#ASSIGNMENT 1

#NAME: HARSHAVARDHAN AILA

#UNT ID: 11637549

Instructions:

- Please download the provided IPython Notebook (ipynb) file and open it in Google Colab. Once opened, enter your code in the same file directly beneath the relevant question's code block.
- Insert a text block below your code to briefly explain it, mentioning any libraries or functions utilized. Answer the questions in brief with examples.
- Submit

1. The IPython Notebook (ipynb) file.
2. A PDF version of the notebook (converted from ipynb).

- The similarity score should be less than 15%

#Task 1: Tokenization (25%) (refer to the spacy tokenization concept which is explained after Question1 in activity-1)

Question -1:

##How can we incorporate contextual information beyond individual words into the tokenization process to improve performance on downstream tasks like machine translation or question answering?

Answer Here: We need to consider the following tasks to incorporate the contextual information beyond individual words into the tokenization process to improve performance downline machine translation or question answering: Subword tokenization, wordpeice tokenization, sentence peice tokenization, Byte level or character level tokenization and task-specific tokenization.

The Byte pair encoding or the unigram language model is used instead of dividing/splitting the text into individual words. The another subword tokenization approach is word piece whcih splits the words into smaller parts. It is same as BPE but differs in handling the spliiting process. It is widely used across NLP frameworks like GPT and BERT.

Another Unsupoervised text tokenizerwhich can handle several tokenization tasks including the character and subword tokenization is SentencePiece which uses the unigran language model algorithm and can be trained on large corpus which captures contextual information effectively.

The Byte-Level or Character-Level Tokenization can tokenize at the byte level or character level. This is useful when with languages that do not have a clear word boundaries or when dealing with where characters play a crucial role, example handwriting recognition.

The Task-Specific Tokenization can design task-specific tokenization approaches which takes into account the specific requirements of the task. In the example of question answering, we may want to include special tokens to mark the start and end of the answer span.

Finally, it's worth saying that tokenization is just the first step, and by using the pre-trained language models like BERT or GPT can further incorporate the contextual information during fine-tuning on downstream tasks. These models leverage transformer architectures that capture contextual dependencies between tokens through attention mechanisms, allowing them to understand the meaning of words in context.

#####**Question 2**

##Develop a tokenizer that considers contractions like "can't" and "doesn't", splitting them into their constituent words.

```
text="I can't believe it's not butter! You're going to love it."

#CODE HERE
import nltk

contraction_mapping = {
    "can't": "can not",
    "doesn't": "does not",
    "it's":"it is",
    "you're":"you are"
}

def expand_contractions(text):
#here I am Tokenizing the text using a custom regular expression
pattern:
    pattern = r"\w+(?:'\w+)?|[^\w\s]"
    tokens = nltk.regexp_tokenize(text, pattern)

#Here I am Expanding contractions:
    expanded_tokens = []
    for token in tokens:
        if token.lower() in contraction_mapping:

expanded_tokens.extend(contraction_mapping[token.lower()].split())
        else:
            expanded_tokens.append(token)

    return expanded_tokens
#Given Text:
text = "I can't believe it's not butter! You're going to love it."
expanded_tokens = expand_contractions(text)
print(expanded_tokens)
```

```
['I', 'can', 'not', 'believe', 'it', 'is', 'not', 'butter', '!',
'you', 'are', 'going', 'to', 'love', 'it', '.']
```

In the above code, I have imported the nltk library, then First using that I am trying to tokenize the custom regular expression patter. For example "can't" as short cut goes as cannot, Then I am trying to match the regular expression pattern with the contractions. In the next step i am tokenizing the text and then checking for tokens are contractions and then i am trying to expand and process . Then finally using the text it gives me the required constituent words.

Question 3:

##Implement a Python script to remove Twitter username handles from a given twitter text.

```
import nltk

def remove_username_handles(text):
#Here I am Tokenizing the text into individual words:
    tokens = nltk.word_tokenize(text)

#Here I am Removing the tokens that start with '@' for the Twitter
username handles:
    cleaned_tokens = [token for token in tokens if not
token.startswith('@')]

#Here I am Joining the cleaned tokens back into a single string:
    cleaned_text = ' '.join(cleaned_tokens)

    return cleaned_text

#Given Text
text = "Great meeting with @JohnDoe and @JaneSmith today! Looking
forward to our next project. Thanks for the insights @TechGuru 🚀
#innovation #teamwork"
cleaned_text = remove_username_handles(text)
print(cleaned_text)
```

```
Great meeting with JohnDoe and JaneSmith today ! Looking forward to
our next project . Thanks for the insights TechGuru 🚀 # innovation #
teamwork
```

Code Explanation: In the above task I have imported the nltk library and then performed tokenization for the text to split into individual words. Then I have removed the tokens which starts with '@' for the twitter username handles. Then I am joining the cleaned tokens into the single string to get back the updated sentence without the usernames. Then finally using the given text I have kept in the function and rettrived the required result.

#Task 2. - Regular Expressions (25%)

###**Regular Expressions** A regular expression, often abbreviated as regex, is a powerful and flexible tool for pattern matching and text manipulation. It consists of a sequence of characters that defines a search pattern, allowing you to perform various text-related tasks such as text validation, data extraction, text cleaning, and more. Regular expressions are used in programming languages and text editors and are constructed using a combination of regular characters, special characters, and metacharacters to specify search criteria. Learning to use

regular expressions effectively can greatly enhance text processing tasks, making them a valuable skill in fields like natural language processing, data extraction, and data validation.

Python includes a builtin module called `re` which provides regular expression matching operations (Click here for the official module documentation). Once the module is imported into your code, you can use all of the available capabilities for performing pattern-based matching or searching using regular expressions.

```
##code block -1
import re

def apply_regex(data, pattern):
  for text in data:
    if re.fullmatch(pattern, text):
      print(f"Test string {text} accepted.")
    else:
      print(f"Test string {text} failed!")
```

##Question - 1

##Same as previous question Implement a Python script to remove Twitter username handles from a given twitter text with Regular Expression

```
#CODE HERE
import re

def remove_twitterusername_handles(text):
#Here I am using the Regular expression pattern to match the Twitter
username handles:
    twitterusername_handle_pattern = re.compile(r'@\w+')

#Here I am Removing the username handