob.sentiment

print(sentiment)
```

Output:

Sentiment(polarity=0.0, subjectivity=0.7)

- **Polarity = 0.0:** This shows a neutral sentiment.
- **Subjectivity = 0.7:** The text is more opinion-based than factual as it's expressing a preference.

Practical Use Cases

This can be useful in various applications including:

1. **Social Media Monitoring:** Analyze sentiment in social media posts to identify public opinion, tracking how positive or negative people feel about topics. This helps in understanding overall sentiment toward trends or events.
2. **Customer Feedback:** Quickly assess customer reviews to identify satisfaction levels, detecting trends in product or service reception. It allows businesses to find customer satisfaction in real-time.
3. **Content Moderation:** Identify and flag negative or inappropriate comments, helping maintain a positive environment in online communities. This improves user experience and promotes healthy discussions.

Limitations

1. **Context Issues:** TextBlob struggles with sarcasm or irony, leading to inaccurate sentiment scores and it may miss the true sentiment of a message. This affects its reliability in certain contexts.
2. **Simplicity:** It relies on a basic lexicon which may miss deeper nuances, making it less accurate for complex or nuanced text. This can limit its application for sophisticated analysis.
3. **Ambiguity:** Mixed sentiments in one sentence can be difficult for it to handle which may result in incorrect classification of sentiment. This reduces its effectiveness in certain cases.

Natural Language Processing (NLP)

NLP Libraries in Python

NLP (Natural Language Processing) helps in the extraction of valuable insights from large amounts of text data. Python has a wide range of libraries specifically designed for text analysis helps in making it easier for data scientists and analysts to process, analyze and derive meaningful insights from text. These libraries handle various NLP tasks such as text preprocessing, tokenization, sentiment analysis, named entity recognition and topic modeling. By using these libraries we can automate text analysis, uncover patterns and make informed, data-driven decisions. In this article, we will see commonly used NLP libraries in Python and find how they can be applied to solve real-world text analysis challenges.

NLP Libraries in Python

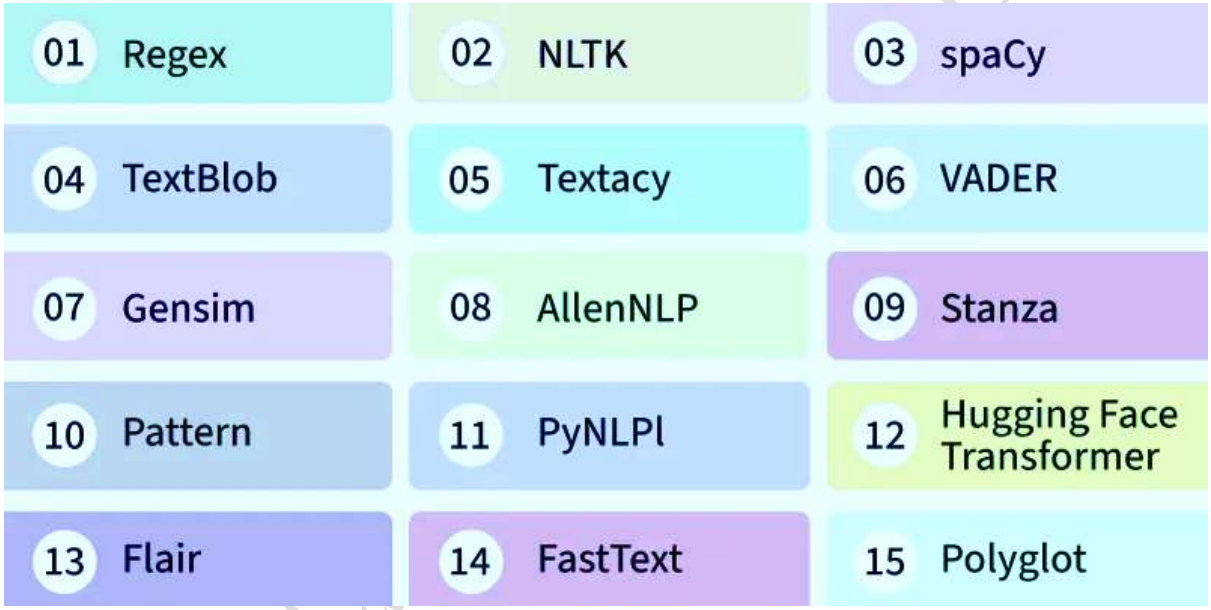


Table of Content

- 1. Regex (Regular Expressions) Library
- 2. NLTK (Natural Language Toolkit)
- 3. spaCy
- 4. TextBlob
- 5. Textacy
- 6. VADER (Valence Aware Dictionary and sEntiment Reasoner)
- 7. Gensim
- 8. AllenNLP

- 9. Stanza
- 10. Pattern
- 11. PyNLPl
- 12. Hugging Face Transformer
- 13. flair
- 14. FastText
- 15. Polyglot

1. Regex (Regular Expressions) Library

Regex is a tool for pattern matching and text modification. It helps in data cleaning, extracting useful information and handling text transformation tasks.

- **Pattern Matching:** Identify and remove unwanted characters, symbols or whitespace in large datasets to prepare text for analysis.
- **Text Extraction:** Extract key pieces of information like product IDs or dates from documents or web pages.

Real-life applications

1. **Data Cleaning:** Extract and clean contact details such as phone numbers or emails from raw datasets**.**
2. **Information Extraction:** Pull out product identifiers, such as SKUs or financial numbers from reports for further analysis.

2. NLTK (Natural Language Toolkit)

NLTK provides various tools for text analysis. It is used for educational and research purposes which offers features for tokenization, stemming and part-of-speech tagging.

- **Tokenization:** Break down text into smaller, meaningful units like words or sentences.
- **Stemming and Lemmatization:** Simplify words to their root form for more consistent analysis.

Real-life applications

1. **Customer Feedback Analysis:** Split reviews into words or sentences for sentiment analysis.
2. **Text Classification:** Automatically categorize content like news articles or social media posts.

3. spaCy

Natural Language Processing (NLP)

spaCy is designed for high-performance text processing. It is good at tasks such as named entity recognition (NER) and dependency parsing which helps in making it ideal for real-time applications.

- **Named Entity Recognition (NER):** Identify and classify entities like names, locations or organizations in text.
- **Dependency Parsing:** Understand the grammatical relationships between words in a sentence.

Real-life applications

1. **Legal Document Analysis:** Identify and extract key entities like company names or legal terms from contracts.
2. **Customer Service Automation:** Extract relevant details like product names or addresses from customer queries for faster responses.

4. TextBlob

TextBlob is an easy-to-use library that simplifies tasks like sentiment analysis and translation. It's great for those just starting with NLP or for quick prototyping.

- **Sentiment Analysis:** Classify the sentiment of a text as positive, negative or neutral.
- **Translation:** Translate text between languages using pre-trained models.

Real-life applications:

1. **Brand Sentiment Monitoring:** Analyze social media posts to get public sentiment about a brand.
2. **Multilingual Customer Support:** Translate support tickets or chat messages to facilitate communication across languages.

5. Textacy

Textacy extends spaCy and provides tools for preprocessing, linguistic feature extraction and topic modeling helps in making it useful for deeper text analysis.

- **Preprocessing:** Clean and prepare text by removing unnecessary words, punctuation and formatting.
- **Topic Modeling:** Identify topics within large corpora to understand underlying themes.

Real-life applications:

1. **Market Research:** Discover trends and themes in customer feedback or product reviews.

Natural Language Processing (NLP)

2. **Content Summarization:** Summarize long articles or reports by extracting the most important topics.

6. VADER (Valence Aware Dictionary and sEntiment Reasoner)

VADER is a rule-based sentiment analysis tool which is designed for analyzing sentiment in social media and informal text. It uses a specialized lexicon to account for the intensity of sentiment including emojis and slang.

- **Sentiment Analysis:** Checks whether a text conveys positive, negative or neutral sentiment.
- **Handling Emojis and Slang:** Understanding the sentiment behind emojis and informal expressions in social media content.

Real-life applications

1. **Social Media Analysis:** Track sentiment in posts or tweets to understand public opinion on a topic.
2. **Customer Feedback Analysis:** Monitor product or service reviews for sentiment trends.

7. Gensim

Gensim is used for unsupervised topic modeling and document similarity analysis which helps in making it ideal for discovering patterns in large text corpora.

- **Topic Modeling:** Identify and classify hidden topics within large datasets using models like LDA.
- **Word Embeddings:** Learn vector representations of words to capture their meanings in context.

Real-life applications

1. **Content Recommendation Systems:** Suggest articles, products or services based on similar topics.
2. **Document Clustering:** Group similar documents together for efficient retrieval.

8. AllenNLP

AllenNLP is built on PyTorch and provides deep learning models for various NLP tasks. It is useful for tasks that require advanced machine learning techniques.

- **Pre-trained Models:** Use pre-trained models for tasks like sentiment analysis and named entity recognition.
- **Custom Model Training:** Train custom models using deep learning tools for specific NLP applications.

Natural Language Processing (NLP)

Real-life applications

1. **Intelligent Customer Support:** Develop AI chatbots to automatically respond to customer queries.
2. **Text Summarization:** Automatically generate concise summaries from long documents.

9. Stanza

Stanza developed by Stanford offers pre-trained models for a variety of NLP tasks like tokenization and named entity recognition. It is built on top of PyTorch which makes it efficient and scalable.

- **Tokenization** : Break down text into smaller components like words or phrases.
- **Dependency Parsing**: Analyze sentence structures to understand relationships between words.

Real-life applications

1. **Legal Text Analysis:** Extract relevant information from legal documents or case files.
2. **Syntactic Text Analysis:** Improve the accuracy of machine learning models by analyzing sentence structure.

10. Pattern

Pattern is a simple library for NLP and web mining with features like part-of-speech tagging and sentiment analysis. It is useful for small projects and learning about NLP.

- **POS Tagging:** Classify words in a sentence into grammatical categories like nouns, verbs or adjectives.
- **Sentiment Analysis:** Find whether the sentiment of text is positive, negative or neutral.

Real-life applications

1. **Basic Text Processing:** Analyze small datasets for sentiment classification or part-of-speech tagging.
2. **Web Scraping:** Extract text from websites for further analysis or research.

11. PyNLPl

PyNLPl is a library for tasks like syntactic parsing and morphological analysis. It's suitable for complex linguistic analysis, especially for multilingual projects.

- **Corpus Processing:** Efficiently handle and process large text corpora for NLP tasks.
- **Syntactic Parsing:** Break down sentences to understand their grammatical structure.

Natural Language Processing (NLP)

Real-life applications

1. **Multilingual Text Processing:** Analyze text in multiple languages helps in making it useful for global projects.
2. **Linguistic Research:** Conduct detailed research on sentence structures and word meanings.

12. Hugging Face Transformer

Hugging Face is known for its transformer-based models such as BERT and GPT. It is used for advanced NLP tasks like text classification, text generation and question answering.

- **Pre-trained Models:** Access pre-trained models like BERT and GPT for various NLP tasks.
- **Fine-Tuning:** Adjust these models to work with specific datasets for better performance on custom tasks.

Real-life applications

1. **AI Assistants:** Enhance virtual assistants such as Siri or Alexa to improve responses.
2. **Content Generation:** Automatically generate text, like articles based on given input.

13. flair

Flair uses deep learning techniques for tasks such as text classification and named entity recognition. It excels in providing high accuracy.

- **NER:** Extract named entities such as people, places or organizations from text.
- **Text Classification:** Classify documents into predefined categories based on their content.

Real-life applications

1. **News Categorization:** Automatically sort articles into categories like politics, health and sports.
2. **Document Classification:** Organize legal or academic documents for easy retrieval.

14. FastText

FastText developed by Facebook AI, is designed for fast text classification and word embeddings. It can handle large datasets efficiently.

- **Text Classification:** Classify text into categories quickly even with large datasets.
- **Word Embeddings:** Create vector representations of words to capture semantic meanings and relationships.

Natural Language Processing (NLP)

Real-life applications

1. **Spam Detection:** Automatically identify spam messages in email or chat platforms.
2. **Real-Time Analysis:** Analyze customer feedback or social media posts in real time.

15. Polyglot

Polyglot is a multilingual library that supports over 130 languages. It's ideal for tasks that require language detection, tokenization or sentiment analysis across various languages.

- **Multilingual Support:** Process text data in more than 130 languages.
- **Language Detection:** Automatically detect the language of any given text.

Real-life applications

1. **Multilingual Customer Support:** Provide global support by handling customer queries in different languages.
2. **Global Sentiment Analysis:** Track sentiment across various languages to gauge worldwide opinions.

Natural Language Processing (NLP)

Normalizing Textual Data in NLP

Text Normalization transforms text into a consistent format improves the quality and makes it easier to process in NLP tasks.

Key steps in text normalization includes:

1. Regular Expressions (RE) are sequences of characters that define search patterns.

- **Textual data** ask systematically collected material consisting of written, printed, or electronically published words, typically either purposefully written or transcribed from speech.
- **Text normalization** is that the method of transforming text into one canonical form that it'd not have had before. Normalizing text before storing or processing it allows for separation of concerns since the input is sure to be consistent before operations are performed thereon. Text normalization requires being conscious of what sort of text is to be normalized and the way it's to be processed afterwards; there's no all-purpose normalization procedure.

Steps Required

Here, we will discuss some basic steps need for Text normalization.

- Input text String,
- Convert all letters of the string to one case(either lower or upper case),
- If numbers are essential to convert to words else remove all numbers,
- Remove punctuations, other formalities of grammar,
- Remove white spaces,
- Remove stop words,
- And any other computations.

We are doing Text normalization with above-mentioned steps, every step can be done in some ways. So we will discuss each and everything in this whole process.

Text String

input string

string = " Python 3.0, released in 2008, was a major revision of the language that is not completely backward compatible and much Python 2 code does not run unmodified on Python 3. With Python 2's end-of-life, only Python 3.6.x[30] and later are supported, with older versions still supporting e.g. Windows 7 (and old installers not restricted to 64-bit Windows)."

print(string)

Output:

" Python 3.0, released in 2008, was a major revision of the language that is not completely backward compatible and much Python 2 code does not run unmodified on Python 3. With Python 2's end-of-life, only Python 3.6.x[30] and later are supported, with older versions still supporting e.g. Windows 7 (and old installers not restricted to 64-bit Windows)."

Case Conversion (Lower Case)

In Python, lower() is a built-in method used for string handling. The lower() methods returns the lowercased string from the given string. It converts all uppercase characters to lowercase. If no uppercase characters exist, it returns the original string.

input string

string = " Python 3.0, released in 2008, was a major revision of the language that is not completely backward compatible and much Python 2 code does not run unmodified on Python 3. With Python 2's end-of-life, only Python 3.6.x[30] and later are supported, with older versions still supporting e.g. Windows 7 (and old installers not restricted to 64-bit Windows)."

Natural Language Processing (NLP)

```
# convert to lower case

lower_string = string.lower()

print(lower_string)
```

Output:

" python 3.0, released in 2008, was a major revision of the language that is not completely backward compatible and much python 2 code does not run unmodified on python 3. with python 2's end-of-life, only python 3.6.x[30] and later are supported, with older versions still supporting e.g. windows 7 (and old installers not restricted to 64-bit windows)."

Removing Numbers

Remove numbers if they're not relevant to your analyses. Usually, regular expressions are used to remove numbers.

```
# import regex

import re


# input string

string = "      Python 3.0, released in 2008, was a major revision of the language that is not completely backward compatible and much Python 2 code does not run unmodified on Python 3. With Python 2's end-of-life, only Python 3.6.x[30] and later are supported, with older versions still supporting e.g. Windows 7 (and old installers not restricted to 64-bit Windows)."


# convert to lower case

lower_string = string.lower()


# remove numbers

no_number_string = re.sub(r'\d+','',lower_string)

print(no_number_string)
```

Output:

" python ., released in , was a major revision of the language that is not completely backward compatible and much python code does not run unmodified on python . with python 's end-of-life, only python ..x[] and later are supported, with older versions still supporting e.g. windows (and old installers not restricted to -bit windows)."

Natural Language Processing (NLP)

Removing punctuation

The part of replacing with punctuation can also be performed using regex. In this, we replace all punctuation by empty string using certain regex.

```
# import regex

import re


# input string

string = "      Python 3.0, released in 2008, was a major revision of the language that is not completely backward compatible and much Python 2 code does not run unmodified on Python 3. With Python 2's end-of-life, only Python 3.6.x[30] and later are supported, with older versions still supporting e.g. Windows 7 (and old installers not restricted to 64-bit Windows)."


# convert to lower case

lower_string = string.lower()


# remove numbers

no_number_string = re.sub(r'\d+','',lower_string)


# remove all punctuation except words and space

no_punc_string = re.sub(r'[^\w\s]','', no_number_string)

print(no_punc_string)
```

Output:

'   python released in was a major revision of the language that is not completely backward compatible and much python code does not run unmodified on python with python s endoflife

```
import re


# input string

string = "      Python 3.0, released in 2008, was a major revision of the language that is not completely backward compatible and much Python 2 code does not run unmodified on Python 3. With Python 2's end-of-life, only Python 3.6.x[30] and later are supported, with
```


Natural Language Processing (NLP)

older versions still supporting e.g. Windows 7 (and old installers not restricted to 64-bit Windows)."

```
# convert to lower case
lower_string = string.lower()

# remove numbers
no_number_string = re.sub(r'\d+','',lower_string)

# remove all punctuation except words and space
no_punc_string = re.sub(r'[^\w\s]','', no_number_string)

# remove white spaces
no_wspace_string = no_punc_string.strip()
print(no_wspace_string)
```

Output:

'python released in was a major revision of the language that is not completely backward compatible and much python code does not run unmodified on python with python s endoflife only python x and later are supported with older versions still supporting eg windows and old installers not restricted to bit windows'

Removing Stop Words

Stop words" are the foremost common words during a language like "the", "a", "on", "is", "all". These words don't carry important meaning and are usually faraway from texts. It is possible to get rid of stop words using tongue Toolkit (NLTK), a set of libraries and programs for symbolic and statistical tongue processing.

```
# download stopwords
import nltk
nltk.download('stopwords')

# import nltk for stopwords
```

Natural Language Processing (NLP)

```
from nltk.corpus import stopwords

stop_words = set(stopwords.words('english'))

print(stop_words)


# assign string

no_wspace_string='python  released in  was a major revision of the language that is not completely backward compatible and much python  code does not run unmodified on python with python s endoflife only python x and later are supported with older versions still supporting eg windows  and old installers not restricted to bit windows'


# convert string to list of words

lst_string = [no_wspace_string][0].split()

print(lst_string)


# remove stopwords

no_stpwords_string=""

for i in lst_string:

   if not i in stop_words:

      no_stpwords_string += i+' '


# removing last space

no_stpwords_string = no_stpwords_string[:-1]

print(no_stpwords_string)
```

Output:

Natural Language Processing (NLP)

```
{'why', 'do', 'her', 'doing', "mightn't", 'didn', 'after', "you're", "hadn't", "it's", 'once', 'than', "weren't", "shouldn't",
"won't", 'my', 'mightn', 'been', 'having', 'he', 'up', 'are', 'should', "wasn't", 'there', 'such', "you'll", 'have', 'mustn',
'for', 'hasn', 'the', "mustn't", 'you', "aren't", 'were', 'had', 'above', 'again', 'she', 'was', 'hers', 'what', 'themselves',
'ma', 'some', "that'll", 'd', 'aren', 'yourself', 're', 'same', 'then', 'on', 'with', "haven't", 'this', 'other', "wouldn't",
"didn't", 'while', "hasn't", 'being', 'did', 'his', 'our', 'll', 'so', 'ourselves', 'if', 'hadn', 'wasn', 'yours', "should've",
'those', 'over', 'all', 'herself', 'now', 'off', 'as', 'more', 's', 'yourselves', 'be', 'against', 'between', 'shouldn', 'me',
'wouldn', 'not', 'very', 've', 'to', 'before', 'y', 'm', 'needn', "shan't", 'him', 'o', 'can', 'during', 'an', 'himself', 'int
o', 'but', 'too', 'in', 'when', 'any', 'they', 'under', 'won', 'does', 'a', 'who', 'i', 't', 'own', 'only', 'them', 'it', 'or',
'both', 'couldn', 'we', "doesn't", 'weren', 'further', 'am', 'of', 'at', 'below', 'just', 'will', 'which', 'through', 'each',
'ours', "she's", 'has', 'where', 'itself', 'theirs', "you'd", "you've", "couldn't", 'and', "don't", 'isn', 'its', 'most', 'is',
'don', 'how', 'down', 'out', 'because', 'few', 'ain', 'here', "isn't", 'their', 'nor', 'that', 'no', 'by', 'your', 'shan', 'unt
il', 'about', 'myself', 'doesn', 'whom', 'these', "needn't", 'from', 'haven'}
['python', 'released', 'in', 'was', 'a', 'major', 'revision', 'of', 'the', 'language', 'that', 'is', 'not', 'completely', 'back
ward', 'compatible', 'and', 'much', 'python', 'code', 'does', 'not', 'run', 'unmodified', 'on', 'python', 'with', 'python',
's', 'endoflife', 'only', 'python', 'x', 'and', 'later', 'are', 'supported', 'with', 'older', 'versions', 'still', 'supportin
g', 'eg', 'windows', 'and', 'old', 'installers', 'not', 'restricted', 'to', 'bit', 'windows']
python released major revision language completely backward compatible much python code run unmodified python python endoflife
python x later supported older versions still supporting eg windows old installers restricted bit windows
```

In this, we can normalize the textual data using Python. Below is the complete python program:

 # import regex

import re

 # download stopwords

import nltk

nltk.download('stopwords')

 # import nltk for stopwords

from nltk.corpus import stopwords

stop_words = set(stopwords.words('english'))

 # input string

string = " Python 3.0, released in 2008, was a major revision of the language that is not completely backward compatible and much Python 2 code does not run unmodified on Python 3. With Python 2's end-of-life, only Python 3.6.x[30] and later are supported, with older versions still supporting e.g. Windows 7 (and old installers not restricted to 64-bit Windows)."

 # convert to lower case

Natural Language Processing (N...

```
lower_string = string.lower()

# remove numbers
no_number_string = re.sub(r'\d+','',lower_string)

# remove all punctuation except words and space
no_punc_string = re.sub(r'[^\w\s]','', no_number_string)

# remove white spaces
no_wspace_string = no_punc_string.strip()
no_wspace_string

# convert string to list of words
lst_string = [no_wspace_string][0].split()
print(lst_string)

# remove stopwords
no_stpwords_string=""
for i in lst_string:
   if not i in stop_words:
      no_stpwords_string += i+' '

# removing last space
no_stpwords_string = no_stpwords_string[:-1]

# output
print(no_stpwords_string)
```

Natural Language Processing (NLP)

Python RegEx

A Regular Expression or RegEx is a special sequence of characters that uses a search pattern to find a string or set of strings.

It can detect the presence or absence of a text by matching it with a particular pattern and also can split a pattern into one or more sub-patterns.

Regex Module in Python

Python has a built-in module named "**re**" that is used for regular expressions in Python. We can import this module by using import statement.

Importing re module in Python using following command:

import re

How to Use RegEx in Python?

You can use RegEx in Python after importing re module.

Example:

This Python code uses regular expressions to search for the word **"portal"** in the given string and then prints the start and end indices of the matched word within the string.

```
import re


s = 'GeeksforGeeks: A computer science portal for geeks'

match = re.search(r'portal', s)


print('Start Index:', match.start())

print('End Index:', match.end())
```

Output

Start Index: 34

End Index: 40

Natural Language Processing (NLP)

Note: *Here r character (r'portal') stands for raw, not regex. The raw string is slightly different from a regular string, it won't interpret the \ character as an escape character. This is because the regular expression engine uses \ character for its own escaping purpose.*

Before starting with the Python regex module let's see how to actually write regex using metacharacters or special sequences.

RegEx Functions

The re module in Python provides various functions that help search, match, and manipulate strings using regular expressions.

Below are main functions available in the re module:

Function	Description
re.findall()	finds and returns all matching occurrences in a list
re.compile()	Regular expressions are compiled into pattern objects
re.split()	Split string by the occurrences of a character or a pattern.
re.sub()	Replaces all occurrences of a character or patter with a replacement string.
resubn	It's similar to re.sub() method but it returns a tuple: (new_string, number_of_substitutions)
re.escape()	Escapes special character
re.search()	Searches for first occurrence of character or pattern

Let's see the working of these RegEx functions with definition and examples:

1. re.findall()

Returns all non-overlapping matches of a pattern in the string as a list. It scans the string from left to right.

Example: This code uses regular expression **\d+** to find all sequences of one or more digits in the given string.

```
import re

string = """Hello my Number is 123456789 and

        my friend's number is 987654321"""


regex = '\d+'

match = re.findall(regex, string)

print(match)
```

Output

```
['123456789', '987654321']
```

2. re.compile()

Compiles a regex into a pattern object, which can be reused for matching or substitutions.

Example 1: This pattern **[a-e]** matches all lowercase letters between '**a**' and '**e**', in the input string **"Aye, said Mr. Gibenson Stark".** The output should be **['e', 'a', 'd', 'b', 'e']**, which are matching characters.

```
import re

p = re.compile('[a-e]')

print(p.findall("Aye, said Mr. Gibenson Stark"))
```

Output

```
['e', 'a', 'd', 'b', 'e', 'a']
```

Explanation:

- First occurrence is 'e' in "Aye" and not 'A', as it is Case Sensitive.
- Next Occurrence is 'a' in "said", then 'd' in "said", followed by 'b' and 'e' in "Gibenson", the Last 'a' matches with "Stark".
- Metacharacter backslash '\' has a very important role as it signals various sequences. If the backslash is to be used without its special meaning as metacharacter, use'\\'

Natural Language Processing (NLP)

Example 2: The code uses regular expressions to find and list all single digits and sequences of digits in the given input strings. It finds single digits with **\d** and sequences of digits with **\d+**.

```
import re

p = re.compile('\d')

print(p.findall("I went to him at 11 A.M. on 4th July 1886"))


p = re.compile('\d+')

print(p.findall("I went to him at 11 A.M. on 4th July 1886"))
```

Output

```
['1', '1', '4', '1', '8', '8', '6']

['11', '4', '1886']
```

Example 3: Word and non-word characters

- **\w** matches a single word character.
- **\w+** matches a group of word characters.
- **\W** matches non-word characters.

```
import re


p = re.compile('\w')

print(p.findall("He said * in some_lang."))


p = re.compile('\w+')

print(p.findall("I went to him at 11 A.M., he \

said *** in some_language."))


p = re.compile('\W')

print(p.findall("he said *** in some_language."))
```

Natural Language Processing (NLP)

Output

['H', 'e', 's', 'a', 'i', 'd', 'i', 'n', 's', 'o', 'm', 'e', '_', 'l', 'a', 'n', 'g']

['I', 'went', 'to', 'him', 'at', '11', 'A', 'M', 'he', 'said', 'in', 'some_language']

[' ', ' ', '*', '*', '*', ' ', ' ', '.']

Example 4: The regular expression pattern '**ab***' to find and list all occurrences of '**ab**' followed by zero or more '**b**' characters. In the input string "ababbaabbb". It returns the following list of matches: ['ab', 'abb', 'abbb'].

```
import re

p = re.compile('ab*')

print(p.findall("ababbaabbb"))
```

Output

['ab', 'abb', 'a', 'abbb']

Explanation:

- Output 'ab', is valid because of single 'a' accompanied by single 'b'.
- Output 'abb', is valid because of single 'a' accompanied by 2 'b'.
- Output 'a', is valid because of single 'a' accompanied by 0 'b'.
- Output 'abbb', is valid because of single 'a' accompanied by 3 'b'.

3. re.split()

Splits a string wherever the pattern matches. The remaining characters are returned as list elements.

Syntax:

re.split(pattern, string, maxsplit=0, flags=0)

- **pattern:** Regular expression to match split points.
- **string:** The input string to split.
- **maxsplit (optional):** Limits the number of splits. Default is 0 (no limit).
- **flags (optional):** Apply regex flags like re.IGNORECASE.

Example 1: Splitting by non-word characters or digits

Natural Language Processing (NLP)

This example demonstrates how to split a string using different patterns like non-word characters (\W+), apostrophes, and digits (\d+).

```
from re import split


print(split('\W+', 'Words, words , Words'))
print(split('\W+', "Word's words Words"))
print(split('\W+', 'On 12th Jan 2016, at 11:02 AM'))
print(split('\d+', 'On 12th Jan 2016, at 11:02 AM'))
```

Output

```
['Words', 'words', 'Words']
['Word', 's', 'words', 'Words']
['On', '12th', 'Jan', '2016', 'at', '11', '02', 'AM']
['On ', 'th Jan ', ', at ', ':', ' AM']
```

Example 2: Using maxsplit and flags

This example shows how to limit the number of splits using maxsplit, and how flags can control case sensitivity.

```
import re
print(re.split('\d+', 'On 12th Jan 2016, at 11:02 AM', 1))
print(re.split('[a-f]+', 'Aey, Boy oh boy, come here', flags=re.IGNORECASE))
print(re.split('[a-f]+', 'Aey, Boy oh boy, come here'))
```

Output

```
['On ', 'th Jan 2016, at 11:02 AM']
['', 'y, ', 'oy oh ', 'oy, ', 'om', ' h', 'r', '']
['A', 'y, Boy oh ', 'oy, ', 'om', ' h', 'r', '']
```

Note: *In the second and third cases of the above , [a-f]+ splits the string using any combination of lowercase letters from 'a' to 'f'. The re.IGNORECASE flag includes uppercase letters in the match.*

4. re.sub()

Natural Language Processing (NLP)

The re.sub() function replaces all occurrences of a pattern in a string with a replacement string.

Syntax:

re.sub(pattern, repl, string, count=0, flags=0)

- **pattern**: The regex pattern to search for.
- **repl**: The string to replace matches with.
- **string**: The input string to process.
- **count** (optional): Maximum number of substitutions (default is 0, which means replace all).
- **flags (optional)**: Regex flags like re.IGNORECASE.

Example 1: The following examples show different ways to replace the pattern '**ub**' with '**~***', using various flags and count values.

```
import re


# Case-insensitive replacement of all 'ub'
print(re.sub('ub', '~*', 'Subject has Uber booked already', flags=re.IGNORECASE))


# Case-sensitive replacement of all 'ub'
print(re.sub('ub', '~*', 'Subject has Uber booked already'))


# Replace only the first 'ub', case-insensitive
print(re.sub('ub', '~*', 'Subject has Uber booked already', count=1, flags=re.IGNORECASE))


# Replace "AND" with "&", ignoring case
print(re.sub(r'\sAND\s', ' & ', 'Baked Beans And Spam', flags=re.IGNORECASE))
```

Output

```
S~*ject has ~*er booked already
S~*ject has Uber booked already
S~*ject has Uber booked already
```

Natural Language Processing (NLP)

Baked Beans & Spam

5. re.subn()

re.subn() function works just like **re.sub()**, but instead of returning only the modified string, it returns a tuple: **(new_string, number_of_substitutions)**

Syntax:

re.subn(pattern, repl, string, count=0, flags=0)

Example: Substitution with count

This example shows how re.subn() gives both the replaced string and the number of times replacements were made.

```
import re


# Case-sensitive replacement
print(re.subn('ub', '~*', 'Subject has Uber booked already'))


# Case-insensitive replacement
t = re.subn('ub', '~*', 'Subject has Uber booked already', flags=re.IGNORECASE)
print(t)
print(len(t))      # tuple length
print(t[0])        # modified string
```

Output

```
('S~*ject has Uber booked already', 1)
('S~*ject has ~*er booked already', 2)
2
S~*ject has ~*er booked already
```

6. re.escape()

re.escape() function adds a backslash (****) before all special characters in a string. This is useful when you want to match a string literally, including any characters that have special meaning in regex (like ., *, [,], etc.).

Syntax:

Natural Language Processing (NLP)

re.escape(string)

Example: Escaping special characters

This example shows how re.escape() treats spaces, brackets, dashes, and tabs as literal characters.

```
import re

print(re.escape("This is Awesome even 1 AM"))

print(re.escape("I Asked what is this [a-9], he said \t ^WoW"))
```

Output

```
This\ is\ Awesome\ even\ 1\ AM

I\ Asked\ what\ is\ this\ \[a\-9\]\,\ he\ said\ \          \ \^WoW
```

7. re.search()

The re.search() function searches for the first occurrence of a pattern in a string. It returns a **match object** if found, otherwise **None**.

Note: *Use it when you want to check if a pattern exists or extract the first match.*

Example: Search and extract values

This example searches for a date pattern with a month name (letters) followed by a day (digits) in a sentence.

```
import re


regex = r"([a-zA-Z]+) (\d+)"

match = re.search(regex, "I was born on June 24")


if match:
    print("Match at index %s, %s" % (match.start(), match.end()))
    print("Full match:", match.group(0))
    print("Month:", match.group(1))
    print("Day:", match.group(2))
else:
```

Natural Language Processing (NLP)

```
print("The regex pattern does not match.")
```

Output

Match at index 14, 21

Full match: June 24

Month: June

Day: 24

Meta-characters

Metacharacters are special characters in regular expressions used to define search patterns. The re module in Python supports several metacharacters that help you perform powerful pattern matching.

Below is a quick reference table:

MetaCharacters	Description
\	Used to drop the special meaning of character following it
[]	Represent a character class
^	Matches the beginning
$	Matches the end
.	Matches any character except newline
|	Means OR (Matches with any of the characters separated by it.
?	Matches zero or one occurrence

MetaCharacters	Description
*	Any number of occurrences (including 0 occurrences)
+	One or more occurrences
{}	Indicate the number of occurrences of a preceding regex to match.
()	Enclose a group of Regex

Let's discuss each of these metacharacters in detail:

1. \ - Backslash

The backslash (****) makes sure that the character is not treated in a special way. This can be considered a way of escaping metacharacters.

For example, if you want to search for the dot(.) in the string then you will find that dot(.) will be treated as a special character as is one of the metacharacters (as shown in the above table). So for this case, we will use the backslash(\) just before the dot(.) so that it will lose its specialty. See the below example for a better understanding.

Example: The first search (**re.search(r'.', s)**) matches any character, not just the period, while the second search (**re.search(r'\.', s)**) specifically looks for and matches the period character.

```
import re


s = 'geeks.forgeeks'


# without using \

match = re.search(r'.', s)

print(match)


# using \
```

Natural Language Processing

```
match = re.search(r'\.', s)

print(match)
```

Output

```
<re.Match object; span=(0, 1), match='g'>

<re.Match object; span=(5, 6), match='.'>
```

2. [] - Square Brackets

Square Brackets ([]) represent a character class consisting of a set of characters that we wish to match. For example, the character class [abc] will match any single a, b, or c.

We can also specify a range of characters using - inside the square brackets. For example,

- [0, 3] is sample as [0123]
- [a-c] is same as [abc]

We can also invert the character class using the caret(^) symbol. For example,

- [^0-3] means any character except 0, 1, 2, or 3
- [^a-c] means any character except a, b, or c

Example: In this code, you're using regular expressions to find all the characters in the string that fall within the range of 'a' to 'm'. The **re.findall()** function returns a list of all such characters. In the given string, the characters that match this pattern are: 'c', 'k', 'b', 'f', 'j', 'e', 'h', 'l', 'd', 'g'.

```
import re


string = "The quick brown fox jumps over the lazy dog"

pattern = "[a-m]"

result = re.findall(pattern, string)


print(result)
```

Output

```
['h', 'e', 'i', 'c', 'k', 'b', 'f', 'j', 'm', 'e', 'h', 'e', 'l', 'a', 'd', 'g']
```

3. ^ - Caret

Natural Language Processing (NLP)

Caret (^) symbol matches the beginning of the string i.e. checks whether the string starts with the given character(s) or not. For example -

- ^g will check if the string starts with g such as geeks, globe, girl, g, etc.
- ^ge will check if the string starts with ge such as geeks, geeksforgeeks, etc.

Example: This code uses regular expressions to check if a list of strings starts with **"The"**. If a string begins with **"The," it's marked as "Matched"** otherwise, it's labeled as **"Not matched".**

```
import re

regex = r'^The'

strings = ['The quick brown fox', 'The lazy dog', 'A quick brown fox']

for string in strings:
    if re.match(regex, string):
        print(f'Matched: {string}')
    else:
        print(f'Not matched: {string}')
```

Output

Matched: The quick brown fox

Matched: The lazy dog

Not matched: A quick brown fox

4. $ - Dollar

Dollar($) symbol matches the end of the string i.e checks whether the string ends with the given character(s) or not. For example-

- s$ will check for the string that ends with a such as geeks, ends, s, etc.
- ks$ will check for the string that ends with ks such as geeks, geeksforgeeks, ks, etc.

Example: This code uses a regular expression to check if the string ends with **"World!".** If a match is found, it prints **"Match found!"** otherwise, it prints **"Match not found"**.

```
import re


string = "Hello World!"

pattern = r"World!$"
```

Natural Language Processing (NLP)

```
match = re.search(pattern, string)

if match:
    print("Match found!")
else:
    print("Match not found.")
```

Output

```
Match found!
```

5. . - Dot

Dot(.) symbol matches only a single character except for the newline character (\n). For example -

- a.b will check for the string that contains any character at the place of the dot such as acb, acbd, abbb, etc
- .. will check if the string contains at least 2 characters

Example: This code uses a regular expression to search for the pattern **"brown.fox"** within the string. The dot (.) in the pattern represents any character. If a match is found, it prints **"Match found!"** otherwise, it prints **"Match not found"**.

```
import re


string = "The quick brown fox jumps over the lazy dog."

pattern = r"brown.fox"


match = re.search(pattern, string)

if match:
    print("Match found!")
else:
    print("Match not found.")
```

Output

Natural Language Processing (NLP)

Match found!

6. | - Or

The | operator means either pattern on its left or right can match. a|b will match any string that contains a or b such as acd, bcd, abcd, etc.

7. ? - Question Mark

The question mark (?) indicates that the preceding element should be matched zero or one time. It allows you to specify that the element is optional, meaning it may occur once or not at all.

For example, ab?c will be matched for the string ac, acb, dabc but will not be matched for abbc because there are two b. Similarly, it will not be matched for abdc because b is not followed by c.

8.* - Star

Star (*) symbol matches zero or more occurrences of the regex preceding the * symbol.

For example, ab*c will be matched for the string ac, abc, abbbc, dabc, etc. but will not be matched for abdc because b is not followed by c.

9. + - Plus

Plus (+) symbol matches one or more occurrences of the regex preceding the + symbol.

For example, ab+c will be matched for the string abc, abbc, dabc, but will not be matched for ac, abdc, because there is no b in ac and b, is not followed by c in abdc.

10. {m, n} - Braces

Braces match any repetitions preceding regex from m to n both inclusive.

For example, a{2, 4} will be matched for the string aaab, baaaac, gaad, but will not be matched for strings like abc, bc because there is only one a or no a in both the cases.

11. (<regex>) - Group

Group symbol is used to group sub-patterns.

For example, (a|b)cd will match for strings like acd, abcd, gacd, etc.

Special Sequences

Special sequences do not match for the actual character in the string instead it tells the specific location in the search string where the match must occur. It makes it easier to write commonly used patterns.

List of special sequences

Natural Language Processing (NLP)

Special Sequence	Description	Examples	
\A	Matches if the string begins with the given character	\Afor	for geeks
			for the world
\b	Matches if the word begins or ends with the given character. \b(string) will check for the beginning of the word and (string)\b will check for the ending of the word.	\bge	geeks
			get
\B	It is the opposite of the \b i.e. the string should not start or end with the given regex.	\Bge	together
			forge
\d	Matches any decimal digit, this is equivalent to the set class [0-9]	\d	123
			gee1
\D	Matches any non-digit character, this is equivalent to the set class [^0-9]	\D	geeks
			geek1

Special Sequence	Description	Examples	
\s	Matches any whitespace character.	\s	gee ks a bc a
\S	Matches any non-whitespace character	\S	a bd abcd
\w	Matches any alphanumeric character, this is equivalent to the class [a-zA-Z0-9_].	\w	123 geeKs4
\W	Matches any non-alphanumeric character.	\W	>$ gee<>
\Z	Matches if the string ends with the given regex	ab\Z	abcdab abababab

Sets for character matching

A **Set** is a set of characters enclosed in '[]' brackets. Sets are used to match a single character in the set of characters specified between brackets. Below is the list of Sets:

Set	Description
\{n,\}	Quantifies the preceding character or group and matches at least n occurrences.
*	Quantifies the preceding character or group and matches zero or more occurrences.
[0123]	Matches the specified digits (0, 1, 2, or 3)
[^arn]	matches for any character EXCEPT a, r, and n
\d	Matches any digit (0-9).
[0-5][0-9]	matches for any two-digit numbers from 00 and 59
\w	Matches any alphanumeric character (a-z, A-Z, 0-9, or _).
[a-n]	Matches any lower case alphabet between a and n.
\D	Matches any non-digit character.
[arn]	matches where one of the specified characters (a, r, or n) are present
[a-zA-Z]	matches any character between a and z, lower case OR upper case

Set	Description
[0-9]	matches any digit between 0 and 9

Match Object

A Match object contains all the information about the search and the result and if there is no match found then None will be returned. Let's see some of the commonly used methods and attributes of the match object.

1. Getting the string and the regex

match.re attribute returns the regular expression passed and match.string attribute returns the string passed.

Example:

The code searches for the letter "G" at a word boundary in the string "Welcome to GeeksForGeeks" and prints the regular expression pattern **(res.re)** and the original string **(res.string)**.

```
import re

s = "Welcome to GeeksForGeeks"

res = re.search(r"\bG", s)


print(res.re)

print(res.string)
```

Output

```
re.compile('\\bG')

Welcome to GeeksForGeeks
```

2. Getting index of matched object

- **start()** method returns the starting index of the matched substring
- **end()** method returns the ending index of the matched substring
- **span()** method returns a tuple containing the starting and the ending index of the matched substring

Example: Getting index of matched object

Natural Language Processing (NLP)

The code searches for substring "**Gee**" at a word boundary in string "Welcome to GeeksForGeeks" and prints start index of the match (**res.start()**), end index of the match (**res.end()**) and span of the match (**res.span()**).

```
import re


s = "Welcome to GeeksForGeeks"

res = re.search(r"\bGee", s)


print(res.start())

print(res.end())

print(res.span())
```

Output

```
11

14

(11, 14)
```

3. Getting matched substring

group() method returns the part of the string for which the patterns match. See the below example for a better understanding.

Example: Getting matched substring

The code searches for a sequence of two non-digit characters followed by a space and the letter 't' in the string "Welcome to GeeksForGeeks" and prints the matched text using **res.group().**

```
import re

s = "Welcome to GeeksForGeeks"

res = re.search(r"\D{2} t", s)

print(res.group())
```

Output

```
me t
```

Natural Language Processing (NLP)

In the above example, our pattern specifies for the string that contains at least 2 characters which are followed by a space, and that space is followed by a **t**.

Basic RegEx Patterns

Let's understand some of the basic regular expressions. They are as follows:

1. Character Classes

Character classes allow matching any one character from a specified set. They are enclosed in square brackets [].

```
import re

print(re.findall(r'[Gg]eeks', 'GeeksforGeeks: \
        A computer science portal for geeks'))
```

Output

```
['Geeks', 'Geeks', 'geeks']
```

2. Ranges

In RegEx, a range allows matching characters or digits within a span using - inside []. For example, [0-9] matches digits, [A-Z] matches uppercase letters.

```
import re

print('Range',re.search(r'[a-zA-Z]', 'x'))
```

Output

```
Range <re.Match object; span=(0, 1), match='x'>
```

3. Negation

Negation in a character class is specified by placing a ^ at the beginning of the brackets, meaning match anything except those characters.

Syntax:

[^a-z]

Example:

```
import re


print(re.search(r'[^a-z]', 'c'))
```

Natural Language Processing (NLP)

```
print(re.search(r'G[^e]', 'Geeks'))
```

Output

None

None

3. Shortcuts

Shortcuts are shorthand representations for common character classes. Let's discuss some of the shortcuts provided by the regular expression engine.

- \w - matches a word character
- \d - matches digit character
- \s - matches whitespace character (space, tab, newline, etc.)
- \b - matches a zero-length character

```
import re


print('Geeks:', re.search(r'\bGeeks\b', 'Geeks'))

print('GeeksforGeeks:', re.search(r'\bGeeks\b', 'GeeksforGeeks'))
```

Output

Geeks: <_sre.SRE_Match object; span=(0, 5), match='Geeks'>

GeeksforGeeks: None

4. Beginning and End of String

The ^ character chooses the beginning of a string and the $ character chooses the end of a string.

```
import re



# Beginning of String

match = re.search(r'^Geek', 'Campus Geek of the month')

print('Beg. of String:', match)
```

Natural Language Processing (NLP)

```
match = re.search(r'^Geek', 'Geek of the month')

print('Beg. of String:', match)


# End of String

match = re.search(r'Geeks$', 'Compute science portal-GeeksforGeeks')

print('End of String:', match)
```

Output

```
Beg. of String: None

Beg. of String: <_sre.SRE_Match object; span=(0, 4), match='Geek'>

End of String: <_sre.SRE_Match object; span=(31, 36), match='Geeks'>
```

5. Any Character

The . character represents any single character outside a bracketed character class.

```
import re

print('Any Character', re.search(r'p.th.n', 'python 3'))
```

Output

```
Any Character <_sre.SRE_Match object; span=(0, 6), match='python'>
```

6. Optional Characters

Regular expression engine allows you to specify optional characters using the ? character. It allows a character or character class either to present once or else not to occur. Let's consider the example of a word with an alternative spelling - color or colour.

```
import re


print('Color',re.search(r'colou?r', 'color'))

print('Colour',re.search(r'colou?r', 'colour'))
```

Output

```
Color <_sre.SRE_Match object; span=(0, 5), match='color'>
```

Natural Language Processing (NLP)

Colour <_sre.SRE_Match object; span=(0, 6), match='colour'>

7. Repetition

Repetition enables you to repeat the same character or character class. Consider an example of a date that consists of day, month, and year. Let's use a regular expression to identify the date (mm-dd-yyyy).

```
import re

print('Date{mm-dd-yyyy}:', re.search(r'[\d]{2}-[\d]{2}-[\d]{4}','18-08-2020'))
```

Output

Date{mm-dd-yyyy}: <_sre.SRE_Match object; span=(0, 10), match='18-08-2020'>

Here, the regular expression engine checks for two consecutive digits. Upon finding the match, it moves to the hyphen character. After then, it checks the next two consecutive digits and the process is repeated.

Let's discuss three other regular expressions under repetition.

7.1 Repetition ranges

The repetition range is useful when you have to accept one or more formats. Consider a scenario where both three digits, as well as four digits, are accepted. Let's have a look at the regular expression.

```
import re


print('Three Digit:', re.search(r'[\d]{3,4}', '189'))

print('Four Digit:', re.search(r'[\d]{3,4}', '2145'))
```

Output

Three Digit: <_sre.SRE_Match object; span=(0, 3), match='189'>

Four Digit: <_sre.SRE_Match object; span=(0, 4), match='2145'>

7.2 Open-Ended Ranges

There are scenarios where there is no limit for a character repetition. In such scenarios, you can set the upper limit as infinitive. A common example is matching street addresses. Let's have a look

```
import re
```

Natural Language Processing (NLP)

```
print(re.search(r'[\d]{1,}','5th Floor, A-118,\
Sector-136, Noida, Uttar Pradesh - 201305'))
```

Output

```
<_sre.SRE_Match object; span=(0, 1), match='5'>
```

7.3 Shorthand

Shorthand characters allow you to use + character to specify one or more ({1,}) and * character to specify zero or more ({0,}.

```
import re

print(re.search(r'[\d]+', '5th Floor, A-118,\
Sector-136, Noida, Uttar Pradesh - 201305'))
```

Output

```
<_sre.SRE_Match object; span=(0, 1), match='5'>
```

8. Grouping

Grouping is the process of separating an expression into groups by using parentheses, and it allows you to fetch each individual matching group.

```
import re

grp = re.search(r'([\d]{2})-([\d]{2})-([\d]{4})', '26-08-2020')

print(grp)
```

Output

```
<_sre.SRE_Match object; span=(0, 10), match='26-08-2020'>
```

Let's see some of its functionality.

8.1 Return the entire match

The re module allows you to return the entire match using the *group()* method

```
import re

grp = re.search(r'([\d]{2})-([\d]{2})-([\d]{4})','26-08-2020')
```

Natural Language Processing (NLP)

```
print(grp.group())
```

Output

```
26-08-2020
```

8.2 Return a tuple of matched groups

You can use groups() method to return a tuple that holds individual matched groups

```
import re
grp = re.search(r'([\d]{2})-([\d]{2})-([\d]{4})','26-08-2020')
print(grp.groups())
```

Output

```
('26', '08', '2020')
```

8.3 Retrieve a single group

Upon passing the index to a group method, you can retrieve just a single group.

```
import re
grp = re.search(r'([\d]{2})-([\d]{2})-([\d]{4})','26-08-2020')
print(grp.group(3))
```

Output

```
2020
```

8.4 Name your groups

The re module allows you to name your groups. Let's look into the syntax.

```
import re
match = re.search(r'(?P<dd>[\d]{2})-(?P<mm>[\d]{2})-(?P<yyyy>[\d]{4})',
            '26-08-2020')
print(match.group('mm'))
```

Output

```
08
```

Natural Language Processing (NLP)

8.5 Individual match as a dictionary

We have seen how regular expression provides a tuple of individual groups. Not only tuple, but it can also provide individual match as a dictionary in which the name of each group acts as the dictionary key.

```
import re

match = re.search(r'(?P<dd>[\d]{2})-(?P<mm>[\d]{2})-(?P<yyyy>[\d]{4})',

        '26-08-2020')

print(match.groupdict())
```

Output

```
{'dd': '26', 'mm': '08', 'yyyy': '2020'}
```

9. Lookahead

In the case of a negated character class, it won't match if a character is not present to check against the negated character. We can overcome this case by using lookahead; it accepts or rejects a match based on the presence or absence of content.

```
import re

print('negation:', re.search(r'n[^e]', 'Python'))

print('lookahead:', re.search(r'n(?!e)', 'Python'))
```

Output

```
negation: None

lookahead: <_sre.SRE_Match object; span=(5, 6), match='n'>
```

Lookahead can also disqualify the match if it is not followed by a particular character. This process is called a positive lookahead, and can be achieved by simply replacing ! character **with = character**.

```
import re

print('positive lookahead', re.search(r'n(?=e)', 'jasmine'))
```

Output

```
positive lookahead <_sre.SRE_Match object; span=(5, 6), match='n'>
```

10. Substitution

Natural Language Processing (NLP)

The regular expression can replace the string and returns the replaced one using the re.sub method. It is useful when you want to avoid characters such as /, -, ., etc. before storing it to a database. It takes three arguments:

- the regular expression
- the replacement string
- the source string being searched

Regex Tutorial - How to write Regular Expressions

A regular expression (regex) is a sequence of characters that define a search pattern. Here's how to write regular expressions:

1. Start by understanding the special characters used in regex, such as ".", "*", "+", "?", and more.
2. Choose a programming language or tool that supports regex, such as Python, Perl, or grep.
3. Write your pattern using the special characters and literal characters.
4. Use the appropriate function or method to search for the pattern in a string.

Examples:

1. To match a sequence of literal characters, simply write those characters in the pattern.
2. To match a single character from a set of possibilities, use square brackets, e.g. [0123456789] matches any digit.
3. To match zero or more occurrences of the preceding expression, use the star (*) symbol.
4. To match one or more occurrences of the preceding expression, use the plus (+) symbol.
5. It is important to note that regex can be complex and difficult to read, so it is recommended to use tools like regex testers to debug and optimize your patterns.

A regular expression (sometimes called a rational expression) is a sequence of characters that define a search pattern, mainly for use in pattern matching with strings, or string matching, i.e. "find and replace" like operations. Regular expressions are a generalized way to match patterns with sequences of characters. It is used in every programming language like C++, Java and Python.

Natural Language Processing (NLP)

What is a regular expression and what makes it so important?

Regex is used in *Google Analytics* in URL matching in supporting search and replaces in most popular editors like Sublime, Notepad++, Brackets, Google Docs, and Microsoft Word.

Example : Regular expression for an email address : ^([a-zA-Z0-9_\-\.]+)@([a-zA-Z0-9_\-\.]+)\.([a-zA-Z]{2,5})$

The above regular expression can be used for checking if a given set of characters is an email address or not.

How to write regular expressions?

There are certain elements used to write regular expressions as mentioned below:

1. Repeaters (*, +, and { })

These symbols act as repeaters and tell the computer that the preceding character is to be used for more than just one time.

2. The asterisk symbol (*)

It tells the computer to match the preceding character (or set of characters) for 0 or more times (upto infinite).

Example : The regular expression ab*c will give ac, abc, abbc, abbbc….and so on

3. The Plus symbol (+)

It tells the computer to repeat the preceding character (or set of characters) at atleast one or more times(up to infinite).

Example : The regular expression ab+c will give abc, abbc, abbbc, … and so on.

4. The curly braces { … }

It tells the computer to repeat the preceding character (or set of characters) for as many times as the value inside this bracket.

Example : {2} means that the preceding character is to be repeated 2 times, {min,} means the preceding character is matches min or more times. {min,max} means that the preceding character is repeated at least min & at most max times.

5. Wildcard (.)

The dot symbol can take the place of any other symbol, that is why it is called the wildcard character.

Natural Language Processing (NLP)

Example :
The Regular expression .* will tell the computer that any character can be used any number of times.

6. Optional character (?)

This symbol tells the computer that the preceding character may or may not be present in the string to be matched.

Example :
We may write the format for document file as – "docx?"
The '?' tells the computer that x may or may not be present in the name of file format.

7. The caret (^) symbol (***Setting position for the match)***

The caret symbol tells the computer that the match must start at the beginning of the string or line.

Example : ^\d{3} will match with patterns like "901" in "901-333-".

8. The dollar ($) symbol

It tells the computer that the match must occur at the end of the string or before \n at the end of the line or string.

Example : -\d{3}$ will match with patterns like "-333" in "-901-333".

9. Character Classes

A character class matches any one of a set of characters. It is used to match the most basic element of a language like a letter, a digit, a space, a symbol, etc.

***\s**: matches any whitespace characters such as space and tab.*
***\S**: matches any non-whitespace characters.*
***\d**: matches any digit character.*
***\D**: matches any non-digit characters.*
***\w** : matches any word character (basically alpha-numeric)*
***\W**: matches any non-word character.*
***\b**: matches any word boundary (this would include spaces, dashes, commas, semi-colons, etc.*
***[set_of_characters]:** Matches any single character in set_of_characters. By default, the match is case-sensitive.*

Example : [abc] will match characters a,b and c in any string.

10. [^set_of_characters] ***Negation:***

Natural Language Processing (NLP)

Matches any single character that is not in set_of_characters. By default, the match is case-sensitive.

Example : [^abc] will match any character except a,b,c .

11. [first-last] *Character range:*

Matches any single character in the range from first to last.

Example : [a-zA-z] will match any character from a to z or A to Z.

12. The Escape Symbol (\ )

If you want to match for the actual '+', '.' etc characters, add a backslash(\ ) before that character. This will tell the computer to treat the following character as a search character and consider it for a matching pattern.

Example : \d+[\+-x*]\d+ will match patterns like "2+2" and "3*9" in "(2+2) * 3*9".

13. Grouping Characters ( )

A set of different symbols of a regular expression can be grouped together to act as a single unit and behave as a block, for this, you need to wrap the regular expression in the parenthesis( ).

Example : ([A-Z]\w+) contains two different elements of the regular expression combined together. This expression will match any pattern containing uppercase letter followed by any character.

14. Vertical Bar (|)

Matches any one element separated by the vertical bar (|) character.

Example : th(e|is|at) will match words - the, this and that.

15. \number

Backreference: allows a previously matched sub-expression(expression captured or enclosed within circular brackets) to be identified subsequently in the same regular expression. \n means that group enclosed within the n-th bracket will be repeated at current position.

Example : ([a-z])\1 will match "ee" in Geek because the character at second position is same as character at position 1 of the match.

16. Comment (?# comment)

Inline comment: The comment ends at the first closing parenthesis.

Example : \bA(?#This is an inline comment)\w+\b

17. # [to end of line]

Natural Language Processing (NLP)

X-mode comment. The comment starts at an unescaped # and continues to the end of the line.

Example : (?x)\bA\w+\b#Matches words starting with A

Properties of Regular Expressions

What is a Regular Expression?

Regular Expression is a way of representing regular languages. The algebraic description for regular languages is done using regular expressions. They can define it in the same language that various forms of finite automata can describe. Regular expressions offer something that finite automata do not, i.e. it is a declarative way to express the strings that we want to accept. They act as input for many systems. They are used for string matching in many systems(Java, python, etc.)

Example: Lexical-analyzer generators, such as Lex or Flex.

The widely used operators in regular expressions are Kleene closure(∗), concatenation(.), and Union(+).

Rules for Regular Expressions

- The set of regular expressions is defined by the following rules.
- Every letter of ∑ can be made into a regular expression, null string, ∈ itself is a regular expression.
  If r1 and r2 are regular expressions, then (r1), r1.r2, r1+r2, r1*, r1 + are also regular expressions.

Example - ∑ = {a, b} and r is a regular expression of language made using these symbols

Regular language	Regular set
∅	{ }
∈	{∈}
a*	{∈, a, aa, aaa .....}

Natural Language Processing (NLP)

Regular language	Regular set
a+ b	{a, b}
a.b	{ab}
a* + ba	{∈, a, aa, aaa,..... , ba}

Operations Performed on Regular Expressions

1. Union

The union of two regular languages, L1 and L2, which are represented using L1 ∪ L2, is also regular and which represents the set of strings that are either in L1 or L2 or both.

Example:

L1 = (1+0).(1+0) = {00 , 10, 11, 01} and L2 = {∈ , 100}
then L1 ∪ L2 = {∈, 00, 10, 11, 01, 100}.

2. Concatenation

The concatenation of two regular languages, L1 and L2, which are represented using L1.L2 is also regular and which represents the set of strings that are formed by taking any string in L1 concatenating it with any string in L2.

Example:

L1 = { 0,1 } and L2 = { 00, 11} then L1.L2 = {000, 011, 100, 111}.

3. Kleene closure

If L1 is a regular language, then the Kleene closure i.e. L1* of L1 is also regular and represents the set of those strings which are formed by taking a number of strings from L1 and the same string can be repeated any number of times and concatenating those strings.

Example:

L1 = { 0,1} = {∈, 0, 1, 00, 01, 10, 11 .....} , then L* is all strings possible with symbols 0 and 1 including a null string**.**

Algebraic Properties of Regular Expressions

Kleene closure is an unary operator and Union(+) and concatenation operator(.) are binary operators.

Natural Language Processing (NLP)

1. Closure

If r1 and r2 are regular expressions(RE), then

- r1* is a RE
- r1+r2 is a RE
- r1.r2 is a RE

2. Closure Laws

- (r*)* = r*, closing an expression that is already closed does not change the language.
- Ø* = ∈, a string formed by concatenating any number of copies of an empty string is empty itself.
- r + = r.r* = r*r, as r* = ∈ + r + rr+ rrr .... and r.r* = r+ rr + rrr .....
- r* = r*+ ∈

3. Associativity

If r1, r2, r3 are RE, then

i.) r1+ (r2+r3) = (r1+r2) +r3

- **For example** : r1 = a , r2 = b , r3 = c, then
- The resultant regular expression in LHS becomes a+(b+ c) and the regular set for the corresponding RE is {a, b, c}.
- for the RE in RHS becomes (a+ b) + c and the regular set for this RE is {a, b, c}, which is same in both cases. Therefore, the associativity property holds for union operator.

ii.) r1.(r2.r3) = (r1.r2).r3

- **For example -** r1 = a , r2 = b , r3 = c
- Then the string accepted by RE a.(b.c) is only abc.
- The string accepted by RE in RHS is (a.b).c is only abc ,which is same in both cases. Therefore, the **associativity property holds for concatenation operator.**

Associativity property does not hold for Kleene closure(*) because it is unary operator.

4. Identity

In the case of union operators,

r + Ø = Ø + r = r,

Therefore, **Ø** is the identity element for a union operator.

In the case of concatenation operator:

Natural Language Processing (NLP)

r.x = r , for x= ∈, r.∈ = r

Therefore, ∈ is the identity element for concatenation operator(.).

5. Annihilator

- If r+ x = r ⇒ r ∪ x= x , there is no annihilator for +
- In the case of a concatenation operator, r.x = x, when x = ∅, then r.∅ = ∅, therefore ∅ is the annihilator for the (.)operator. For example {a, aa, ab}.{ } = { }

6. Commutative Property

If r1, r2 are RE, then

- r1+r2 = r2+r1. For example, for r1 =a and r2 =b, then RE a+ b and b+ a are equal.
- r1.r2 ≠ r2.r1. For example, for r1 = a and r2 = b, then RE a.b is not equal to b.a.

7. Distributed Property

If r1, r2, r3 are regular expressions, then

- (r1+r2).r3 = r1.r3 + r2.r3 i.e. Right distribution
- r1.(r2+ r3) = r1.r2 + r1.r3 i.e. left distribution
- (r1.r2) +r3 ≠ (r1+r3)(r2+r3)

8. Idempotent Law

- r1 + r1 = r1 ⇒ r1 ∪ r1 = r1 , therefore the union operator satisfies idempotent property.
- r.r ≠ r ⇒ concatenation operator does not satisfy idempotent property.

9. Identities for Regular Expression

There are many identities for the regular expression. Let p, q and r are regular expressions.

- ∅ + r = r
- ∅.r= r.∅ = ∅
- ∈.r = r.∈ =r
- ∈* = ∈ and ∅* = ∈
- r + r = r
- r*.r* = r*
- r.r* = r*.r = r + .
- (r*)* = r*

Natural Language Processing (NLP)

- $\in + r.r^* = r^* = \in + r.r^*$
- $(p.q)^*.p = p.(q.p)^*$
- $(p + q)^* = (p^*.q^*)^* = (p^* + q^*)^*$
- $(p+ q).r = p.r + q.r$ and $r.(p+q) = r.p + r.q$

2. Tokenization is a process of splitting text into smaller units called tokens.

- Tokenization
- Word Tokenization
- Rule-based Tokenization
- Subword Tokenization
- Dictionary-Based Tokenization
- Whitespace Tokenization
- WordPiece Tokenization

What is tokenization?

Tokenization is a fundamental process in Natural Language Processing (NLP) that involves breaking down a stream of text into smaller units called tokens. These tokens can range from individual characters to full words or phrases, Based on how detailed it needs to be. By converting text into these manageable chunks, machines can more effectively analyze and understand human language.

Tokenization Explained

Tokenization can be likened to teaching someone a new language by starting with the alphabet, then moving on to syllables, and finally to complete words and sentences. This process allows for the dissection of text into parts that are easier for machines to process. For example, consider the sentence, "Chatbots are helpful." When tokenized by words, it becomes:

["Chatbots", "are", "helpful"]

Natural Language Processing (NLP)

If tokenized by characters, it becomes:

["C", "h", "a", "t", "b", "o", "t", "s", " ", "a", "r", "e", " ", "h", "e", "l", "p", "f", "u", "l"]

Each approach has its own advantages depending on the context and the specific NLP task at hand.

Types of Tokenization

Tokenization can be classified into several types based on how the text is segmented. Here are some types of tokenization:

1. **Word Tokenization**: This is the most common method where text is divided into individual words. It works well for languages with clear word boundaries, like English.
2. **Character Tokenization**: In this method, text is split into individual characters. This is particularly useful for languages without clear word boundaries or for tasks that require a detailed analysis, such as spelling correction.
3. **Sub-word Tokenization**: Sub-word tokenization strikes a balance between word and character tokenization by breaking down text into units that are larger than a single character but smaller than a full word.
4. **Sentence Tokenization**: Sentence tokenization is also a common technique used to make a division of paragraphs or large set of sentences into separated sentences as tokens.
5. **N-gram Tokenization**: N-gram tokenization splits words into fixed-sized chunks (size = n) of data.

Tokenization Use Cases

Tokenization is critical in numerous applications, including:

- **Information Retrieval:** Tokenization is essential for indexing and searching in systems that store and retrieve information efficiently based on words or phrases.
- **Search Engines**: Use tokenization to process and understand user queries. By breaking down a query into tokens, enhance efficiency match and return precise search results.
- **Machine Translation**: Tools like Google Translate rely on tokenization to convert sentences from one language into another. Segment and Reconstruct to preserve meaning.
- **Speech Recognition**: Voice assistants such as Siri and Alexa use tokenization to process spoken language. Command is first converted into text and then tokenized, enabling the system to understand and execute it accurately.

Natural Language Processing (NLP)

Tokenization Challenges

Despite its importance, tokenization faces several challenges:

1. **Ambiguity**: Human language is inherently ambiguous. A sentence like "I saw her duck" can have multiple interpretations depending on the tokenization and context.
2. **Languages Without Clear Boundaries**: Languages like Chinese and Japanese do not have clear word boundaries, making tokenization more complex.
3. **Special Characters**: Handling special characters such as punctuation, email addresses, and URLs can be tricky. For instance, "john.doe@email.com" could be tokenized in multiple ways and interpretations, complicating text analysis.

Advanced tokenization methods, like the BERT tokenizer, and techniques such as character or sub-word tokenization can help address these challenges.

Implementing Tokenization

Several tools and libraries are available to implement tokenization effectively:

1. NLTK (Natural Language Toolkit): A comprehensive Python library that offers word and sentence tokenization. It's suitable for a wide range of linguistic tasks.
2. SpaCy: A modern and efficient NLP library in Python, known for its speed and support for multiple languages. It is ideal for large-scale applications.
3. BERT Tokenizer: Emerging from the BERT pre-trained model, this tokenizer is context-aware and adept at handling the nuances of language, making it suitable for advanced NLP projects.

Natural Language Processing

4. Byte-Pair Encoding (BPE): An adaptive method that tokenizes based on the most frequent byte pairs in a text. It is effective for languages that combine smaller units to form meaning.

5. Sentence Piece: An unsupervised text tokenizer and de-tokenizer, particularly useful for neural network-based text generation tasks. It supports multiple languages and can tokenize text into sub-words.

How can Tokenization be used for a Rating Classifier Project

Tokenization can be used to develop a deep-learning model for classifying user reviews based on their ratings. Here's a step-by-step outline of the process:

1. **Data Cleaning**: Use NLTK's word_tokenize function to clean and tokenize the text, removing stop words and punctuation.

2. **Preprocessing**: Using the Tokenizer class from Keras, I transformed the text into sequences of tokens.

3. **Padding**: Before feeding the sequences into the model, I used padding to ensure all sequences had the same length.

4. **Model Training**: I trained a Bidirectional LSTM model on the tokenized data, achieving excellent classification results.

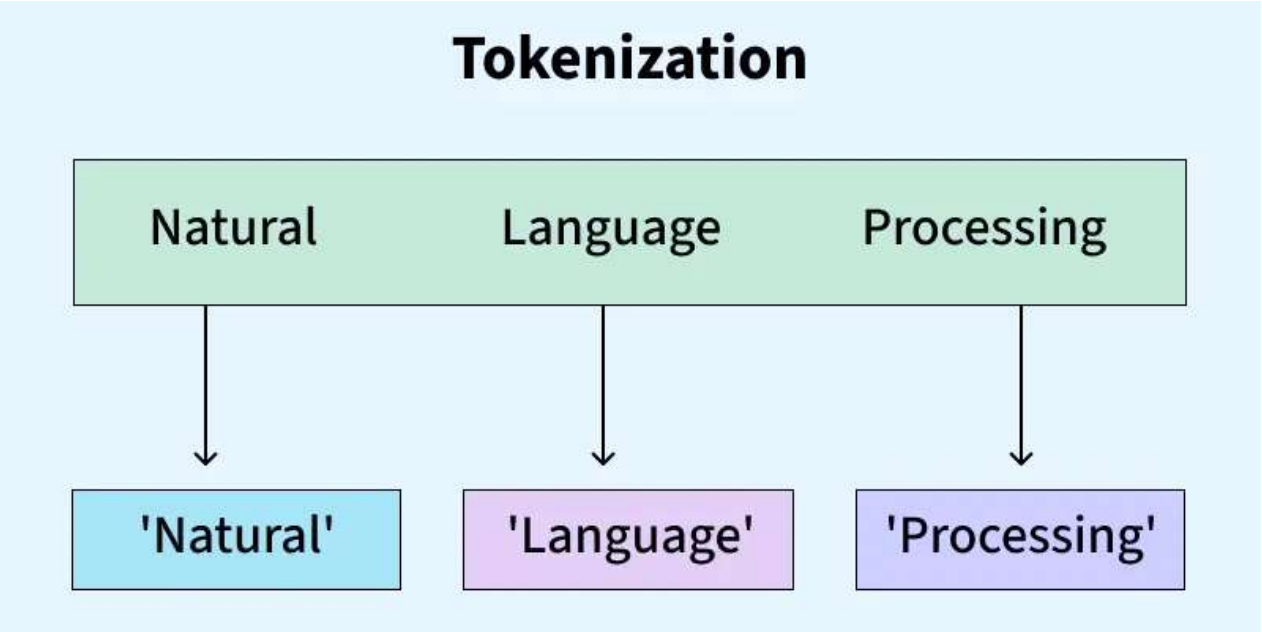
5. **Evaluation**: Finally, I evaluated the model on a testing set to ensure its effectiveness.

Python NLTK | nltk.tokenizer.word_tokenize()

Tokenization is the process of breaking text into smaller units called tokens. These may be sentences, words, sub-words or characters depending on the level of granularity we need for our NLP task. Tokens are the basic building blocks for most NLP operations, such as analysis, information extraction, sentiment assessment and more.

Natural Language Processing (NLP)

Tokenization



NLTK (Natural Language Toolkit) is a Python library that provides a range of tokenization tools including methods for splitting text into words, punctuation and even syllables. In this article we will learn about **word_tokenize** which splits a sentence or phrase into words/punctuation.

Lets a Example:

from nltk.tokenize import word_tokenize

text = "The company spent $30,000,000 last year."

tokens = word_tokenize(text)

print(tokens)

Output: *['The', 'company', 'spent', '$', '30,000,000', 'last', 'year', '.']*

nltk.tokenize.word_tokenize() tokenizes sentences into words, numbers and punctuation marks. It does not split words into syllables, but simply splits text at word boundaries.

Syntax:

from nltk.tokenize import word_tokenize

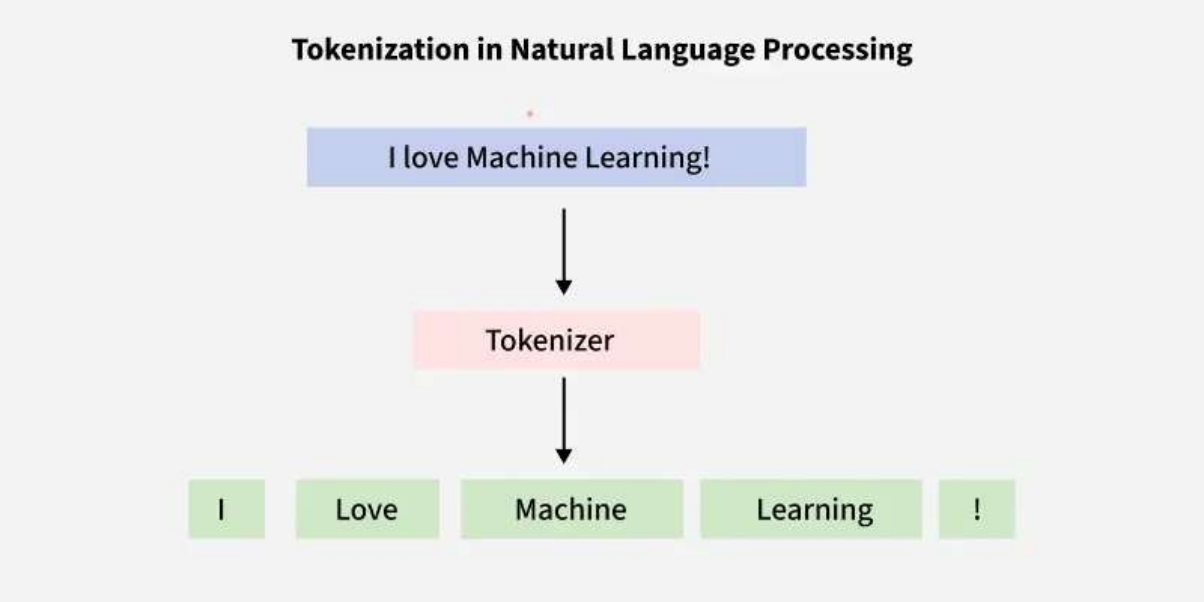
tokens = word_tokenize(text)

Rule-Based Tokenization in NLP

Natural Language Processing

Natural Language Processing (NLP) allows machines to interpret and process human language in a structured way. NLP systems uses tokenization which is a process of breaking text into smaller units called tokens. These tokens serve as the foundation for further linguistic analysis.

Tokenization example



Rule-based tokenization is a common method which has predefined rules based on whitespace, punctuation or patterns. While deep learning-based models are used in many areas, rule-based tokenization remains relevant especially in structured domains where deterministic behaviour is important.

Rule-based tokenization follows a deterministic process. It uses explicit instructions to segment input text, it often considers:

- Whitespace (spaces, tabs, newlines)
- Punctuation (commas, periods)
- Regular expressions for matching patterns
- Language-specific structures

This approach ensures consistent results and requires no training data.

1. Whitespace Tokenization

The simplest method splits text using whitespace characters. While efficient, it may leave punctuation attached to tokens.

Example:

text = "The quick brown fox jumps over the lazy dog."

Natural Language Proc

```
tokens = text.split()

print(tokens)
```

Output:

['The', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy', 'dog.']

2. Regular Expression Tokenization

Regular expressions (regex) offer flexibility for extracting structured patterns like email addresses or identifiers.

Example:

```
import re


text = "Hello, I am working at X-Y-Z and my email is ZYX@gmail.com"

pattern = r'([\w]+-[\w]+-[\w]+)|([\w\.-]+@[\w]+\.[\w]+)'

matches = re.findall(pattern, text)


for match in matches:

    print(f"Company Name: {match[0]}" if match[0] else f"Email Address: {match[1]}")
```

Output:

Company Name: X-Y-Z
Email Address: ZYX@gmail.com

This method is ideal for structured data but requires careful rule design to avoid false matching.

3. Punctuation-Based Tokenization

This method removes or uses punctuation as delimiters for splitting text. It's often used to simplify further analysis.

Example:

```
import re


text = "Hello Geeks! How can I help you?"

clean_text = re.sub(r'\W+', ' ', text)
```

Natural Language Processing (NLP)

```
tokens = re.findall(r'\b\w+\b', clean_text)

print(tokens)
```

Output:

['Hello', 'Geeks', 'How', 'can', 'I', 'help', 'you']

While useful, this method may eliminate important punctuation if not handled carefully.

4. Language-Specific Tokenization

Languages like Sanskrit, Chinese or German often require special handling due to script or grammar differences.

Example (Sanskrit):

```
from indicnlp.tokenize import indic_tokenize


# Sanskrit or Devanagari text
text = "ॐ भूर्भवः स्वः तत्सवितुर्वरेण्यं भर्गो देवस्य धीमहि धियो यो नः प्रचोदयात्।"


# Tokenize (lang code can be 'hi' as a proxy for Sanskrit)
tokens = list(indic_tokenize.trivial_tokenize(text, lang='hi'))


print(tokens)
```

Output:

['ॐ', 'भूर्भवः', ':', 'स्वः', ':', 'तत्सवितुर्वरेण्यं', 'भर्गो', 'देवस्य', 'धीमहि', 'धियो', 'यो', 'नः', ':', 'प्रचोदयात्', '।']

Language-specific models handle morphology and context better but often rely on external libraries and pre-trained data.

5. Hybrid Tokenization

In practice, combining multiple rules improves coverage. Structured patterns can be extracted using regex, followed by standard tokenization.

Example:

```
import re


text = "Contact us at support@example.com! We're open 24/7."
```

Natural Language Processing (NLP)

```
emails = re.findall(r'[\w\.-]+@[\w\.-]+\.\w+', text)

clean_text = re.sub(r'[\w\.-]+@[\w\.-]+\.\w+', '', text)

words = re.findall(r'\b\w+\b', clean_text)

tokens = emails + words

print(tokens)
```

Output:

['support@example.com', 'Contact', 'us', 'at', 'We', 're', 'open', '24', '7']

Hybrid tokenization is highly adaptable but requires thoughtful rule ordering to prevent conflicts.

6. Tokenization with NLP Libraries

Rather than building from scratch, libraries like NLTK and spaCy provide robust tokenizers that incorporate rule-based logic with language awareness.

Using NLTK:

```
from nltk.tokenize import word_tokenize, sent_tokenize


text = "Dr. Smith went to New York. He arrived at 10 a.m.!"

sentences = sent_tokenize(text)

words = word_tokenize(text)


print("Sentences:", sentences)

print("Words:", words)
```

Output:

Sentences: ['Dr. Smith went to New York.', 'He arrived at 10 a.m.!'] Words: ['Dr.', 'Smith', 'went', 'to', 'New', 'York', '.', 'He', 'arrived', 'at', '10', 'a.m.', '!']

NLTK handles common punctuation and sentence boundaries effectively with pre-defined patterns.

Using spaCy:

```
import spacy
```

Natural Language Processing (NLP)

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

text = "Visit https://www.geeksforgeeks.org/ for tutorials."

doc = nlp(text)

tokens = [token.text for token in doc]

print(tokens)
```

Output:

['Visit', 'https://www.geeksforgeeks.org/', 'for', 'tutorials', '.']

spaCy is optimized for speed and accuracy, automatically handling edge cases like URLs and contractions.

Limitations

- Whitespace and punctuation methods may leave attached symbols or split wrongly.
- Regex-based approaches can be brittle if rules are overly specific or poorly structured.
- Language-specific models may require external dependencies and setup time.
- Rule conflicts may occur in hybrid tokenization if ordering is not handled carefully.

Subword Tokenization in NLP

Natural Language Processing models often struggle to handle the wide variety of words in human language, especially within limited computing resources. Using traditional word-level tokenization seems like an ideal solution but it doesn't work well for large vocabularies or complex languages. Subword tokenization is a better solution by breaking words into smaller parts, capturing both meaning and structure more efficiently.

Understanding the Vocabulary Problem

Traditional word tokenization creates a unique token for every distinct word form. Words like "run", "running", "ran" and "runner" would each occupy separate vocabulary slots despite their semantic relationship. Multiplying this across thousands of word families, technical terms and misspellings and our vocabulary can explode to millions of unique tokens.

This vocabulary explosion creates several problems:

- **Memory overhead**: Each token requires embedding parameters making models computationally expensive

- **Out-of-vocabulary (OOV) issues**: Rare or unseen words become impossible to process
- **Poor generalization**: Related word forms are treated as completely independent entities

Subword tokenization addresses these issues by breaking words into meaningful subunits. Frequent words remain intact, while rare words are broken down into more common subword pieces that the model has likely encountered before.

Implementing Subword Tokenization

Here we will see various Subword Tokenization metods:

1. Basic Tokenization

Let's start with a practical implementation to understand the progression from word-level to subword tokenization.

- Imports regex and collections.
- Defines preprocess_text() to lowercase and tokenize text, keeping words and punctuation.
- Processes a sample paragraph using this function.
- Prints the list of tokens and the number of unique ones.

```
import re
from collections import OrderedDict, defaultdict

def preprocess_text(text):
    """
    Clean and tokenize text, handling punctuation appropriately.
    Returns a list of tokens with lowercase normalization.
    """
    # Convert to lowercase and handle punctuation
    text = text.lower()

    # Split on whitespace and punctuation, keeping punctuation as separate tokens
    tokens = re.findall(r'\w+|[^\w\s]', text)

    return tokens

# Example text processing
sample_text = """GeeksforGeeks is a fantastic resource for geeks
who are looking to enhance their programming skills,
and if you're a geek who wants to become an expert programmer,
then GeeksforGeeks is definitely the go-to place for geeks like you."""

word_tokens = preprocess_text(sample_text)
```

```
print("Word-level tokens:")
print(word_tokens)
print(f"Total unique tokens: {len(set(word_tokens))}")
```

Output:

Word-level tokens:
['geeksforgeeks', 'is', 'a', 'fantastic', 'resource', 'for', 'geeks', 'who', 'are', 'looking', 'to', 'enhance', 'their', 'programming', 'skills', ',', 'and', 'if', 'you', '"', 're', 'a', 'geek', 'who', 'wants', 'to', 'become', 'an', 'expert', 'programmer', ',', 'then', 'geeksforgeeks', 'is', 'definitely', 'the', 'go', '-', 'to', 'place', 'for', 'geeks', 'like', 'you', '.']
Total unique tokens: 35

This preprocessing step creates clean tokens while preserving punctuation as separate elements. The output shows how a short paragraph generates large number of unique tokens, highlighting the vocabulary size challenge.

2. Character-Level Tokenization

Before implementing sophisticated subword algorithms, we need to understand character-level representation. This method involves creating a frequency dictionary where each word is represented as a sequence of characters separated by spaces.

- Defines a function create_char_vocabulary() that takes a list of word tokens.
- For each word, it splits the word into characters and joins them with spaces.
- It counts how many times each unique space-separated character sequence appears.
- The vocabulary is stored as an OrderedDict, sorted by frequency.
- Prints the top 10 most frequent character sequences.

```
def create_char_vocabulary(tokens):

    char_vocab = defaultdict(int)

    for token in tokens:
        # Convert each word to space-separated characters
        char_sequence = ' '.join(list(token))
        char_vocab[char_sequence] += 1

    return OrderedDict(sorted(char_vocab.items(), key=lambda x: x[1], reverse=True))

# Create character vocabulary from our tokens
char_vocab = create_char_vocabulary(word_tokens)
```

```
print("Character-level vocabulary (top 10):")
for i, (char_seq, freq) in enumerate(char_vocab.items()):
  if i < 10:
    print(f"'{char_seq}': {freq}")
  else:
    break
```

Output:

```
Character-level vocabulary (top 10):
't o': 3
'g e e k s f o r g e e k s': 2
'is': 2
'a': 2
'f o r': 2
'g e e k s': 2
'w h o': 2
',': 2
'y o u': 2
'f a n t a s t i c': 1
```

This character-level representation serves as the foundation for **Byte-Pair Encoding**. Each word is now a sequence of individual characters and we can observe which character combinations appear most frequently across our corpus.

3. Byte-Pair Encoding Implementation

Byte-Pair Encoding works on iteratively merging the most frequent pair of symbols until reaching a desired vocabulary size. This creates a data-driven subword segmentation that balances between character granularity and word-level meaning.

- **Initial Vocabulary:** Words are split into characters with their frequencies (e.g., "l o w e r": 2).
- **Get Symbol Pairs:** get_pairs counts how often each adjacent character pair appears.
- **Merge Step:** merge_vocab replaces the most frequent pair with a combined token.
- **BPE Loop:** Repeats merging the most common pair for a set number of times (5 here), updating the vocabulary each time.
- **Final Output:** Prints the updated vocabulary after all merges, showing how characters group into subword units.

```
from collections import Counter
```

Natural Language Processing (NLP)

```
# Step 1: Input vocabulary (word -> frequency)
vocab = {
   'l o w': 5,
   'l o w e r': 2,
   'n e w e s t': 6,
   'w i d e s t': 3
}

# Step 2: Find symbol pairs
def get_pairs(vocab):
   pairs = Counter()
   for word, freq in vocab.items():
      symbols = word.split()
      for i in range(len(symbols)-1):
         pairs[(symbols[i], symbols[i+1])] += freq
   return pairs

# Step 3: Merge most frequent pair
def merge_vocab(vocab, pair):
   new_vocab = {}
   old = ' '.join(pair)
   new = ''.join(pair)
   for word, freq in vocab.items():
      new_word = word.replace(old, new)
      new_vocab[new_word] = freq
   return new_vocab

# Step 4: Run BPE
num_merges = 5
for _ in range(num_merges):
   pairs = get_pairs(vocab)
   if not pairs: break
   most_common = pairs.most_common(1)[0][0]
   vocab = merge_vocab(vocab, most_common)
   print(f"Merged: {most_common} -> {''.join(most_common)}")

# Step 5: Final vocab
print("\nFinal Vocabulary:")
for word in vocab:
   print(word)
```

Output:

Merged: ('e', 's') -> es
Merged: ('es', 't') -> est
Merged: ('l', 'o') -> lo
Merged: ('lo', 'w') -> low
Merged: ('n', 'e') -> ne
Final Vocabulary:
low low e r
ne w est
w i d est

Advantages of BPE (Byte-Pair Encoding):

- **Flexible vocabulary:** It can learn useful subword patterns specific to a domain or dataset.
- **Handles unknown words:** New or rare words can be broken into smaller known parts like characters, so the model can still understand them.
- **Efficient representation:** It keeps the vocabulary size manageable while still capturing meaningful parts of words.
- **Language-independent:** Works well with different languages and writing systems.

Limitations of BPE:

- **Dependent on training data**: If the training text doesn't represent real-world usage well, the subword splits may be poor.
- **Not dynamic**: Once trained, the BPE vocabulary doesn't learn new patterns unless retrained.
- **Inconsistent splits**: The same word might be split differently depending on context.
- **No understanding of grammar**: BPE doesn't know about grammar or word structure, it only uses frequency of character patterns.

Real-World Applications

- Subword tokenization is essential in transformer models like GPT, BERT and T5.
- GPT-2 uses Byte-Pair Encoding (BPE) on bytes, allowing it to process any Unicode text.
- BERT uses WordPiece, which selects subword units based on how likely they are to appear.
- Vocabulary size is compact (30k–50k tokens), making it efficient for memory and computation.

- It replaces huge word lists (with millions of entries) while still handling a wide variety of words.

Dictionary Based Tokenization in NLP

In Natural Language Processing (NLP), dictionary-based tokenization is the process in which the text is split into tokens using a predefined dictionary of words, phrases and expressions. This method is useful when we need to treat multi-word expressions such as names or domain-specific phrases as a single token.

For example, in Named Entity Recognition (NER) the phrase "New York" should be recognized as one token not two separate words ("New" and "York"). Dictionary-based tokenization makes this possible by referencing a predefined list of multi-word expressions. Unlike regular word tokenization which simply splits a sentence into individual words, it groups specific terms together based on the dictionary entries. This ensures that important phrases are treated as a single unit which is important for many NLP tasks.

How Does Dictionary-Based Tokenization Work?

In dictionary-based tokenization, the process of splitting the text into tokens is guided by a predefined dictionary of multi-word expressions, words and phrases. Let's see how the process typically works:

1. Input Text: We start with a string of text that needs to be tokenized. For example: *"San Francisco is a beautiful city."*

2. Dictionary Lookup: Each word or multi-word expression in the input text is checked against the dictionary. If a word or phrase matches an entry in the dictionary, it is extracted as a single token.

3. Token Matching: If the word or phrase exists in the dictionary, it is grouped as a token. For example:

"San Francisco" will be treated as a single token.

4. Handling Unmatched Words: If a word is not found in the dictionary, it is left as is or further split into smaller components. This can involve:

- Splitting a word into subwords or characters.
- Keeping it as a single token if no dictionary match is found.

For example, if a word like "NLP" is not in the dictionary, it may be broken into:

- 'N' and 'LP' or treated as a special token if predefined in the dictionary.

5. Types of Dictionaries Used:

1. **Word Lists:** These are basic dictionaries containing standard words.

2. **Subword Dictionaries:** These include common prefixes, suffixes and smaller units that help in handling rare words or out-of-vocabulary (OOV) terms.
3. **Special Tokens:** Predefined tokens for handling specific terms like numbers, punctuation or symbols (e.g "!", "$").

Steps for Implementing Dictionary-Based Tokenization

Let's see the steps required to implement Dictionary-Based Tokenization in NLP using Python and NLTK (Natural Language Toolkit).

1. Importing Required Libraries

First, we need to import the necessary libraries such as NLTK, spaCy and NumPy for handling arrays and processing data.

```
import nltk
from nltk.tokenize import word_tokenize
from nltk.tokenize import MWETokenizer
import numpy as np
```

2. Preparing the Dictionary

The key part of dictionary-based tokenization is to have a predefined dictionary that contains multi-word expressions. For example, we will create a custom dictionary containing phrases that should be treated as single tokens like place names, organizations etc.

```
custom_dict = [('San', 'Francisco'), ('New', 'York'), ('United', 'Nations')]
```

3. Preprocessing the Text

Before tokenizing, we need to clean the text. This may involve removing punctuation marks, stop words or any irrelevant characters to prepare the text for further processing.

```
def preprocess_text(text):
    text = text.lower()
    return text
sample_text = "San Francisco is a beautiful city. The United Nations meets regularly."
cleaned_text = preprocess_text(sample_text)
```

4. Tokenizing the Text

Next, we split the cleaned text into individual words using a basic tokenizer. This is where the dictionary will come into play, as multi-word expressions should remain intact.

```
tokens = word_tokenize(cleaned_text)
print("Tokenized text:", tokens)
```

Output:

Tokenized text: ['san', 'francisco', 'is', 'a', 'beautiful', 'city', '.', 'the', 'united', 'nations', 'meets', 'regularly', '.']

5. Applying Dictionary-Based Tokenization

Now, we apply the dictionary-based tokenization. The MWETokenizer (Multi-Word Expression Tokenizer) in NLTK helps in grouping multi-word expressions from the predefined dictionary into a single token.

```
tokenizer = MWETokenizer(custom_dict)

tokenized_text = tokenizer.tokenize(tokens)
print("Dictionary-based tokenized text:", tokenized_text)
```

Output:

Dictionary-based tokenized text: ['san', 'francisco', 'is', 'a', 'beautiful', 'city', '.', 'the', 'united', 'nations', 'meets', 'regularly', '.']

6. Handling Unmatched Tokens

In the tokenization process, words that are not found in the dictionary remain as individual tokens. This helps in keeping the flexibility of the process while ensuring multi-word expressions are correctly tokenized.

```
unmatched_tokens = [token for token in tokens if token not in ['San', 'Francisco', 'United', 'Nations']]
print("Unmatched Tokens:", unmatched_tokens)
```

Output:

Unmatched Tokens: ['san', 'francisco', 'is', 'a', 'beautiful', 'city', '.', 'the', 'united', 'nations', 'meets', 'regularly', '.']

7. Example of Dictionary-Based Tokenization in Action

To see dictionary-based tokenization in action, let's consider the sentence "San Francisco is part of the United Nations."

```
sentence = "San Francisco is part of the United Nations."
cleaned_sentence = preprocess_text(sentence)
tokens = word_tokenize(cleaned_sentence)

tokenized_sentence = tokenizer.tokenize(tokens)

print("Tokenized sentence:", tokenized_sentence)
```

Output:

Tokenized sentence: ['san', 'francisco', 'is', 'part', 'of', 'the', 'united', 'nations', '.']

8. Customizing the Dictionary

If we're working with domain-specific text, we can continuously expand the dictionary to include more multi-word expressions, ensuring accurate tokenization in specialized applications.

```
custom_dict.extend([('Machine', 'Learning'), ('Natural', 'Language', 'Processing')])

tokenizer = MWETokenizer(custom_dict)
```

9. Visualizing Tokenization Output

For better understanding, we can visualize how the dictionary-based tokenization is working over various sentences. This can help confirm whether multi-word expressions are accurately grouped as single tokens.

```
sentences = [
    "San Francisco is a beautiful place.",
    "The United Nations is headquartered in New York.",
    "Machine Learning is a subset of Artificial Intelligence."
]

for sentence in sentences:
    cleaned_sentence = preprocess_text(sentence)
    tokens = word_tokenize(cleaned_sentence)
    tokenized_sentence = tokenizer.tokenize(tokens)
    print(f"Original: {sentence}")
    print(f"Tokenized: {tokenized_sentence}\n")
```

Output:

Visualizing Tokenization Output

Advantages of Dictionary-Based Tokenization

1. **Handling Multi-Word Entities:** This method works great for handling complex entities like locations, names or other specific terms that should remain intact.
2. **Efficiency:** It is faster compared to more complex techniques that rely on machine learning models.
3. **Simplicity:** The approach is easy to implement and doesn't require a large amount of training data, making it a good choice for smaller projects or real-time applications.

Challenges and Limitations

1. **Out-of-Vocabulary (OOV) Words:** If a word or phrase isn't included in the dictionary, it could be split incorrectly or missed entirely.
2. **Limited Coverage:** The dictionary may not be comprehensive enough to handle all possible variations or new terms in the text.
3. **Ambiguity:** Some words might have different meanings based on context. For example, "lead" could be a noun or verb and dictionary-based tokenization might struggle to handle such ambiguities.

Python NLTK | nltk.WhitespaceTokenizer

The Natural Language Toolkit (NLTK) provides various text processing tools for Python developers. Its tokenization utilities include the WhitespaceTokenizer class which offers a simple yet effective approach to split text based on whitespace characters.

It helps in breaking text wherever whitespace occurs. This method treats spaces, tabs, newlines and other whitespace characters as natural boundaries between tokens.

Understanding NLTK's WhitespaceTokenizer

NLTK's standard tokenizer interface provides consistent methods for text processing. Unlike basic string splitting, it offers additional functionality and integrates seamlessly with other NLTK components.

Key features of WhitespaceTokenizer:

- Splits text on any whitespace character
- Handles multiple consecutive whitespace characters gracefully
- Provides span information for token positions
- Integrates with NLTK's broader text processing pipeline
- Follows consistent tokenizer interface patterns

The tokenizer works particularly well for English and other space-separated languages, making it a reliable choice for preprocessing tasks in natural language processing workflows.

Installation and Setup

To use WhitespaceTokenizer, ensure NLTK is properly installed:

```
!pip install nltk

import nltk
from nltk.tokenize import WhitespaceTokenizer
```

Basic Implementation and Usage

Getting started with WhitespaceTokenizer requires importing from NLTK's tokenize module:

```
tokenizer = WhitespaceTokenizer()

text = "The quick brown fox jumps over the lazy dog."
tokens = tokenizer.tokenize(text)
print(tokens)

messy_text = "  Hello\tworld\n\nHow    are   you?  "
clean_tokens = tokenizer.tokenize(messy_text)
print(clean_tokens)
```

Output:

['The', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy', 'dog.']
['Hello', 'world', 'How', 'are', 'you?']

Advanced Features

1. Span Tokenization

WhitespaceTokenizer provides span information through the `tokenize_sents()` and `span_tokenize()` methods:

```
tokenizer = WhitespaceTokenizer()
text = "Python NLTK is powerful. Try it today!"

spans = list(tokenizer.span_tokenize(text))
print("Token spans:")
for i, (start, end) in enumerate(spans):
    token = text[start:end]
    print(f"Token {i}: '{token}' at positions {start}-{end}")
```

Output:

Token spans:
Token 0: 'Python' at positions 0-6
Token 1: 'NLTK' at positions 7-11
Token 2: 'is' at positions 12-14
Token 3: 'powerful.' at positions 15-24
Token 4: 'Try' at positions 25-28
Token 5: 'it' at positions 29-31
Token 6: 'today!' at positions 32-38

2. Working with Multiple Sentences

The tokenizer can process multiple sentences efficiently:

```
sentences = [
    "NLTK makes text processing easy.",
    "WhitespaceTokenizer splits on whitespace.",
```

```
    "Perfect for preprocessing tasks."
]

for i, sentence in enumerate(sentences):
    tokens = tokenizer.tokenize(sentence)
    print(f"Sentence {i+1}: {tokens}")

all_spans = [list(tokenizer.span_tokenize(sent)) for sent in sentences]
```

Output:

Sentence 1: ['NLTK', 'makes', 'text', 'processing', 'easy.']
Sentence 2: ['WhitespaceTokenizer', 'splits', 'on', 'whitespace.']
Sentence 3: ['Perfect', 'for', 'preprocessing', 'tasks.']

Comparison with Built-in Methods

While Python's built-in `split()` method provides similar functionality, WhitespaceTokenizer offers several advantages:

```
text = "  Multiple\t\tspaces\n\nand\r\nlinebreaks  "

# Built-in method
builtin_tokens = text.split()
print("Built-in split():", builtin_tokens)

# NLTK WhitespaceTokenizer
tokenizer = WhitespaceTokenizer()
nltk_tokens = tokenizer.tokenize(text)
print("NLTK tokenizer:", nltk_tokens)
```

Output:

Built-in split(): ['Multiple', 'spaces', 'and', 'linebreaks']
NLTK tokenizer: ['Multiple', 'spaces', 'and', 'linebreaks']

Advantages of WhitespaceTokenizer

- Consistent interface with other NLTK tokenizers
- Built-in span tracking capabilities
- Better integration with NLTK processing pipelines
- Standardized error handling and edge case management

How WordPiece Tokenization Addresses the Rare Words Problem in NLP

One of the key challenges in Natural Language Processing is handling words that models have never seen before. Traditional methods often fail to address this effectively making them unsuitable for modern applications. There are several problems with existing methods:

- Word-level tokenization creates massive vocabularies (500,000+ tokens)
- Out-of-vocabulary(OOV) words (rare or new words) break model predictions completely
- Character-level tokenization loses semantic meaning
- Models must learn word formation from scratch which is inefficient

WordPiece tokenization offers a solution that has become the foundation of transformer models like BERT and GPT. It strikes the balance between vocabulary size and semantic preservation.

Vocabulary Explosion and Rare Words

Consider processing the sentence "The bioengineering startup developed unbreakable materials." A traditional word-level tokenizer would need separate entries for "bioengineering", "startup", "unbreakable" and "materials". If any of these words weren't in the training vocabulary, the model would fail.

Key challenges in traditional tokenization:

- Vocabulary grows exponentially with text corpus size
- Technical terms and proper nouns create endless edge cases
- Memory requirements become prohibitive for large vocabularies
- Model training becomes computationally expensive

WordPiece solves this by breaking words into meaningful subunits. Instead of treating "unbreakable" as a single unknown token, it breaks it down into recognizable pieces: ["un", "##break", "##able"]. The "##" prefix indicates that a token continues from the previous piece, preserving word boundaries and also enabling flexible decomposition.

This approach ensures that even completely new words can be understood through their constituent parts which results in improving model robustness and generalization.

How WordPiece Tokenization Works

The algorithm follows a data-driven approach to build its vocabulary. It starts with individual characters and gradually merges the most frequently occurring pairs until reaching a target vocabulary size.

Algorithm steps:

- Initialize vocabulary with all individual characters

Natural Language Processing (NLP)

- Count frequency of all adjacent symbol pairs in the corpus
- Merge the most frequent pair into a single token
- Update the corpus with the new merged token
- Repeat until reaching desired vocabulary size (typically 30K-50K tokens)

During actual tokenization, WordPiece uses a greedy longest-match strategy. For each word, it finds the longest possible subword that exists in its vocabulary then marks it as a token and repeats for the remaining characters.

Tokenization process:

- Start from the beginning of each word
- Find the longest matching subword in vocabulary
- Add it to the token list with appropriate prefix
- Move to the next unprocessed characters
- Continue until the entire word is processed

This statistical foundation ensures that common patterns naturally emerge as single tokens while rare combinations get broken into more familiar components.

Wordpiece Implementation

Let's implement basic WordPiece tokenization using the transformers library. This example focuses on core functionality without unnecessary complexity.

- Imports BertTokenizer and loads the pre-trained bert-base-uncased model which applies WordPiece tokenization and lowercases input text.
- Defines a function simple_tokenize(text) to show how BERT tokenizes input into subwords and maps them to token IDs.
- Tokenizes the full sentence, prints the resulting WordPiece tokens and converts them into their corresponding vocabulary IDs.
- Splits the input into individual words and shows how each word is broken down into subword tokens by BERT.
- Tests the function on three sample sentences to illustrate handling of rare, compound and complex words, with clear separation between examples.

```
from transformers import BertTokenizer


# Initialize the tokenizer with pre-trained BERT vocabulary
```

Natural Language Processing (NLP)

```
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')


def simple_tokenize(text):
    """
    Basic WordPiece tokenization example
    """
    print(f"Original text: {text}")

    # Convert text to WordPiece tokens
    tokens = tokenizer.tokenize(text)
    print(f"WordPiece tokens: {tokens}")

    # Convert tokens to numerical IDs
    token_ids = tokenizer.convert_tokens_to_ids(tokens)
    print(f"Token IDs: {token_ids}")

    # Show how individual words break down
    words = text.split()
    print("\nWord breakdown:")
    for word in words:
        word_tokens = tokenizer.tokenize(word)
        print(f"  '{word}' → {word_tokens}")

# Test with different examples
test_sentences = [
    "Tokenization helps handle rare words.",
    "The unbreakable smartphone survived.",
    "Bioengineering revolutionizes manufacturing."
```

Natural Language Processing (NLP)

]

```
for sentence in test_sentences:
    simple_tokenize(sentence)
    print("-" * 40)
```

Output:

```
Original text: Tokenization helps handle rare words.
WordPiece tokens: ['token', '##ization', 'helps', 'handle', 'rare', 'words', '.']
Token IDs: [19204, 3989, 7126, 5047, 4678, 2616, 1012]

Word breakdown:
  'Tokenization' → ['token', '##ization']
  'helps' → ['helps']
  'handle' → ['handle']
  'rare' → ['rare']
  'words.' → ['words', '.']
----------------------------------------
Original text: The unbreakable smartphone survived.
WordPiece tokens: ['the', 'un', '##break', '##able', 'smartphone', 'survived', '.']
Token IDs: [1996, 4895, 23890, 3085, 26381, 5175, 1012]

Word breakdown:
  'The' → ['the']
  'unbreakable' → ['un', '##break', '##able']
  'smartphone' → ['smartphone']
  'survived.' → ['survived', '.']
----------------------------------------
Original text: Bioengineering revolutionizes manufacturing.
WordPiece tokens: ['bio', '##eng', '##ine', '##ering', 'revolution', '##izes', 'manufacturing', '.']
Token IDs: [16012, 13159, 3170, 7999, 4329, 10057, 5814, 1012]

Word breakdown:
  'Bioengineering' → ['bio', '##eng', '##ine', '##ering']
  'revolutionizes' → ['revolution', '##izes']
  'manufacturing.' → ['manufacturing', '.']
```

BERT output

Vocabulary Comparison Analysis

To understand WordPiece's efficiency, let's compare it with traditional word-level tokenization using a practical example.

- The function compares WordPiece tokenization (using BERT) with basic word-level tokenization on a list of input texts.
- It collects the total and unique tokens from both methods.
- WordPiece tokens are generated using tokenizer.tokenize() while word-level tokens use text.lower().split().
- It calculates a compression ratio as the total word count divided by the total WordPiece count.

- The results show how WordPiece reduces vocabulary size by reusing subword units.
- Sample sentences are used to demonstrate and print the comparison.

```
def compare_tokenization_methods(texts):

    # Collect all unique tokens for each method
    wordpiece_tokens = set()
    word_level_tokens = set()

    total_wordpiece_count = 0
    total_word_count = 0

    for text in texts:
        # WordPiece tokenization
        wp_tokens = tokenizer.tokenize(text)
        wordpiece_tokens.update(wp_tokens)
        total_wordpiece_count += len(wp_tokens)

        # Word-level tokenization
        words = text.lower().split()
        word_level_tokens.update(words)
        total_word_count += len(words)

    return {
        'unique_wordpiece_tokens': len(wordpiece_tokens),
        'unique_word_tokens': len(word_level_tokens),
        'total_wordpiece_count': total_wordpiece_count,
        'total_word_count': total_word_count,
        'compression_ratio': total_word_count / total_wordpiece_count
```

Natural Language Processing (NLP)

```
    }

# Test with sample texts
sample_texts = [
    "Machine learning algorithms process natural language effectively.",
    "Deep neural networks revolutionize artificial intelligence applications.",
    "Transformer architectures enable unprecedented language understanding.",
    "Biomedical researchers utilize computational linguistics for analysis."
]

results = compare_tokenization_methods(sample_texts)
print("Tokenization Comparison:")
print(f"Unique WordPiece tokens needed: {results['unique_wordpiece_tokens']}")
print(f"Unique word-level tokens needed: {results['unique_word_tokens']}")
print(f"Compression ratio: {results['compression_ratio']:.2f}")
```

Output:

```
Tokenization Comparison:
Unique WordPiece tokens needed: 30
Unique word-level tokens needed: 26
Compression ratio: 0.79
```

Output

The compression ratio shows WordPiece's representational efficiency. While word-level tokenization might need 1000+ unique tokens for a technical corpus, WordPiece achieves the same coverage with 300-400 tokens.

Practical Limitations

WordPiece tokenization has limitations that we should understand before implementation.

1. Character handling issues:

- Unknown Unicode characters map to [UNK] tokens

Natural Language Processing (NLP)

- Information loss occurs with unsupported character sets
- Emoji and special symbols may not tokenize intuitively
- To resolve this ensure training data covers expected character ranges

2. Language-specific challenges:

- Struggles with languages lacking clear word boundaries (Chinese, Japanese)
- Morphologically rich languages may over-segment
- Languages that form words by combining many smaller units often produce long token sequences during processing.

3. Vocabulary limitations:

- Fixed vocabulary cannot adapt to new domains without retraining
- Specialized terminology may produce many [UNK] tokens
- Domain shift can significantly degrade performance
- Medical/legal texts often require domain-specific vocabularies

3. Lemmatization reduces words to their base or root form.

Lemmatization is an important text pre-processing technique in Natural Language Processing (NLP) that reduces words to their base form known as a "lemma." For example, the lemma of "running" is "run" and "better" becomes "good." Unlike stemming which simply removes prefixes or suffixes, it considers the word's meaning and part of speech (POS) and ensures that the base form is a valid word. This makes lemmatization more accurate as it avoids generating non-dictionary words.

Lemmatization is important for various reasons in NLP:

- **Improves accuracy:** It ensures words with similar meanings like "running" and "ran" are treated as the same.
- **Reduced Data Redundancy:** By reducing words to their base forms, it reduces redundancy in the dataset. This leads to smaller datasets which makes it easier to handle and process large amounts of text for analysis or training machine learning models.
- **Better NLP Model Performance:** By treating all similar word as same, it improves the performance of NLP models by making text more consistent. For example, treating "running," "ran" and "runs" as the same word improves the model's understanding of context and meaning.

Lemmatization Techniques

Natural Language Processing (NLP)

There are different techniques to perform lemmatization each with its own advantages and use cases:

1. Rule Based Lemmatization

In rule-based lemmatization, predefined rules are applied to a word to remove suffixes and get the root form. This approach works well for regular words but may not handle irregularities well.

For example:

Rule: *For regular verbs ending in "-ed," remove the "-ed" suffix.*

Example: "walked" -> "walk"

While this method is simple and interpretable, it doesn't account for irregular word forms like "better" which should be lemmatized to "good".

2. Dictionary-Based Lemmatization

It uses a predefined dictionary or lexicon such as WordNet to look up the base form of a word. This method is more accurate than rule-based lemmatization because it accounts for exceptions and irregular words.

For example:

- *'running' -> 'run'*
- *'better' -> 'good'*
- *'went' -> 'go*

"I was running to become a better athlete and then I went home," -> "I was run to become a good athlete and then I go home."

By using dictionaries like WordNet this method can handle a range of words effectively, especially in languages with well-established dictionaries.

3. Machine Learning-Based Lemmatization

It uses algorithms trained on large datasets to automatically identify the base form of words. This approach is highly flexible and can handle irregular words and linguistic nuances better than the rule-based and dictionary-based methods.

For example:

A trained model may deduce that “went” corresponds to “go” even though the suffix removal rule doesn’t apply. Similarly, for 'happier' the model deduces 'happy' as the lemma.

Machine learning-based lemmatizers are more adaptive and can generalize across different word forms which makes them ideal for complex tasks involving diverse vocabularies.

For more details regarding these techniques refer to: Python - Lemmatization Approaches with Examples

Implementation of Lemmatization in Python

Lets see step by step how Lemmatization works in Python:

Step 1: Installing NLTK and Downloading Necessary Resources

In Python, the NLTK library provides an easy and efficient way to implement lemmatization. First, we need to install the NLTK library and download the necessary datasets like WordNet and the punkt tokenizer.

Python

```
pip install nltk
```

Now lets import the library and download the necessary datasets.

Python

```
import nltk
nltk.download('punkt_tab')
nltk.download('wordnet')
nltk.download('omw-1.4')
nltk.download('averaged_perceptron_tagger_eng')
```

Step 2: Lemmatizing Text with NLTK

Now we can tokenize the text and apply lemmatization using NLTK's **WordNetLemmatizer**.

Python

```
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer

lemmatizer = WordNetLemmatizer()

text = "The cats were running faster than the dogs."

tokens = word_tokenize(text)

lemmatized_words = [lemmatizer.lemmatize(word) for word in tokens]

print(f"Original Text: {text}")
print(f"Lemmatized Words: {lemmatized_words}")
```

Output:

Lemmatizing Text with NLTK

In this output, we can see that:

- "cats" is reduced to its lemma "cat" (noun).
- "running" remains "running" (since no POS tag is provided, NLTK doesn't convert it to "run").

Step 3: Improving Lemmatization with Part of Speech (POS) Tagging

To improve the accuracy of lemmatization, it's important to specify the correct Part of Speech (POS) for each word. By default, NLTK assumes that words are nouns when no POS tag is provided. However, it can be more accurate if we specify the correct POS tag for each word.

For example:

- "running" (as a verb) should be lemmatized to "run".
- "better" (as an adjective) should be lemmatized to "good".

Python

```
from nltk.tokenize import word_tokenize
from nltk import pos_tag
from nltk.stem import WordNetLemmatizer

lemmatizer = WordNetLemmatizer()

sentence = "The children are running towards a better place."

tokens = word_tokenize(sentence)

tagged_tokens = pos_tag(tokens)

def get_wordnet_pos(tag):
    if tag.startswith('J'):
        return 'a'
    elif tag.startswith('V'):
        return 'v'
    elif tag.startswith('N'):
        return 'n'
    elif tag.startswith('R'):
        return 'r'
    else:
        return 'n'

lemmatized_sentence = []

for word, tag in tagged_tokens:
    if word.lower() == 'are' or word.lower() in ['is', 'am']:
        lemmatized_sentence.append(word)
    else:
        lemmatized_sentence.append(lemmatizer.lemmatize(word,
get_wordnet_pos(tag)))

print("Original Sentence: ", sentence)
print("Lemmatized Sentence: ", ' '.join(lemmatized_sentence))
```

Output:

Improving Lemmatization with POS Tagging

In this improved version:

- "children" is lemmatized to "child" (noun).
- "running" is lemmatized to "run" (verb).
- "better" is lemmatized to "good" (adjective).

Advantages of Lemmatization with NLTK

Lets see some key advantages:

1. **Efficient Data Processing:** It reduces the number of unique words by grouping similar variations together. This reduction helps to process large datasets more efficiently, conserving both memory and computational resources.
2. **Enhanced Search and Retrieval:** In tasks like search and information retrieval, it improves results by making it easier to match different forms of a word like "run," "running," "ran" to the same base form increasing the relevance of search queries.
3. **Consistency in NLP Models:** Standardizing words to their base form improves the consistency of input data which enhances the performance of NLP models. With consistent data, models are more likely to make accurate predictions and understand the underlying context of the text.

Disadvantages of Lemmatization with NLTK

1. **Time-consuming:** It can be slower compared to other techniques such as stemming because it involves parsing the text and performing dictionary lookups or morphological analysis.
2. **Not Ideal for Real-Time Applications:** Due to its time-consuming nature, it may not be well-suited for real-time applications where fast processing is important.
3. **Risk of Ambiguity:** It may sometimes produce ambiguous results, when a word has multiple meanings based on its context. For example, the word "lead" can refer to both the noun (a type of metal) and a verb (to guide). Without context, the lemmatizer might not always resolve these ambiguities correctly.

4. Stemming reduces works to their root by removing suffixes. Types of stemmers include:

- Stemming
- Porter Stemmer
- Lancaster Stemmer
- Snowball Stemmer
- Rule-based Stemming

Introduction to Stemming

Stemming is an important text-processing technique that reduces words to their base or root form by removing prefixes and suffixes. This process standardizes words which helps to improve the efficiency and effectiveness of various natural language processing (NLP) tasks.

In NLP, stemming simplifies words to their most basic form, making it easier to analyze and process text. For example, "chocolates" becomes "chocolate" and "retrieval" becomes "retrieve". This is important in the early stages of NLP tasks where words are extracted from a document and tokenized (broken into individual words).

It helps in tasks such as text classification, information retrieval and text summarization by reducing words to a base form. While it is effective, it can sometimes introduce drawbacks including potential inaccuracies and a reduction in text readability.

Note: *It's important to thoroughly understand the concept of 'tokenization' as it forms the foundational step in text preprocessing.*

Examples of stemming for the word "like":

- "likes" → "like"
- "liked" → "like"
- "likely" → "like"
- "liking" → "like"

Types of Stemmer in NLTK

Python's NLTK (Natural Language Toolkit) provides various stemming algorithms each suitable for different scenarios and languages. Lets see an overview of some of the most commonly used stemmers:

1. Porter's Stemmer

Porter's Stemmer is one of the most popular and widely used stemming algorithms. Proposed in 1980 by Martin Porter, this stemmer works by applying a series of rules to remove common suffixes from English words. It is well-known for its simplicity, speed and reliability. However, the stemmed output is not guaranteed to be a meaningful word and its applications are limited to the English language.

Example:

- 'agreed' → 'agree'
- **Rule**: If the word has a suffix **EED** (with at least one vowel and consonant) remove the suffix and change it to **EE**.

Advantages:

- Very fast and efficient.
- Commonly used for tasks like information retrieval and text mining.

Limitations:

- Outputs may not always be real words.
- Limited to English words.

Now lets implement Porter's Stemmer in Python, here we will be using NLTK library.

```
from nltk.stem import PorterStemmer

porter_stemmer = PorterStemmer()

words = ["running", "jumps", "happily", "running", "happily"]
```

```
stemmed_words = [porter_stemmer.stem(word) for word in words]

print("Original words:", words)
print("Stemmed words:", stemmed_words)
```

Output:

```
Original words: ['running', 'jumps', 'happily', 'running', 'happily']
Stemmed words: ['run', 'jump', 'happili', 'run', 'happili']
```

Porter's Stemmer

2. Snowball Stemmer

The Snowball Stemmer is an enhanced version of the Porter Stemmer which was introduced by Martin Porter as well. It is referred to as Porter2 and is faster and more aggressive than its predecessor. One of the key advantages of this is that it supports multiple languages, making it a multilingual stemmer.

Example:

- 'running' → 'run'
- 'quickly' → 'quick'

Advantages:

- More efficient than Porter Stemmer.
- Supports multiple languages.

Limitations:

- More aggressive which might lead to over-stemming.

Now lets implement Snowball Stemmer in Python, here we will be using NLTK library.

```
from nltk.stem import SnowballStemmer

stemmer = SnowballStemmer(language='english')

words_to_stem = ['running', 'jumped', 'happily', 'quickly', 'foxes']

stemmed_words = [stemmer.stem(word) for word in words_to_stem]

print("Original words:", words_to_stem)
print("Stemmed words:", stemmed_words)
```

Output:

```
Original words: ['running', 'jumped', 'happily', 'quickly', 'foxes']
Stemmed words: ['run', 'jump', 'happili', 'quick', 'fox']
```

Snowball Stemmer

3. Lancaster Stemmer

The Lancaster Stemmer is known for being more aggressive and faster than other stemmers. However, it's also more destructive and may lead to excessively shortened stems. It uses a set of external rules that are applied in an iterative manner.

Example:

- 'running' → 'run'
- 'happily' → 'happy'

Advantages:

- Very fast.
- Good for smaller datasets or quick preprocessing.

Limitations:

- Aggressive which can result in over-stemming.
- Less efficient than Snowball in larger datasets.

Now lets implement Lancaster Stemmer in Python, here we will be using NLTK library.

```
from nltk.stem import LancasterStemmer

stemmer = LancasterStemmer()

words_to_stem = ['running', 'jumped', 'happily', 'quickly', 'foxes']

stemmed_words = [stemmer.stem(word) for word in words_to_stem]

print("Original words:", words_to_stem)
print("Stemmed words:", stemmed_words)
```

Output:

```
Original words: ['running', 'jumped', 'happily', 'quickly', 'foxes']
Stemmed words: ['run', 'jump', 'happy', 'quick', 'fox']
```

Lancaster Stemmer

4. Regexp Stemmer

The Regexp Stemmer or Regular Expression Stemmer is a flexible stemming algorithm that allows users to define custom rules using regular expressions (regex). This stemmer can be helpful for very specific tasks where predefined rules are necessary for stemming.

Example:

- 'running' → 'runn'
- Custom rule: r'ing$' removes the suffix ing.

Advantages:

- Highly customizable using regular expressions.
- Suitable for domain-specific tasks.

Limitations:

- Requires manual rule definition.
- Can be computationally expensive for large datasets.

Now let's implement Regexp Stemmer in Python, here we will be using NLTK library.

```
from nltk.stem import RegexpStemmer
```

```
custom_rule = r'ing$'
regexp_stemmer = RegexpStemmer(custom_rule)

word = 'running'
stemmed_word = regexp_stemmer.stem(word)

print(f'Original Word: {word}')
print(f'Stemmed Word: {stemmed_word}')
```

Output:

```
Original Word: running
Stemmed Word: runn
```

Regexp Stemmer

5. Krovetz Stemmer

The Krovetz Stemmer was developed by Robert Krovetz in 1993. It is designed to be more linguistically accurate and tends to preserve meaning more effectively than other stemmers. It includes steps like converting plural forms to singular and removing ing from past-tense verbs.

Example:

- 'children' → 'child'
- 'running' → 'run'

Advantages:

- More accurate, as it preserves linguistic meaning.
- Works well with both singular/plural and past/present tense conversions.

Limitations:

- May be inefficient with large corpora.
- Slower compared to other stemmers.

***Note**: The Krovetz Stemmer is not natively available in the NLTK library, unlike other stemmers such as Porter, Snowball or Lancaster.*

Stemming vs. Lemmatization

Let's see the tabular difference between Stemming and Lemmatization for better understanding:

Stemming	Lemmatization
Reduces words to their root form often resulting in non-valid words.	Reduces words to their base form (lemma) ensuring a valid word.
Based on simple rules or algorithms.	Considers the word's meaning and context to return the base form.
May not always produce a valid word.	Always produces a valid word.

Stemming	Lemmatization
Example: "Better" → "bet"	Example: "Better" → "good"
No context is considered.	Considers the context and part of speech.

Applications of Stemming

Stemming plays an important role in many NLP tasks. Some of its key applications include:

1. **Information Retrieval:** It is used in search engines to improve the accuracy of search results. By reducing words to their root form, it ensures that documents with different word forms like "run," "running," "runner" are grouped together.
2. **Text Classification:** In text classification, it helps in reducing the feature space by consolidating variations of words into a single representation. This can improve the performance of machine learning algorithms.
3. **Document Clustering:** It helps in grouping similar documents by normalizing word forms, making it easier to identify patterns across large text corpora.
4. **Sentiment Analysis:** Before sentiment analysis, it is used to process reviews and comments. This allows the system to analyze sentiments based on root words which improves its ability to understand positive or negative sentiments despite word variations.

Challenges in Stemming

While stemming is beneficial but also it has some challenges:

1. **Over-Stemming:** When words are reduced too aggressively, leading to the loss of meaning. For example, "arguing" becomes "argu" making it harder to understand.
2. **Under-Stemming:** Occurs when related words are not reduced to a common base form, causing inconsistencies. For example, "argument" and "arguing" might not be stemmed similarly.
3. **Loss of Meaning**: Stemming ignores context which can result in incorrect interpretations in tasks like sentiment analysis.
4. **Choosing the Right Stemmer**: Different stemmers may produce diffierent results which requires careful selection and testing for the best fit.

These challenges can be solved by fine-tuning the stemming process or using lemmatization when necessary.

Advantages of Stemming

Stemming provides various benefits which are as follows:

1. **Text Normalization:** By reducing words to their root form, it helps to normalize text which makes it easier to analyze and process.
2. **Improved Efficiency:** It reduces the dimensionality of text data which can improve the performance of machine learning algorithms.
3. **Information Retrieval:** It enhances search engine performance by ensuring that variations of the same word are treated as the same entity.
4. **Facilitates Language Processing:** It simplifies the text by reducing variations of words which makes it easier to process and analyze large text datasets.

Porter Stemmer Technique in Natural Language Processing

It is one of the most popular stemming methods proposed in 1980 by Martin Porter . It simplifies words by reducing them to their root forms, a process known as "stemming." For example, the words "running," "runner," and "ran" can all be reduced to their root form, "run." In this article we will explore more on the Porter Stemming technique and how to perform stemming in Python.

Prerequisites: NLP Pipeline, Stemming

Implementing Porter Stemmer

You can easily implement the Porter Stemmer using Python's Natural Language Toolkit (NLTK).

```
import nltk

from nltk.stem import PorterStemmer


# Create a Porter Stemmer instance

porter_stemmer = PorterStemmer()


# Example words for stemming

words = ["running", "jumps", "happily", "programming"]


# Apply stemming to each word

stemmed_words = [porter_stemmer.stem(word) for word in words]


print("Original words:", words)
```

Natural Language Processing (NLP)

```
print("Stemmed words:", stemmed_words)
```

Output:

Original words: ['running', 'jumps', 'happily', 'programming']

Stemmed words: ['run', 'jump', 'happi', 'program']

How the Porter Stemmer Works

The **Porter Stemmer** works by applying a series of rules to remove suffixes from words in five steps. It identifies and strips common endings, reducing words to their **base forms (stems).** For example, "eating" becomes "eat" and "happily" becomes "happi." This helps in text analysis by standardizing word forms.

Key Features & Benefits of Porter Stemmer

- The algorithm takes off common endings like "-ing," "-ed," and "-ly," changing "running" to "run" and "happily" to "happi."
- The stemming process uses several steps to deal with different suffixes, making sure only the right ones are removed.
- It counts groups of consonants in a word to help decide if certain endings should be taken off.
- The Lancaster Stemmer is easy to implement and understand, making it beginner-friendly.
- It processes text quickly, which is useful for handling large amounts of data.
- It provides good results for most common English words and is widely used in NLP projects.
- By simplifying words to their base forms, it reduces the number of unique words in a dataset, making analysis easier.

Limitations of Porter Stemmer

- It can produce stems that are not meaningful, such as turning "iteration" into "iter."
- The algorithm is primarily designed for English and may not work well with other languages.
- Compared to other stemmers , it may remove suffixes more aggressively, making words more similar to each other.
- Different words may be reduced to the same stem, resulting in a loss of meaning.

Natural Language Processing (NLP)

Lancaster Stemming Technique in NLP

The Lancaster Stemmer or the Paice-Husk Stemmer, is a robust algorithm used in natural language processing to reduce words to their root forms. Developed by C.D. Paice in 1990, this algorithm aggressively applies rules to strip suffixes such as "ing" or "ed."

Prerequisites: ***NLP Pipeline*****,** ***Stemming***

Implementing Lancaster Stemming

You can easily implement the Lancaster Stemmer using Python. Here's a simple example using the 'stemming' library, which can be installed using the following command:

!pip install stemming

Now, proceed with the implementation:

```
import nltk

nltk.download('punkt_tab')


from stemming.paicehusk import stem

from nltk.tokenize import word_tokenize


text = "The cats are running swiftly."

words = word_tokenize(text)

stemmed_words = [stem(word) for word in words]


print("Original words:", words)

print("Stemmed words:", stemmed_words)
```

Output:

Original words: ['The', 'cats', 'are', 'running', 'swiftly', '.']

Stemmed words: ['Th', 'cat', 'ar', 'run', 'swiftli', '.']

How the Lancaster Stemmer Works?

The Lancaster Stemmer works by repeatedly applying a set of rules to remove endings from words until no more changes can be made. It simplifies words like "running" or "runner" into their root form, such as "run" or even "r" depending on how aggressively the algorithm applies its rules.

Natural Language Processing (NLP)

Key Features and Benefits of Lancaster Stemmer

- The Lancaster Stemmer is designed for speed, making it suitable for processing large datasets quickly.
- It reduces the diversity of word forms by consolidating various forms into a single root, enhancing the efficiency of search operations.
- Utilizing over 100 rules, it can handle complex word forms that might be overlooked by less comprehensive stemmers.
- The stemmer is straightforward to implement in programming environments, making it accessible for beginners.

Limitations of Lancaster Stemmer

- The aggressive nature of the algorithm can result in stems that are not meaningful, such as reducing "university" and "universe" to "univers."
- Primarily optimized for English, its performance may degrade with other languages.
- Due to its aggressive stemming, it can conflate words with different meanings into the same stem, leading to potential ambiguity.

Snowball Stemmer - NLP

Snowball Stemmer: It is a stemming algorithm which is also known as the Porter2 stemming algorithm as it is a better version of the Porter Stemmer since some issues of it were fixed in this stemmer.

First, let's look at what is stemming-

Stemming: It is the process of reducing the word to its word stem that affixes to suffixes and prefixes or to roots of words known as a lemma. In simple words stemming is reducing a word to its base word or stem in such a way that the words of similar kind lie under a common stem. For example - The words care, cared and caring lie under the same stem 'care'. Stemming is important in natural language processing(NLP).

Some few common **rules of Snowball stemming** are:

Few Rules:

ILY -----> ILI

LY -----> Nil

Natural Language Processing (NLP)

SS -----> SS

S -----> Nil

ED -----> E,Nil

- *Nil* means the suffix is replaced with nothing and is just removed.
- There may be cases where these rules vary depending on the words. As in the case of the suffix '*ed'* if the words are 'cared' and 'bumped' they will be stemmed as '*care*' and '*bump*'. Hence, here in cared the suffix is considered as *'d'* only and not *'ed'*. One more interesting thing is in the word '*stemmed*' it is replaced with the word '*stem*' and not '*stemmed*'. Therefore, the suffix depends on the word.

Let's see a few examples:-

Word	**Stem**
cared	care
university	univers
fairly	fair
easily	easili
singing	sing
sings	sing
sung	sung
singer	singer
sportingly	sport

Code: Python code implementation of Snowball Stemmer using NLTK library

Loading Playground...

```
import nltk
from nltk.stem.snowball import SnowballStemmer


#the stemmer requires a language parameter
snow_stemmer = SnowballStemmer(language='english')


#list of tokenized words
```

Natural Language Processing (NLP)

```
words = ['cared','university','fairly','easily','singing',
    'sings','sung','singer','sportingly']


#stem's of each word
stem_words = []
for w in words:
  x = snow_stemmer.stem(w)
  stem_words.append(x)


#print stemming results
for e1,e2 in zip(words,stem_words):
  print(e1+' ----&gt; '+e2)
```

Output:

```
cared ----> care
university ----> univers
fairly ----> fair
easily ----> easili
singing ----> sing
sings ----> sing
sung ----> sung
singer ----> singer
sportingly ----> sport
```

You can also quickly check what stem would be returned for a given word or words using the snowball site. Under its demo section, you can easily see what this algorithm does for various different words.

Other Stemming Algorithms:

- **Porter Stemmer:** This is an old stemming algorithm which was developed by Martin Porter in 1980. As compared to other algorithms it is a very gentle stemming algorithm.

Natural Language Processing (NLP)

- **Lancaster Stemmer:** It is the most aggressive stemming algorithm. We can also add our own custom rules in this algorithm when we implement this using the NLTK package. Since it's aggressive it can sometimes give strange stems as well.

There are other stemming algorithms as well.

Difference Between Porter Stemmer and Snowball Stemmer:

- Snowball Stemmer is more aggressive than Porter Stemmer.
- Some issues in Porter Stemmer were fixed in Snowball Stemmer.
- There is only a little difference in the working of these two.
- Words like '*fairly*' and '*sportingly*' were stemmed to 'fair' and 'sport' in the snowball stemmer but when you use the porter stemmer they are stemmed to '*fairli*' and '*sportingli*'.
- The difference between the two algorithms can be clearly seen in the way the word 'Sportingly' in stemmed by both. Clearly Snowball Stemmer stems it to a more accurate stem.

Drawbacks of Stemming:

- Issues of over stemming and under stemming may lead to not so meaningful or inappropriate stems.
- Stemming does not consider how the word is being used. For example - the word '*saw*' will be stemmed to '*saw*' itself but it won't be considered whether the word is being used as a noun or a verb in the context. For this reason, Lemmatization is used as it keeps this fact in consideration and will return either 'see' or 'saw' depending on whether the word 'saw' was used as a verb or a noun.

Rule-based Stemming in Natural Language Processing

Rule-based stemming is a technique in natural language processing (NLP) that reduces words to their root forms by applying specific rules for removing suffixes and prefixes. This method relies on a predefined set of rules that dictate how words should be altered, making it a straightforward approach to stemming.

Prerequisites: NLP Pipeline, Stemming

Implementing Rule-Based Stemming Technique

Here, we are defining a simple rule-based stemmer function. The function **rule_based_stemmer** takes a word and applies predefined suffix-stripping rules to

Natural Language Processing (NLP)

stem the word, removing common English suffixes like 'ing', 'ed', 'ly', 'es', and 's'. If no rule applies, it returns the word unchanged.

Loading Playground...

```
# Define a simple rule-based stemmer
def rule_based_stemmer(word):
    # Define simple stemming rules
    rules = {
        'ing': '',
        'ed': '',
        'ly': '',
        'es': '',
        's': ''
    }

    # Apply rules
    for suffix, replacement in rules.items():
        if word.endswith(suffix):
            return word[:-len(suffix)] + replacement

    return word  # Return the original word if no rule applies

# Example words to stem
words_to_stem = ['running', 'jumped', 'happily', 'quickly', 'foxes']

# Apply the rule-based stemmer
stemmed_words = [rule_based_stemmer(word) for word in words_to_stem]

# Output the results
```

Natural Language Processing (NLP)

print("Original words:", words_to_stem)

Output:

Original words: ['running', 'jumped', 'happily', 'quickly', 'foxes']

Stemmed words: ['run', 'jump', 'happi', 'quick', 'fox']

How Rule-Based Stemming Works

Rule-based stemming operates by checking each word against a list of rules that specify which endings can be removed. The algorithm applies these rules iteratively until no more changes can be made. For example, it can transform "running" into "run" and "happily" into "happi." The process continues until the word no longer matches any suffix in the rule set.

Key Features and Benefits of Rule-Based Stemming

- It removes common endings from words like "jumping" to "jump."
- It uses a specific set of rules for stemming.
- The algorithm processes large datasets quickly.
- Rule-based stemming is simple and easy to understand, making it accessible for beginners.
- It works quickly, which is beneficial when handling large volumes of text data.
- It effectively reduces many common English words to their root forms.

Limitations of Rule-Based Stemming

- It may miss some relevant word variations.
- Maintaining extensive rules can be challenging.
- It can incorrectly stem different words or fail to reduce similar ones properly.
- It is primarily designed for English, with less effectiveness in other languages. The algorithm can produce stems that are not meaningful, such as turning "university" into "univers."

5. **Stopword removal** is a process to remove common words from the document.

Natural language processing tasks often involve filtering out commonly occurring words that provide no or very little semantic value to text analysis. These words are known as stopwords include articles, prepositions and pronouns like "the",

Natural Language Processing (NLP)

"and", "is" and "in." While they seem insignificant, proper stopword handling can dramatically impact the performance and accuracy of NLP applications.

Stop-words Impact

Consider the sentence: "The quick brown fox jumps over the lazy dog"

- **Stopwords**: "the" and "over"
- **Content words**: "quick", "brown", "fox", "jumps", "lazy", "dog"

It becomes particularly important when dealing with large text corpora where computational efficiency matters. Processing every single word including high-frequency stopwords can consume unnecessary resources and potentially skew analysis results.

When to Remove Stopwords

The decision to remove stopwords depends heavily on the specific NLP task at hand:

Tasks that benefit from stopword removal:

- Text classification and sentiment analysis
- Information retrieval and search engines
- Topic modelling and clustering
- Keyword extraction

Tasks that require preserving stopwords:

- Machine translation (maintains grammatical structure)
- Text summarization (preserves sentence coherence)
- Question-answering systems (syntactic relationships matter)
- Grammar checking and parsing

Language modeling presents an interesting middle ground where the decision depends on the specific application requirements and available computational resources.

Categories of Stopwords

Understanding different types of stopwords helps in making informed decisions:

- **Standard Stopwords**: Common function words like articles ("a", "the"), conjunctions ("and", "but") and prepositions ("in", "on")
- **Domain-Specific Stopwords**: Context-dependent terms that appear frequently in specific fields like "patient" in medical texts
- **Contextual Stopwords**: Words with extremely high frequency in particular datasets
- **Numerical Stopwords**: Digits, punctuation marks and single characters

Implementation with NLTK

NLTK provides robust support for stopword removal across 16 different languages. The implementation involves tokenization followed by filtering:

- **Setup**: Import NLTK modules and download required resources like stopwords and tokenizer data.
- **Text preprocessing**: Convert the sample sentence to lowercase and tokenize it into words.

- **Stopword removal**: Load English stopwords and filter them out from the token list.
- **Output**: Print both the original and cleaned tokens for comparison.

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

nltk.download('stopwords')
nltk.download('punkt')

# Sample text
text = "This is a sample sentence showing stopword removal."

# Get English stopwords and tokenize
stop_words = set(stopwords.words('english'))
tokens = word_tokenize(text.lower())

# Remove stopwords
filtered_tokens = [word for word in tokens if word not in stop_words]

print("Original:", tokens)
print("Filtered:", filtered_tokens)
```
Output:

Original: ['this', 'is', 'a', 'sample', 'sentence', 'showing', 'stopword', 'removal', '.']
Filtered: ['sample', 'sentence', 'showing', 'stopword', 'removal', '.']

Other Methods for Stopword Removal

Lets see various methods for stopwords removal:

1. Implementation using SpaCy

SpaCy offers a more sophisticated approach with built-in linguistic analysis:

- **Imports spaCy:** Used for natural language processing.
- **Load model:** Loads the English NLP model with tokenization and stopword detection.
- **Process text:** Converts the sentence into a `Doc` object with linguistic features.
- **Remove stopwords:** Filters out common words using `token.is_stop`.
- **Print output:** Displays non-stopword tokens like `['researchers', 'developing', 'advanced', 'algorithms']`.

Python
```
import spacy

nlp = spacy.load("en_core_web_sm")
doc = nlp("The researchers are developing advanced algorithms.")

# Filter stopwords using spaCy
filtered_words = [token.text for token in doc if not token.is_stop]
```

```
print("Filtered:", filtered_words)
```

Output:

Filtered: ['researchers', 'developing', 'advanced', 'algorithms', '.']

2. Removing stop words with Genism

We can use Genism for stopword removal:

- **Import function**: Brings in `remove_stopwords` from Gensim.
- **Define text**: A sample sentence is used.
- **Apply stopword removal**: Removes common words like "the," "a".
- **Print output**: Shows original and filtered text.

Python

```
from gensim.parsing.preprocessing import remove_stopwords

# Another sample text
new_text = "The majestic mountains provide a breathtaking view."

# Remove stopwords using Gensim
new_filtered_text = remove_stopwords(new_text)

print("Original Text:", new_text)
print("Text after Stopword Removal:", new_filtered_text)
```

Output:

Original Text: The majestic mountains provide a breathtaking view.
Text after Stopword Removal: The majestic mountains provide breathtaking view.

3. Implementation with Scikit Learn

We can use Scikit Learn for stopword removal:

- Imports necessary modules from `sklearn` and `nltk` for tokenization and stopword removal.
- Defines a sample sentence
- Tokenizes the sentence into individual words using NLTK's `word_tokenize`.
- Filters out common English stopwords from the token list.
- Prints both the original and stopword-removed versions of the text.

Python

```
from sklearn.feature_extraction.text import CountVectorizer
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize

# Another sample text
new_text = "The quick brown fox jumps over the lazy dog."

# Tokenize the new text using NLTK
new_words = word_tokenize(new_text)

# Remove stopwords using NLTK
new_filtered_words = [
```

```
    word for word in new_words if word.lower() not in
stopwords.words('english')]

# Join the filtered words to form a clean text
new_clean_text = ' '.join(new_filtered_words)

print("Original Text:", new_text)
print("Text after Stopword Removal:", new_clean_text)
```

Output:

Original Text: The quick brown fox jumps over the lazy dog. Text after Stopword Removal: quick brown fox jumps lazy dog .

Among all libraries NLTK provides best performance.

Advanced Techniques and Custom Stopwords

Real-world applications often require custom stopword lists tailored to specific domains:

- Imports Counter to count word frequencies.
- Tokenizes all texts and flattens them into one word list.
- Calculates frequency of each word.
- Adds words to custom stopwords if they exceed a set frequency threshold.
- Merges custom stopwords with NLTK's default stopword list.

Python

```
from collections import Counter

def create_custom_stopwords(texts, threshold=0.1):
    # Count word frequencies across all texts
    all_words = []
    for text in texts:
        words = word_tokenize(text.lower())
        all_words.extend(words)

    word_freq = Counter(all_words)
    total_words = len(all_words)

    # Words appearing more than threshold become stopwords
    custom_stops = {word for word, freq in word_freq.items()
                    if freq / total_words > threshold}

    return custom_stops.union(set(stopwords.words('english')))
```

This approach identifies domain-specific high-frequency words that may not appear in standard stopword lists but function as noise in particular contexts.

Edge Cases and Limitations

Stopword removal is essential in NLP but must be handled carefully. It requires normalization (e.g., handling case and contractions) and language-specific lists for multilingual text. Removing words like "not" or certain prepositions can harm tasks such as sentiment analysis or entity recognition. Over-removal may lose valuable

signals while under-removal can keep noise. Its impact varies—beneficial in classification but risky in tasks needing full semantic context.

Aspect	Details
Normalization	Handle case differences and contractions (e.g., "don't", "THE")
Language Specificity	Use stopword lists tailored to each language
Context Risk	Important words like "not" or prepositions may be needed for meaning
Signal vs. Noise	Too much removal = loss of signal or too little = extra noise
Task Sensitivity	Helps in classification but may hurt tasks needing deeper understanding

Modern deep learning approaches sometimes learn to ignore irrelevant words automatically, but traditional machine learning methods and resource-constrained applications still benefit significantly from thoughtful stopword handling.

6. Parts of Speech (POS) Tagging assigns a part of speech to each word in sentence based on definition and context.

Parts of Speech (PoS) tagging is a fundamental task in Natural Language Processing (NLP) where each word in a sentence is assigned a grammatical category such as noun, verb, adjective or adverb. This process help machines to understand the structure and meaning of sentences by identifying the roles of words and their relationships.

It plays an important role in various NLP applications including machine translation, sentiment analysis and information retrieval, by bridging the gap between human language and machine understanding.

Key Concepts in POS Tagging

1. **Parts of Speech**: These are categories like nouns, verbs, adjectives, adverbs, etc that define the role of a word in a sentence.
2. **Tagging**: The process of assigning a specific part-of-speech label to each word in a sentence.

3. **Corpus**: A large collection of text data used to train POS taggers.

Example of POS Tagging

Consider the sentence: "**The quick brown fox jumps over the lazy dog.**"

After performing POS Tagging, we get:

- *"The" is tagged as determiner (DT)*
- *"quick" is tagged as adjective (JJ)*
- *"brown" is tagged as adjective (JJ)*
- *"fox" is tagged as noun (NN)*
- *"jumps" is tagged as verb (VBZ)*
- *"over" is tagged as preposition (IN)*
- *"the" is tagged as determiner (DT)*
- *"lazy" is tagged as adjective (JJ)*
- *"dog" is tagged as noun (NN)*

(NLP)

Each word is assigned a tag based on its role in the sentence. For example, "quick" and "brown" are adjectives that describe the noun "fox."

Working of POS Tagging

Let's see various steps involved in POS tagging:

- **Tokenization**: The input text is split into individual tokens (words or subwords), this step is necessary for further analysis.
- **Preprocessing**: The text is cleaned such as converting it to lowercase and removing special characters, to improve accuracy.
- **Loading a Language Model**: Tools like NLTK or SpaCy use pre-trained language models to understand the grammatical rules of the language, these models have been trained on large datasets.
- **Linguistic Analysis**: The structure of the sentence is analyzed to understand the role of each word in context.
- **POS Tagging**: Each word is assigned a part-of-speech label based on its role in the sentence and the context provided by surrounding words.
- **Evaluation**: The results are checked for accuracy. If there are any errors or misclassifications, they are corrected.

Types of POS Tagging

There are different types and each has its strengths and use cases. Let's see few common methods:

1. Rule-Based Tagging

Rule-based POS tagging assigns POS tags based on predefined grammatical rules. These rules are crafted based on morphological features (like word endings) and syntactic context, making the approach highly interpretable and transparent.

Example:

1. Rule: Assign the POS tag "Noun" to words ending in "-tion" or "-ment".

2. Sentence: "The presentation highlighted the key achievements of the project's development."

3. Tagged Output:

- "presentation" → Noun (N)
- "highlighted" → Verb (V)
- "development" → Noun (N)

2. Transformation-Based Tagging (TBT)

TBT refines POS tags through a series of context-based transformations. Unlike statistical taggers that rely on probabilities or rule-based taggers, it starts with initial tags and improves them iteratively by applying transformation rules.

Example:

- **Text:** "The cat chased the mouse."
- **Initial Tags:** "The" – DET, "cat" – NOUN, "chased" – VERB, "the" – DET, "mouse" – NOUN
- **Rule Applied:** Change “chased” from Verb to Nounbecause it follows “the”.
- **Updated Tags:** "chased" becomes Noun.

3. Statistical POS Tagging

It uses probabilistic models to assign grammatical categories (e.g noun, verb, adjective) to words in a text. Unlike rule-based methods which rely on handcrafted rules, it learns patterns from large annotated corpora using machine learning techniques.

These models calculate the likelihood of a tag based on a word and its context, helping to resolve ambiguities and handle complex grammar. Common models include:

- Hidden Markov Models (HMMs)
- Conditional Random Fields (CRFs)

Implementing POS Tagging with NLTK

Let's see step by step process how POS Tagging works with NLTK:

Step 1: Installing Required Resources

Here we import the NLTK library and download the necessary datasets using nltk.download().

Python

```
import nltk
from nltk.tokenize import word_tokenize
from nltk import pos_tag
nltk.download('punkt_tab')
nltk.download('averaged_perceptron_tagger_eng')
```

Step 2: Applying POS Tagging

First we store the sentence and tokenize it into words using **word_tokenize(text)**. Then we apply POS tagging to the tokenized words using **pos_tag(words).** This assigns a part-of-speech to each word.

Python

```
text = "NLTK is a powerful library for natural language processing."
words = word_tokenize(text)

pos_tags = pos_tag(words)
```

Step 3: Displaying Results

Now we print the original sentence and loop through the tagged words to show each word with its POS tag.

Python

```
print("Original Text:")
print(text)

print("\nPoS Tagging Result:")
for word, pos_tag in pos_tags:
    print(f"{word}: {pos_tag}")
```

Output:

POS Tagging with NLTK

Implementing POS Tagging with SpaCy

Let's see step by step process how POS Tagging works with SpaCy:

Step 1: Installing and importing Required Resources

Here we install and import the SpaCy library and install the pre-trained English language model (en_core_web_sm).

Python

```
!pip install spacy
!python -m spacy download en_core_web_sm
import spacy
nlp = spacy.load("en_core_web_sm")
```

Step 2: Defining and Processing the Text

We store the sentence in a variable and process it with **nlp(text)** to get linguistic annotations.

Python

```
text = "SpaCy is a popular natural language processing library."
doc = nlp(text)
```

Step 3: Displaying POS Tagging

Print the original sentence then loop through the tokens and display each word with its POS tag.

Python

```
print("Original Text: ", text)
print("PoS Tagging Result:")
for token in doc:
    print(f"{token.text}: {token.pos_}")
```

Output:

POS Tagging with SpaCy

Advantages of POS tagging

1. **Text Simplification**: POS tagging can break down complex sentences into simpler parts, making them easier to understand.
2. **Improved Search**: It improves information retrieval by categorizing words, making it easier to index and search text.
3. **Named Entity Recognition (NER)**: It helps identify names, places and organizations by recognizing the role of words in a sentence.

Challenges of POS Tagging

1. **Ambiguity**: Words can have multiple meanings depending on the context which can make tagging difficult.
2. **Idiomatic Expressions**: Informal language and idioms may be hard to tag correctly.
3. **Out-of-Vocabulary Words**: Words that haven't been seen before can lead to incorrect tagging.
4. **Domain Dependence**: Models may not work well outside the specific language domain they were trained on.

Natural Language Processing Chatbots

NLP chatbots are computer programs designed to interact with users in natural language helps in seamless communication between humans and machines. By using NLP techniques, these chatbots understand, interpret and generate human language.

Natural Language Processing (NLP) chatbots are computer programs designed to interact with users in natural language, enabling seamless communication between humans and machines. These chatbots use various NLP techniques to understand, interpret, and generate human language, allowing them to comprehend user queries, extract relevant information, and provide appropriate responses. In this article, we will discuss chatbots thoroughly.

What are chatbots?

Chatbots are software applications designed to engage in conversations with users, either through text or voice interfaces, by utilizing artificial intelligence and natural language processing techniques. Rule-based chatbots operate on predefined rules and patterns, while AI-powered chatbots leverage machine learning algorithms to understand and respond to natural language input. These chatbots find widespread use across various industries, including customer service, sales, healthcare, and finance, offering businesses an efficient means to automate processes, provide instant support, and enhance user engagement. By simulating human-like interactions, chatbots enable seamless communication between users and technology, transforming the way businesses interact with their customers and users.

Types of NLP Chatbots

NLP chatbots come in various types, each designed to serve different purposes and leverage different NLP techniques. A few of them are as follows:

- **A fewRetrieval-Based Chatbots:** These chatbots choose the most appropriate answers based on their learning. Access-based chatbots use predefined responses, strategizing, target recognition, response options, content management, feedback, and more. They

also use technologies such as natural language understanding (NLU) to produce human-like responses.

- **Generative Chatbots:** Generative AI chatbots create new answers from scratch based on learning data. These bots use large language models (LLMs). The most popular LLM in 2023 is ChatGPT, but there are many different LLMs and different methods and these systems will evolve rapidly in the coming months. Chatbots use LLM to understand user input, generate responses, enable conversation to flow across multiple interactions, generate content, transform/learn, and more.
- **Hybrid Chatbots:** Hybrid chatbots are conversational AI that combines various technologies and methods to provide a versatile and effective experience. These chatbots often combine custom rules with machine learning and artificial intelligence tools. female gender. It combines proprietary and machine learning-based content to provide an overall comprehensive discussion.
- **Contextual Chatbots:** Content chatbots are virtual assistants designed to engage in human-like conversations with users, providing personalized and helpful service. Contextual Chatbots Chatbots use NLP and ML to understand the context of the conversation and respond accordingly. They can solve complex questions, learn from user interaction, and provide more human and personal information.

Working of NLP Chatbots

The workings of NLP chatbots involve several key components that enable them to understand, interpret, and generate human language in a conversational manner. Here's a simplified overview of how NLP chatbots function:

1. **Input Processing**: When a user sends a message or query to the chatbot, the input is first processed to extract relevant information. This involves tasks such as tokenization (breaking the text into words or tokens), part-of-speech tagging (identifying the grammatical components of each word), and named entity recognition (identifying important entities such as names, dates, and locations).
2. **Intent Recognition**: After processing the input, the chatbot identifies the user's intent or the purpose behind the message. This involves understanding what the user wants to accomplish, such as asking a question, making a request, or providing feedback.
3. **Dialogue Management**: Once the intent is recognized, the chatbot manages the conversation flow by keeping track of the context and history of the interaction. It determines the appropriate response based on the current dialogue state and the user's intent.
4. **Response Generation**: After understanding the user's intent and context, the chatbot generates a response. This can involve retrieving information from a knowledge base,

Natural Language Processing (NLP)

executing commands, or generating a natural language response using techniques such as text generation and natural language generation.

5. **Feedback and Learning**: NLP chatbots often incorporate machine learning algorithms to improve their performance over time. They learn from user interactions and feedback to enhance their understanding of language and the accuracy of their responses.

6. **Output Rendering**: Finally, the chatbot delivers the response back to the user in a human-readable format, such as text or speech.

Overall, the workings of NLP chatbots involve a combination of text processing, intent recognition, dialogue management, response generation, and machine learning techniques to enable natural and intuitive interactions between humans and machines.

Code Implementation of NLP Chatbot

- The provided Python code utilizes the NLTK library to create an NLP chatbot capable of engaging in conversational interactions. The chatbot instance, named 'NLPChatBot', is trained using both the built-in English language corpus and custom conversations. It incorporates logic adapters for matching the best response and handling time-related queries.
- The training process involves teaching the chatbot to understand and generate appropriate responses based on the input it receives. After training, the chatbot is deployed to interact with users in a conversational manner.
- Users can input messages, and the chatbot responds accordingly, simulating natural language conversations.
- This chatbot implementation demonstrates the fundamental steps involved in developing an NLP chatbot using the ChatterBot library, including data preprocessing, training, and interaction handling.

```
import nltk
from nltk.chat.util import Chat, reflections


# Define some simple rules-based responses
rules = [
    (r'(.*)hello(.*)', ['Hello!', 'Hi there!', 'Hey!']),
    (r'(.*)how are you(.*)', ['I am doing well, thank you!', 'I\'m great, thanks for asking!']),
```

Natural Language Processing (NLP)

```
    (r'(.*)your name(.*)', ['I am an NLP chatbot.', 'You can call me ChatBot.']),
    (r'(.*)exit(.*)', ['Goodbye!', 'Bye!', 'Take care!']),
]

# Create a chatbot instance
chatbot = Chat(rules, reflections)

# Start chatting with the bot
print("Welcome to the NLP ChatBot. Type 'exit' to end the conversation.")
while True:
    user_input = input("You: ")
    if user_input.lower() == 'exit':
        break
    response = chatbot.respond(user_input)
    print("Bot:", response)
```

Output:

Welcome to the NLP ChatBot. Type 'exit' to end the conversation.
You: hello
Bot: Hi there!

Difference Between Traditional Chatbots vs NLP Chatbots

Features	Traditional Chatbot	NLP Chatbot
How do they work?	Keyword-based	AI-based
How scalable are they?	Limited scalability	Unlimited scalability

Natural Language Processing (NLP)

Features	Traditional Chatbot	NLP Chatbot
How are they updated?	Must be trained explicitly	Learns from each interaction
How do customers interact with them?	Button-focused interaction	Text and voice
How much can them understand?	Only what it has been pre-trained to	Almost anything, thanks to Automatic Semantic Understanding

Uses of NLP Chatbots

NLP chatbots find diverse applications across various industries and domains due to their ability to understand and generate human-like language. Some common uses of NLP chatbots include:

1. **Customer Service**: NLP chatbots are widely used in customer service to provide instant assistance and support to customers. They can answer frequently asked questions, troubleshoot issues, and handle basic inquiries, thereby reducing the workload on human support agents and improving customer satisfaction.
2. **Virtual Assistants**: NLP chatbots serve as virtual assistants, helping users with tasks such as setting reminders, scheduling appointments, sending notifications, and providing personalized recommendations. Popular virtual assistants like Siri, Alexa, and Google Assistant leverage NLP to understand user commands and perform tasks accordingly.
3. **E-commerce**: NLP chatbots enhance the shopping experience by assisting users with product search, recommendations, and purchase decisions. They can answer product-related queries, provide information about order status and shipping details, and offer personalized shopping suggestions based on user preferences and browsing history.
4. **Healthcare**: In the healthcare industry, NLP chatbots are used for patient engagement, remote monitoring, and health-related consultations. They can provide medical advice, answer health-related questions, remind patients to take medication, and schedule appointments with healthcare providers.

Natural Language Processing (NLP)

5. **Education**: NLP chatbots are utilized in educational settings to facilitate learning and provide personalized tutoring. They can answer students' questions, provide explanations of complex concepts, offer study tips, and deliver interactive learning experiences through quizzes and simulations.

6. **Finance**: NLP chatbots assist users with financial tasks such as checking account balances, transferring funds, paying bills, and managing budgets. They can provide personalized financial advice, help users track expenses, and offer insights into investment opportunities based on market trends and user preferences.

7. **HR and Recruitment**: NLP chatbots streamline the recruitment process by automating tasks such as resume screening, candidate pre-screening interviews, and scheduling interviews. They can engage with job applicants, answer questions about job openings and company policies, and provide status updates on application submissions.

8. **Travel and Hospitality**: NLP chatbots enhance the travel experience by assisting users with travel planning, booking flights and accommodations, and providing destination recommendations. They can answer travel-related queries, offer local information and travel tips, and handle customer support inquiries during the trip.

Overall, NLP chatbots play a crucial role in improving efficiency, enhancing user experience, and driving innovation across various industries, offering organizations valuable tools for automation, customer engagement, and service delivery.

Challenges of NLP Chatbots

Despite their numerous benefits, NLP chatbots face several challenges that can affect their performance and usability. Some of the key challenges include:

1. **Understanding Natural Language**: Natural language is inherently complex and nuanced, making it challenging for chatbots to accurately understand user queries and intents. Ambiguity, slang, spelling errors, and colloquialisms further complicate the task of natural language understanding.

2. **Contextual Understanding**: NLP chatbots often struggle to maintain context and understand the nuances of ongoing conversations. They may misinterpret user messages or fail to recognize changes in context, leading to irrelevant or inaccurate responses.

3. **Handling Complex Queries**: Chatbots may struggle to handle complex or multi-turn queries that require deep understanding and reasoning. Tasks such as answering open-ended questions, providing explanations, and solving complex problems pose significant challenges for NLP chatbots.

4. **Data Limitations**: NLP chatbots rely heavily on training data to learn language patterns and generate responses. Limited or biased training data can result in poor

Natural Language Processing (NLP)

performance, as chatbots may fail to capture the diversity of language usage and user preferences.

5. **Personalization and User Engagement**: Delivering personalized experiences and engaging users in meaningful conversations is challenging for NLP chatbots. They may struggle to adapt to individual preferences, maintain user interest over time, and provide relevant recommendations or responses.

6. **Privacy and Security Concerns**: NLP chatbots may handle sensitive information such as personal data, financial details, and health records. Ensuring the privacy and security of user data is essential to build trust and comply with regulatory requirements.

7. **Ethical Considerations**: Chatbots can inadvertently perpetuate biases and stereotypes present in training data, leading to unfair or discriminatory outcomes. Ethical concerns such as transparency, fairness, and accountability must be addressed to mitigate potential harm to users.

8. **Integration with Other Systems**: Integrating NLP chatbots with existing systems and platforms can be challenging, particularly in complex enterprise environments. Ensuring seamless interoperability and data exchange requires careful planning and coordination.

9. **Continuous Learning and Improvement**: NLP chatbots need to continuously learn from user interactions and feedback to improve their performance over time. Implementing effective mechanisms for learning, adaptation, and feedback loop is essential for enhancing chatbot capabilities.

Addressing these challenges requires advancements in NLP techniques, robust training data, thoughtful design, and ongoing evaluation and optimization of chatbot performance. Despite the hurdles, overcoming these challenges can unlock the full potential of NLP chatbots to revolutionize human-computer interaction and drive innovation across various domains.

What is the future of NLP Chatbots?

The future of NLP chatbots holds immense promise, driven by advancements in artificial intelligence, natural language processing, and human-computer interaction. Here are some key trends and possibilities shaping the future of NLP chatbots:

1. **Advancements in AI and NLP**: Continued progress in AI and NLP technologies will lead to chatbots with enhanced capabilities for understanding, reasoning, and generating human-like responses. Deep learning techniques, such as transformer models like GPT (Generative Pre-trained Transformer), will enable chatbots to comprehend context more accurately and generate more contextually relevant responses.

Natural Language Processing (NLP)

2. **Conversational AI Assistants**: NLP chatbots will evolve into sophisticated conversational AI assistants that can handle complex tasks and engage in meaningful, context-aware conversations with users. These assistants will integrate seamlessly into various applications and platforms, offering personalized assistance and automating a wide range of tasks across different domains.

3. **Multimodal Interaction**: Future NLP chatbots will support multimodal interaction, allowing users to engage through a combination of text, voice, images, and gestures. Integrating speech recognition, computer vision, and other modalities will enable more natural and intuitive interactions, enhancing user experience and accessibility.

4. **Hyper-personalization**: NLP chatbots will leverage user data and contextual information to deliver hyper-personalized experiences tailored to individual preferences, behavior, and history. They will anticipate user needs, provide proactive recommendations, and adapt their responses dynamically based on user context and feedback.

5. **Integration with IoT and Smart Devices**: NLP chatbots will be integrated with IoT devices and smart environments, enabling users to interact with their surroundings using natural language commands. They will act as central hubs for controlling smart home devices, accessing information, and managing daily tasks, offering seamless connectivity and convenience.

6. **Domain-specific Applications**: NLP chatbots will be increasingly specialized for specific industries and use cases, offering tailored solutions and expertise in areas such as healthcare, finance, education, customer service, and more. Domain-specific chatbots will provide deeper insights, domain-specific knowledge, and industry-specific functionalities to meet the unique needs of users.

7. **Ethical AI and Responsible Design**: With growing concerns about AI ethics and bias, the future of NLP chatbots will prioritize ethical considerations, fairness, transparency, and accountability. Chatbot developers will adopt responsible AI practices and design principles to mitigate biases, ensure privacy and security, and promote inclusivity and diversity.

8. **Human-Centered Design and Collaboration**: NLP chatbots will be designed with a focus on human-centered principles, emphasizing empathy, empathy, and user empowerment. They will collaborate seamlessly with human counterparts, augmenting human capabilities, and enhancing productivity and creativity through synergistic human-computer interaction.

Text Representation and Embedding Techniques in NLP

Lets see how these techniques works in NLP.

Natural Language Processing (NLP)

Text representation Techniques

It converts textual data into numerical vectors that are processed by the following methods:

- One-Hot Encoding
- Bag of Words (BOW)
- Term Frequency-Inverse Document Frequency (TF-IDF)
- N-Gram Language Modeling with NLTK
- Latent Semantic Analysis (LSA)
- Latent Dirichlet Allocation (LDA)

Natural Language Processing (NLP)

One Hot Encoding in Machine Learning

One Hot Encoding is a ***method for converting categorical variables into a binary format.*** It creates new columns for each category where **1** means the category is present and **0** means it is not. The primary purpose of One Hot Encoding is to ensure that categorical data can be effectively used in machine learning models.

Importance of One Hot Encoding

We use one hot Encoding because:

1. **Eliminating Ordinality**: Many categorical variables have no inherent order (e.g., "Male" and "Female"). If we were to assign numerical values (e.g., Male = 0, Female = 1) the model might mistakenly interpret this as a ranking and lead to biased predictions. One Hot Encoding eliminates this risk by treating each category independently.
2. **Improving Model Performance**: By providing a more detailed representation of categorical variables. One Hot Encoding can help to improve the performance of machine learning models. It allows models to capture complex relationships within the data that might be missed if categorical variables were treated as single entities.
3. **Compatibility with Algorithms**: Many machine learning algorithms particularly based on linear regression and gradient descent which require numerical input. It ensures that categorical variables are converted into a suitable format.

How One-Hot Encoding Works: An Example

To grasp the concept better let's explore a simple example. Imagine we have a dataset with fruits their categorical values and corresponding prices. Using one-hot encoding we can transform these categorical values into numerical form. For example:

- Wherever the fruit is "Apple," the Apple column will have a value of 1 while the other fruit columns (like Mango or Orange) will contain 0.
- This pattern ensures that each categorical value gets its own column represented with binary values (1 or 0) making it usable for machine learning models.

Fruit	Categorical value of fruit	Price
apple	1	5
mango	2	10
apple	1	15
orange	3	20

The output after applying one-hot encoding on the data is given as follows,

Fruit_apple	Fruit_mango	Fruit_orange	price
1	0	0	5
0	1	0	10
1	0	0	15
0	0	1	20

Implementing One-Hot Encoding Using Python

To implement one-hot encoding in Python we can use either the **Pandas library or the Scikit-learn library** both of which provide efficient and convenient methods for this task.

1. Using Pandas

Pandas offers the **get_dummies function** which is a simple and effective way to perform one-hot encoding. This method **converts categorical variables into multiple binary columns**.

- For example the Gender column with values 'M' and 'F' becomes two binary columns: Gender_F and Gender_M.
- **drop_first=True in pandas** drops one redundant column e.g., keeps only Gender_F to avoid multicollinearity.

```
import pandas as pd

from sklearn.preprocessing import OneHotEncoder


data = {

   'Employee id': [10, 20, 15, 25, 30],

   'Gender': ['M', 'F', 'F', 'M', 'F'],

   'Remarks': ['Good', 'Nice', 'Good', 'Great', 'Nice']

}


df = pd.DataFrame(data)

print(f"Original Employee Data:\n{df}\n")

# Use pd.get_dummies() to one-hot encode the categorical columns

df_pandas_encoded = pd.get_dummies(df, columns=['Gender', 'Remarks'], drop_first=True)

print(f"One-Hot Encoded Data using Pandas:\n{df_pandas_encoded}\n")


encoder = OneHotEncoder(sparse_output=False)


one_hot_encoded = encoder.fit_transform(df[categorical_columns])


one_hot_df = pd.DataFrame(one_hot_encoded,
```

Natural Language Processing (NLP)

```
                columns=encoder.get_feature_names_out(categorical_columns))

    df_sklearn_encoded = pd.concat([df.drop(categorical_columns, axis=1), one_hot_df],
axis=1)

    print(f"One-Hot Encoded Data using Scikit-Learn:\n{df_sklearn_encoded}\n")
```

Output:

Original Employee Data:

	Employee id	Gender	Remarks
0	10	M	Good
1	20	F	Nice
2	15	F	Good
3	25	M	Great
4	30	F	Nice

One-Hot Encoded Data using Pandas:

	Employee id	Gender_M	Remarks_Great	Remarks_Nice
0	10	True	False	False
1	20	False	False	True
2	15	False	False	False
3	25	True	True	False
4	30	False	False	True

We can observe that we have *3 Remarks* and *2 Gender* columns in the data. **However you can just use *n-1* columns to define parameters if it has *n* unique labels.** For example if we only keep the *Gender_Female* column and drop the *Gender_Male* column then also we can convey the entire information as when the label is 1 it means female and when the label is 0 it means male. This way we can encode the categorical data and reduce the number of parameters as well.

2. One Hot Encoding using Scikit Learn Library

Natural Language Processing (NLP)

Scikit-learn(sklearn) is a popular machine-learning library in Python that provide numerous tools for data preprocessing. It provides a **OneHotEncoder** function that we use for encoding categorical and numerical variables into binary vectors. Using **df.select_dtypes(include=['object'])** in Scikit Learn Library:

- This selects **only the columns with categorical data** (data type object).
- In this case, ['Gender', 'Remarks'] are identified as categorical columns.

```
import pandas as pd
from sklearn.preprocessing import OneHotEncoder

data = {'Employee id': [10, 20, 15, 25, 30],
    'Gender': ['M', 'F', 'F', 'M', 'F'],
    'Remarks': ['Good', 'Nice', 'Good', 'Great', 'Nice'],
    }
df = pd.DataFrame(data)
print(f"Employee data : \n{df}")

categorical_columns = df.select_dtypes(include=['object']).columns.tolist()
encoder = OneHotEncoder(sparse_output=False)

one_hot_encoded = encoder.fit_transform(df[categorical_columns])

one_hot_df = pd.DataFrame(one_hot_encoded, columns=encoder.get_feature_names_out(categorical_columns))

df_encoded = pd.concat([df, one_hot_df], axis=1)

df_encoded = df_encoded.drop(categorical_columns, axis=1)
print(f"Encoded Employee data : \n{df_encoded}")
```

Output:

Natural Language Processing (NLP)

Employee data :

	Employee id	Gender	Remarks
0	10	M	Good
1	20	F	Nice
2	15	F	Good
3	25	M	Great
4	30	F	Nice

Encoded Employee data :

	Employee id	Gender_F	Gender_M	Remarks_Good	Remarks_Great	Remarks_Nice
0	10	0.0	1.0	1.0	0.0	0.0
1	20	1.0	0.0	0.0	0.0	1.0
2	15	1.0	0.0	1.0	0.0	0.0
3	25	0.0	1.0	0.0	1.0	0.0
4	30	1.0	0.0	0.0	0.0	1.0

Both **Pandas** and **Scikit-Learn** offer robust solutions for one-hot encoding.

- Use **Pandas get_dummies()** when you need quick and simple encoding.
- Use **Scikit-Learn OneHotEncoder** when working within a machine learning pipeline or when you need finer control over encoding behavior.

Bag of words (BoW) model in NLP

In Natural Language Processing (NLP) text data needs to be converted into numbers so that machine learning algorithms can understand it. One common method to do this is Bag of Words (BoW) model. It turns text like sentence, paragraph or document into a collection of words and counts how often each word appears but ignoring the order of the words. It does not consider the order of the words or their grammar but focuses on counting how often each word appears in the text.

This makes it useful for tasks like text classification, sentiment analysis and clustering.

Key Components of BoW

- **Vocabulary:** It is a list of all unique words from the entire dataset. Each word in the vocabulary corresponds to a feature in the model.

Natural Language Processing (NLP)

- **Document Representation:** Each document is represented as a vector where each element shows the frequency of the words from the vocabulary in that document. The frequency of each word is used as a feature for the model.

Steps to Implement the Bag of Words (BoW) Model

Lets see how to implement the BoW model using Python. Here we will be using NLTK, Heapq, Matplotlib, Word cloud, Numpy and Seaborn libraries for this implementation.

Step 1: Preprocessing the Text

Before applying the BoW model, we need to preprocess the text. This includes:

- Converting the text to lowercase
- Removing non-word characters
- Removing extra spaces

Lets consider a sample text for this implementation:

Beans. I was trying to explain to somebody as we were flying in, that's corn. That's beans. And they were very impressed at my agricultural knowledge. Please give it up for Amaury once again for that outstanding introduction. I have a bunch of good friends here today, including somebody who I served with, who is one of the finest senators in the country and we're lucky to have him, your Senator, Dick Durbin is here. I also noticed, by the way, former Governor Edgar here, who I haven't seen in a long time and somehow he has not aged and I have. And it's great to see you, Governor. I want to thank President Killeen and everybody at the U of I System for making it possible for me to be here today. And I am deeply honored at the Paul Douglas Award that is being given to me. He is somebody who set the path for so much outstanding public service here in Illinois. Now, I want to start by addressing the elephant in the room. I know people are still wondering why I didn't speak at the commencement.

```
import nltk

import re


text = """Beans. I was trying to explain to somebody as we were flying in, that's corn. That's beans. And they were very impressed at my agricultural knowledge. Please give it up for Amaury once again for that outstanding introduction. I have a bunch of good friends here today, including somebody who I served with, who is one of the finest senators in the country, and we're lucky to have him, your Senator, Dick Durbin is here. I also noticed, by the way, former Governor Edgar here, who I haven't seen in a long time, and somehow he has not aged and I have. And it's great to see you, Governor. I want to thank President Killeen and everybody at the U of I System for making it possible for me to be here today. And I am deeply honored
```

Natural Language Processing (NLP)

at the Paul Douglas Award that is being given to me. He is somebody who set the path for so much outstanding public service here in Illinois. Now, I want to start by addressing the elephant in the room. I know people are still wondering why I didn't speak at the commencement."""

```
dataset = nltk.sent_tokenize(text)


for i in range(len(dataset)):
    dataset[i] = dataset[i].lower()
    dataset[i] = re.sub(r'\W', ' ', dataset[i])
    dataset[i] = re.sub(r'\s+', ' ', dataset[i])


for i, sentence in enumerate(dataset):
    print(f"Sentence {i+1}: {sentence}")
```

We can further preprocess the text depending on the dataset and specific requirements.

Step 2: Counting Word Frequencies

In this step, we count the frequency of each word in the preprocessed text. We will store these counts in a pandas DataFrame to view them easily in a tabular format.

- We initialize a dictionary to hold our word counts.
- Then, we tokenize each sentence into words.
- For each word, we check if it exists in our dictionary. If it does, we increment its count. If it doesn't, we add it to the dictionary with a count of 1.

```
word2count = {}


for data in dataset:
    words = nltk.word_tokenize(data)
    for word in words:
        if word not in word2count:
            word2count[word] = 1
```

Natural Language Processing (NLP)

```
        else:
            word2count[word] += 1

    stop_words = set(stopwords.words('english'))

    filtered_word2count = {word: count for word, count in word2count.items() if word not in stop_words}

    word_freq_df = pd.DataFrame(list(filtered_word2count.items()), columns=['Word', 'Frequency'])

    word_freq_df = word_freq_df.sort_values(by='Frequency', ascending=False)

    print(word_freq_df)
```

Output:

Counting Word Frequencies

Step 3: Selecting the Most Frequent Words

Now that we have counted the word frequencies, we will select the top N most frequent words (e.g top 10) to be used in the BoW model. We can visualize these frequent words using a bar chart to understand the distribution of words in our dataset.

```
    import heapq
    import matplotlib.pyplot as plt

    freq_words = heapq.nlargest(10, word2count, key=word2count.get)
```

Natural Language Processing (NLP)

```
print(f"Top 10 frequent words: {freq_words}")


top_words = sorted(word2count.items(), key=lambda x: x[1], reverse=True)[:10]
words, counts = zip(*top_words)


plt.figure(figsize=(10, 6))
plt.bar(words, counts, color='skyblue')
plt.xticks(rotation=45)
plt.title('Top 10 Most Frequent Words')
plt.xlabel('Words')
plt.ylabel('Frequency')
plt.show()
```

Output:

Selecting the Most Frequent Words

Step 4: Building the Bag of Words (BoW) Model

Now we will build the Bag of Words (BoW) model. This model is represented as a binary matrix where each row corresponds to a sentence and each column represents one of the top N frequent words. A 1 in the matrix shows that the word is present in the sentence and a 0 shows its absence.

We will use a heatmap to visualize this binary matrix where green shows the presence of a word (1) and red shows its absence (0).

```
import numpy as np
import seaborn as sns


X = []


for data in dataset:
    vector = []
```

Natural Language Processing (NLP)

```
        for word in freq_words:
            if word in nltk.word_tokenize(data):
                vector.append(1)
            else:
                vector.append(0)
        X.append(vector)

X = np.asarray(X)

plt.figure(figsize=(10, 6))
sns.heatmap(X, cmap='RdYlGn', cbar=False, annot=True, fmt="d", xticklabels=freq_words, yticklabels=[f"Sentence {i+1}" for i in range(len(dataset))])

plt.title('Bag of Words Matrix')
plt.xlabel('Frequent Words')
plt.ylabel('Sentences')
plt.show()
```

Output:

Building the Bag of Words (BoW) Model

Step 5: Visualizing Word Frequencies with a Word Cloud

Finally, we can create a Word Cloud to visually represent the word frequencies. In a word cloud, the size of each word is proportional to its frequency which makes it easy to identify the most common words at a glance.

```
from wordcloud import WordCloud

wordcloud = WordCloud(width=800, height=400, background_color='white').generate_from_frequencies(word2count)

plt.figure(figsize=(10, 6))
```

Natural Language Processing (NLP)

```
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis("off")
plt.title("Word Cloud of Frequent Words")
plt.show()
```

Advantages of the Bag of Words Model

- **Simplicity**: It is easy to implement and computationally efficient.
- **Versatility**: It can be used for various NLP tasks such as text classification, sentiment analysis and document clustering.
- **Interpretability**: The resulting vectors are interpretable which makes it easy to understand which words are most important in a document.

Understanding TF-IDF (Term Frequency-Inverse Document Frequency)

TF-IDF (Term Frequency–Inverse Document Frequency) is a statistical method used in natural language processing and information retrieval to evaluate how important a word is to a document in relation to a larger collection of documents. TF-IDF combines two components:

1. Term Frequency (TF): Measures how often a word appears in a document. A higher frequency suggests greater importance. If a term appears frequently in a document, it is likely relevant to the document's content.

Term Frequency (TF)

2. Inverse Document Frequency (IDF): Reduces the weight of common words across multiple documents while increasing the weight of rare words. If a term appears in fewer documents, it is more likely to be meaningful and specific.

Natural Language Processing (NLP)

Inverse Document Frequency (IDF)

This balance allows TF-IDF to highlight terms that are both frequent within a specific document and distinctive across the text document, making it a useful tool for tasks like search ranking, text classification and keyword extraction.

Converting Text into vectors with TF-IDF

Let's take an example where we have a corpus (a collection of documents) with three documents and our goal is to calculate the TF-IDF score for specific terms in these documents.

1. **Document 1:** "The cat sat on the mat."
2. **Document 2:** "The dog played in the park."
3. **Document 3:** "Cats and dogs are great pets."

Our goal is to calculate the TF-IDF score for specific terms in these documents. Let's focus on the word **"cat"** and see how TF-IDF evaluates its importance.

Step 1: Calculate Term Frequency (TF)

For Document 1:

- The word **"**cat**"** appears 1 time.
- The total number of terms in Document 1 is 6 ("the", "cat", "sat", "on", "the", "mat").
- So, TF(cat,Document 1) = 1/6

For Document 2:

- The word **"**cat**"** does not appear.
- So, TF(cat,Document 2)=0.

For Document 3:

- The word **"**cat" appears 1 time.
- The total number of terms in Document 3 is **6** ("cats", "and", "dogs", "are", "great", "pets").
- So TF (cat,Document 3)=1/6

In Document 1 and Document 3 the word **"**cat**"** has the same TF score. This means it appears with the same relative frequency in both documents. In Document 2 the TF score is 0 because the word **"**cat**"** does not appear.

Step 2: Calculate Inverse Document Frequency (IDF)

- **Total number of documents in the corpus (D):** 3
- **Number of documents containing the term "cat":** 2 (Document 1 and Document 3).

$IDF(cat,D)=log32≈0.176IDF(cat,D)=log23≈0.176$

Step 3: Calculate TF-IDF

The TF-IDF score for "cat" is 0.029 in Document 1 and Document 3 and 0 in Document 2 that reflects both the frequency of the term in the document (TF) and its rarity across the corpus (IDF).
The TF-IDF score is the product of TF and IDF:

TF-IDF

- For Document 1: TF-IDF (cat, Document 1, D)-0.167 * 0.176 - 0.029
- For Document 2: TF-IDF(cat, Document 2, D)-0x 0.176-0
- For Document 3: TF-IDF (cat, Document 3, D)-0.167 x 0.176 ~ 0.029

Implementing TF-IDF in Python

Step 1: Import modules

We will import scikit learn for this.

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

Step 2: Collect strings from documents and create a corpus

```
d0 = 'Geeks for geeks'
d1 = 'Geeks'
d2 = 'r2j'
string = [d0, d1, d2]
```

Step 3: Get TF-IDF values

Here we are using TfidfVectorizer() from scikit learn to perform tf-idf and apply on our courpus using fit_transform.

```
tfidf = TfidfVectorizer()
result = tfidf.fit_transform(string)
```

Step 4: Display IDF values

```
print('\nidf values:')
for ele1, ele2 in zip(tfidf.get_feature_names_out(), tfidf.idf_):
    print(ele1, ':', ele2)
```

Output:

Step 5: Display TF-IDF values along with indexing

```
print('\nWord indexes:')
print(tfidf.vocabulary_)
print('\ntf-idf value:')
print(result)
print('\ntf-idf values in matrix form:')
print(result.toarray())
```

Output:

Output

The result variable consists of unique words as well as the tf-if values. It can be elaborated using the below image:

From the above image the below table can be generated:

Document	Word	Document Index	Word Index	tf-idf value
d0	for	0	0	0.549
d0	geeks	0	1	0.8355
d1	geeks	1	1	1.000
d2	r2j	2	2	1.000

Applications

1. **Document Similarity and Clustering:** By converting documents into numerical vectors TF-IDF enables comparison and grouping of related texts. This is valuable for clustering news articles, research papers or customer support tickets into meaningful categories.
2. **Text Classification:** It helps in identify patterns in text for spam filtering, sentiment analysis and topic classification.
3. **Keyword Extraction:** It ranks words by importance making it possible to automatically highlight key terms, generate document tags or create concise summaries.
4. **Recommendation Systems:** Through comparison of textual descriptions TF-IDF supports suggesting related articles, videos or products enhancing user engagement.

N-Gram Language Modelling with NLTK

Language modeling involves determining the probability of a sequence of words. It is fundamental to many Natural Language Processing (NLP) applications such as speech recognition, machine translation and spam filtering where predicting or ranking the likelihood of phrases and sentences is crucial.

N-gram

N-gram is a language modelling technique that is defined as the contiguous sequence of n items from a given sample of text or speech. The N-grams are collected from a text or speech corpus. Items can be:

- Words like “This”, “article”, “is”, “on”, “NLP” → unigrams
- Word pairs likw “This article”, “article is”, “is on”, “on NLP” → bigrams
- Triplets (trigrams) or larger combinations

N-gram Language Model

N-gram models predict the probability of a word given the previous n−1 words. For example, a trigram model uses the preceding two words to predict the next word:

Goal: Calculate $\mathrm{p(w|h)}$ $p(w|h)$, the probability that the next word is w w, given context/history h h.

Example: For the phrase: “*This article is on...*”, if we want to predict the likelihood of “*NLP*” as the next word:

$$p("NLP"|"This","article","is","on")p("NLP"|"This","article","is","on")$$

Chain Rule of Probability

The probability of a sequence of words is computed as:

$$P(w1,w2,...,wn)=\prod i=1nP(wi|w1,w2,...,wi−1)P(w1,w2,...,wn)=\prod i=1nP(wi|w1,w2,...,wi−1)$$

Markov Assumption

To reduce complexity, N-gram models assume the probability of a word depends only on the previous n−1 words.

$$P(wi|w1,...,wi−1)\approx P(wi|wi−(n−1),...,wi−1)P(wi|w1,...,wi−1)\approx P(wi|wi−(n−1),...,wi−1)$$

Evaluating Language Models

1. Entropy: Measures the uncertainty or information content in a distribution.

$$H(p)=\sum xp(x)\cdot(−log⁡f0(p(x)))H(p)=\sum xp(x)\cdot(−log(p(x)))$$

It always give non negative.

2. Cross-Entropy: Measures how well a probability distribution predicts a sample from test data.

$$H(p,q)=−\sum xp(x)log⁡f0(q(x))H(p,q)=−\sum xp(x)log(q(x))$$

Usually ≥ entropy; reflects model “surprise” at the test data.

3. Perplexity: Exponential of cross-entropy; lower values indicate a better model.

$$Perplexity(W)=\prod i=1N1P(wi|wi−1)NPerplexity(W)=N\prod i=1NP(wi|wi−1)1$$

Implementing N-Gram Language Modelling in NLTK

- **words = nltk.word_tokenize(' '.join(reuters.words()))**: tokenizes the entire Reuters corpus into words
- **tri_grams = list(trigrams(words)):** creates 3-word sequences from the tokenized words

- **model = defaultdict(lambda: defaultdict(lambda: 0)):** initializes nested dictionary for trigram counts
- **model[(w1, w2)][w3] += 1:** counts occurrences of third word w3 after (w1, w2)
- **model[w1_w2][w3] /= total_count:** converts counts to probabilities
- **return max(next_word_probs, key=next_word_probs.get):** returns the most likely next word based on highest probability

```
import nltk
from nltk import trigrams
from nltk.corpus import reuters
from collections import defaultdict

nltk.download('reuters')
nltk.download('punkt')

words = nltk.word_tokenize(' '.join(reuters.words()))
tri_grams = list(trigrams(words))

model = defaultdict(lambda: defaultdict(lambda: 0))
for w1, w2, w3 in tri_grams:
    model[(w1, w2)][w3] += 1

for w1_w2 in model:
    total_count = float(sum(model[w1_w2].values()))
    for w3 in model[w1_w2]:
        model[w1_w2][w3] /= total_count


def predict_next_word(w1, w2):
    next_word_probs = model[w1, w2]
    if next_word_probs:
        return max(next_word_probs, key=next_word_probs.get)
    else:
        return "No prediction available"


print("Next Word:", predict_next_word('the', 'stock'))
```

Output:

Next Word: of

Advantages

- **Simple and Fast:** Easy to build and fast to run for small n.
- **Interpretable:** Easy to understand and debug.
- **Good Baseline:** Useful as a starting point for many NLP tasks.

Limitations

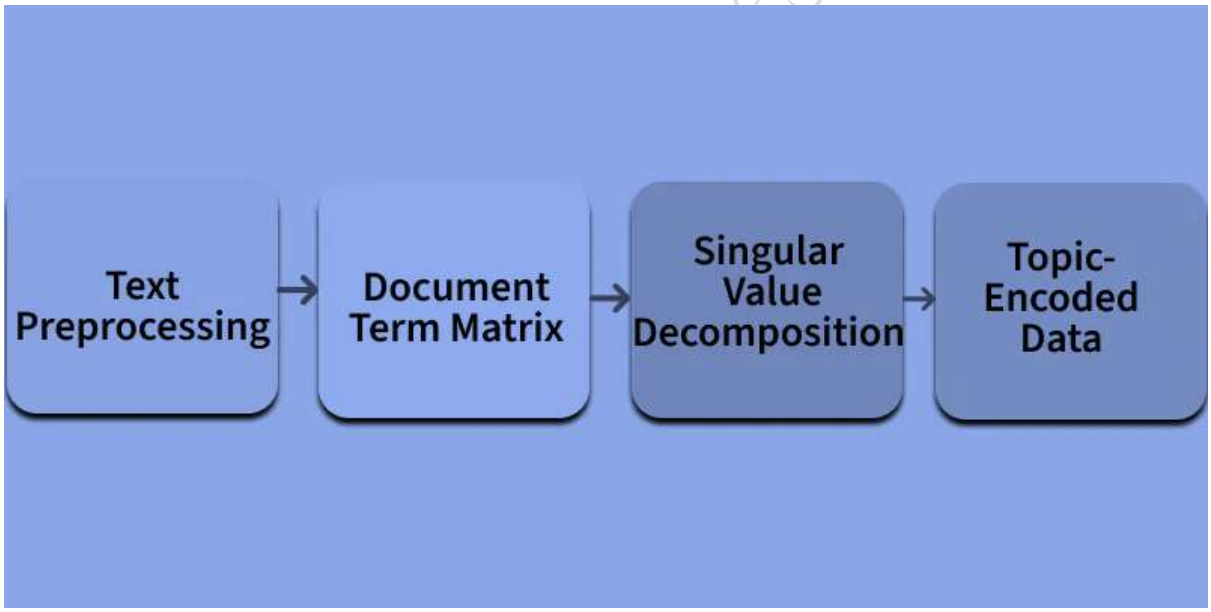
- **Limited Context:** Only considers a few previous words, missing long-range dependencies.

- **Data Sparsity:** Needs lots of data; rare n-grams are common as n increases.
- **High Memory:** Bigger n-gram models require lots of storage.
- **Poor with Unseen Words:** Struggles with new or rare words unless smoothing is applied.

Latent Semantic Analysis

Latent Semantic Analysis (LSA) is a method used to find hidden meanings in text. It looks at how words appear in different documents and discovers patterns in their usage. Instead of just counting how often words show up LSA tries to understand the context and relationship between words. It works by turning text into a big table of word counts and then using math to shrink that table down keeping only the most important parts. This helps computers group similar words and documents together based on meaning not just exact words.

Latent Semantic Analysis



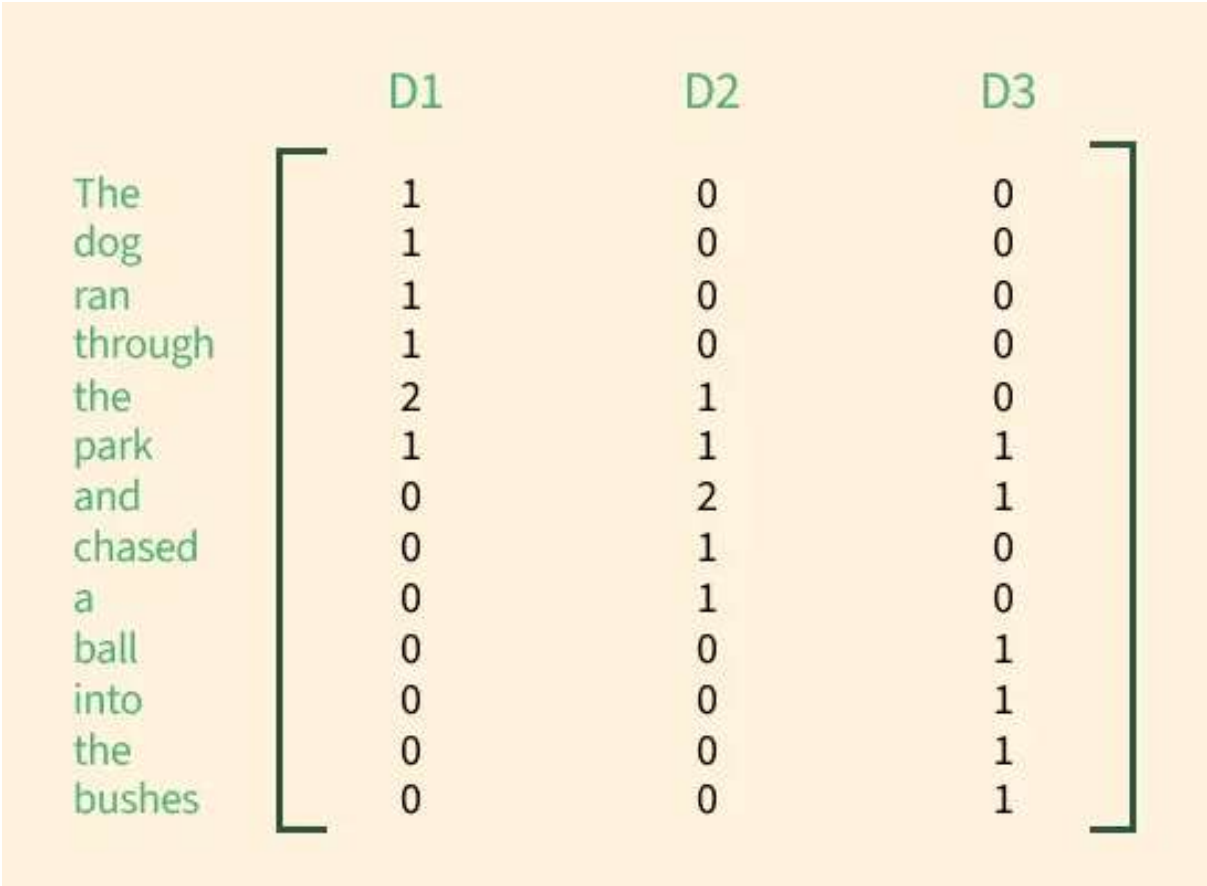
How does it Work?

Latent Semantic Analysis (LSA) works by first creating a Document term matrix showing word frequencies. It then uses Singular Value Decomposition (SVD) to reduce dimensions capturing important patterns and removing noise. This helps in identifying hidden relationships between words and documents based on meaning not just exact word matches.

1. Document term matrix

Document Term Matrix

- The first step in LSA is to create a Document Term Matrix (DTM).



- This is a table where each row represents a word each column represents a document and each cell shows how many times that word appears in that document.
- Sometimes instead of raw counts we use TF-IDF scores to give more importance to rare and meaningful words. This matrix is the foundation for analyzing patterns in word usage across documents.

2. Dimensionality Reduction

Dimensionality Reduction

- Once the DTM is created it's usually very large and sparse.
- To simplify, it applies Singular Value Decomposition (SVD) technique which breaks the matrix into three smaller matrices and we keep only the top k components that capture the most important patterns.
- This step reduces noise and focuses on the core structure of the data revealing hidden topics that link related words and documents.

$$\Sigma = \begin{bmatrix} 2.285 & 0 & 0 & 0 & 0 \\ 0 & 2.010 & 0 & 0 & 0 \\ 0 & 0 & 1.361 & 0 & 0 \\ 0 & 0 & 0 & 1.118 & 0 \\ 0 & 0 & 0 & 0 & 0.797 \end{bmatrix}$$

3. Analyse Semantic Relationships

- After dimensionality reduction each word and each document is now represented in a smaller semantic space based on the topics identified.
- Words that appear in similar contexts end up close together in this space even if they are not exactly the same.
- This helps LSA detect synonyms and understand conceptual similarity between different terms.

4. Document comparison

- Now that documents are represented in this semantic space it's easy to compare them using measures like cosine similarity.
- Documents that talk about similar topics will be close together even if they use different words.
- This makes LSA useful for tasks like clustering, ranking search results and grouping similar articles even when the vocabulary differs.

Applications

1. **Information Retrieval:** It improves search engines by matching user queries to relevant documents based on meaning not just keyword matching which helps retrieve documents even if they don't contain the exact search terms.
2. **Document Clustering and Classification:** It groups similar documents into clusters based on shared topics. This is useful in news categorization, topic discovery and automatic tagging.
3. **Plagiarism Detection:** By comparing documents semantically LSA can detect paraphrased or reworded content making it valuable for identifying plagiarism even when wording is changed.
4. **Question Answering Systems:** In QA systems, it helps match user questions to relevant answer passages by analyzing the semantic similarity between them.

Latent Dirichlet Allocation and Topic Modelling

Topic modeling is a technique in Natural Language Processing (NLP) that helps uncover hidden themes or "topics" across large sets of raw text. By recognizing patterns in how words appear together, topic models can organize documents by their underlying ideas without needing labeled data. Latent Dirichlet Allocation (LDA), the most widely applied topic modeling method, works as an unsupervised probabilistic model. It assumes that similar documents will share similar word usage and thus, will likely belong to the same topics. Each document is viewed as a mixture of topics and each topic is characterized by a distribution over words.

- Documents are expressed as probabilities over topics.
- Topics are defined as probabilities over words.

Components of Latent Dirichlet Allocation(LDA)

Probabilistic Generative Model

LDA assumes that each document is generated using a two-step random process:

- For each document, sample a distribution over topics (using a Dirichlet prior).
- For each word in the document, sample a topic from the document's topic distribution, then sample a word from the selected topic's word distribution.

Role of Dirichlet Distributions

The model uses Dirichlet distributions in two places:

- To model the diversity of topic proportions for each document (parameter α).
- To model the diversity of word proportions for each topic (parameter β).

LDA as a Mixture Model

Each document is viewed as a random mixture of topics and each topic as a mixture over words. For example, an article about sports might be a combination of topics like "teams," "games," and "scores." LDA discovers these topics based on patterns in word usage across the corpus.

Bayesian Inference in LDA

LDA uses Bayesian inference to "reverse engineer" the hidden topics from the observed words in documents. Techniques like Gibbs sampling or variational Bayes are used to estimate the latent variables:

- The topic proportions in each document.
- The word probabilities in each topic.

Key Model Parameters

- $\alpha\alpha$: Controls per-document topic diversity (high α means documents have many topics).

- $\beta\beta$: Controls per-topic word diversity (high β means topics use many different words).

Step-by-Step Implementation

Let's see the implementation of LDA topic modeling pipeline,

Step 1: Install and Import libraries

We install and import the required libraries,

- pandas: Loads, manipulates and inspects tabular data.
- numpy: Enables efficient numerical computations; sometimes useful for arrays.
- string: Helps remove punctuation during text cleaning.
- spacy: Processes text (tokenizes, tags, lemmatizes) for NLP tasks.
- nltk: Supplies English stopwords and other language tools.
- gensim: Performs topic modeling and creates bag-of-words matrices.
- matplotlib.pyplot: Creates charts and plots for data visualization.

```
!pip install --upgrade gensim pyLDAvis spacy pandas scikit-learn
import spacy.cli
spacy.cli.download("en_core_web_md")

import pandas as pd
import string
import spacy
import nltk
import gensim
from gensim import corpora
from gensim.models import CoherenceModel
import pyLDAvis.gensim_models as gensimvis
import pyLDAvis
from nltk.corpus import stopwords
import en_core_web_md
nltk.download('wordnet')
nltk.download('stopwords')
```

Step 2: Load Data

We load the dataset for operations,

- **pd.read_csv('/content/mock_yelp.csv'):** Loads Yelp-style reviews from a CSV into a pandas DataFrame.
- **print(len(yelp_review)), groupby('business_id'):** Quickly checks how many reviews, unique businesses and users are present.

```
yelp_review = pd.read_csv('/content/mock_yelp.csv')
print("Number of reviews:", len(yelp_review))
print("Unique businesses:", len(yelp_review.groupby('business_id')))
print("Unique users:", len(yelp_review.groupby('user_id')))
```

Output:

number of reviews:10
Unique Business:5
Unique User:5

Step 3: Preprocess Text

3.1 Clean text: clean_text(text): Removes punctuation and digits, lowercases text and discards short/non-informative words. Ensures input text is standardized for modeling.

```
def clean_text(text):
    delete_dict = {sp_char: '' for sp_char in string.punctuation}
    delete_dict[' '] = ' '
    table = str.maketrans(delete_dict)
    text1 = text.translate(table)
    textArr = text1.split()
    text2 = ' '.join([w for w in textArr if not w.isdigit() and len(w)
> 3])
    return text2.lower()


yelp_review['text'] = yelp_review['text'].apply(clean_text)
yelp_review['Num_words_text'] = yelp_review['text'].apply(
    lambda x: len(str(x).split()))
```

3.2 Remove Stopwards:

- **Calls to nltk.download('stopwords') and stopwords.words('english'):** Retrieves an extensive list of English stopwords.
- **remove_stopwords(text):** Filters these stopwords from reviews so only content-rich words remain.

```
stop_words = stopwords.words('english')


def remove_stopwords(text):
    textArr = text.split(' ')
    rem_text = " ".join([i for i in textArr if i not in stop_words])
    return rem_text


yelp_review['text'] = yelp_review['text'].apply(remove_stopwords)
```

3.3 Lemmatization(nouns, adjectives):

- **spacy.cli.download("en_core_web_md"):** Downloads spaCy's medium English model with vocabulary and grammatical info.
- **en_core_web_md.load(disable=['parser', 'ner']):** Loads the model for fast lemmatization, ignoring other NLP features to speed up code.
- **lemmatization(texts, allowed_postags=['NOUN', 'ADJ']):** Converts all reviews into lists of base-form words (lemmas), only keeping nouns and adjectives, which are most useful for discovering themes.

```
nlp = en_core_web_md.load(disable=['parser', 'ner'])


def lemmatization(texts, allowed_postags=['NOUN', 'ADJ']):
```

```
    output = []
    for sent in texts:
        doc = nlp(sent)
        output.append(
            [token.lemma_ for token in doc if token.pos_ in
allowed_postags])
    return output


text_list = yelp_review['text'].tolist()
tokenized_reviews = lemmatization(text_list)
```

Step 4: Create Document-Term Matrix

We create the Document-Term Matrix,

- **corpora.Dictionary(tokenized_reviews):** Creates an ID-to-word mapping from tokenized reviews.
- **[dictionary.doc2bow(rev) for rev in tokenized_reviews]:** Builds a bag-of-words matrix needed for LDA input.

```
dictionary = corpora.Dictionary(tokenized_reviews)
if len(dictionary) > 0:
    doc_term_matrix = [dictionary.doc2bow(rev) for rev in
tokenized_reviews]
else:
    doc_term_matrix = []
```

Step 5: Fit LDA Model

We prepare the LDA Model,

- Instantiates LdaModel from gensim using the corpus and dictionary.
- Parameters like num_topics, passes and iterations control how many topics to find and how thoroughly to search for them.
- print(lda_model.print_topics()): Outputs the top words and their weights for each detected topic.

```
if doc_term_matrix:
    LDA = gensim.models.ldamodel.LdaModel
    lda_model = LDA(
        corpus=doc_term_matrix,
        id2word=dictionary,
        num_topics=10,
        random_state=100,
        chunksize=1000,
        passes=50,
        iterations=100
    )
    print(lda_model.print_topics())
else:
    print("Document term matrix is empty, cannot build LDA model.")
```

Step 6: Model Evaluation

We evaluate the results of model,

- **lda_model.log_perplexity(...):** Measures how well the model fits the data (lower is better for perplexity).
- **CoherenceModel(...):** Calculates topic coherence, indicating the interpretability and meaningfulness of the topics (higher is better).

```
total_docs = len(doc_term_matrix)
if total_docs > 0:
    print('\nPerplexity:', lda_model.log_perplexity(
        doc_term_matrix, total_docs=total_docs))
    coherence_model_lda = CoherenceModel(
        model=lda_model,
        texts=tokenized_reviews,
        dictionary=dictionary,
        coherence='c_v'
    )
    coherence_lda = coherence_model_lda.get_coherence()
    print('Coherence:', coherence_lda)
else:
    print("No documents to evaluate coherence or perplexity.")
```

Output:

Perplexity: -5.0528945582253595

Coherence: 0.48202029896063986

Step 7: Visualize

- **pyLDAvis.gensim_models.prepare(...):** Prepares topic and term distributions for visualization using LDA results.
- **pyLDAvis.enable_notebook():** Ensures the visualization will display interactively in Colab/Jupyter.
- **vis_data:** Containing the topic maps and relevance charts for interactive exploration.

```
if total_docs > 0:
    pyLDAvis.enable_notebook()
    vis_data = gensimvis.prepare(lda_model, doc_term_matrix,
dictionary)
    vis_data
    pyLDAvis.save_html(vis_data, 'lda_visualization.html')
else:
    print("No documents for visualization.")
```

Output:

Visualization

The result can also be download from here.

Applications of LDA

Let's see a few applications of LDA,

- **Document Clustering:** LDA is widely used to automatically organize large collections of text (news, reviews, research papers) into thematic groups.

- **Recommendation Systems:** By identifying document/topic overlap, LDA can recommend articles, books, products or videos with similar themes.
- **Content Summarization:** LDA helps summarize corpora by surfacing prominent topics and representative terms.
- **Information Retrieval:** LDA-powered search systems can find and rank documents based on topic relevance, beyond simple keyword matches.

Advantages

- **Interpretable Output:** Each topic is a clear distribution over words; users can read topics and see representative terms.
- **Handles Large Data:** Efficient and scalable to large datasets and corpora.
- **Flexible:** Can be applied to various domains (text, genetics, images, etc.) and supports extension to dynamic, hierarchical or correlated topic models.
- **Improves Personalized Recommendations:** By modeling user preferences as distributions over topics, personalization is improved.

Natural Language Processing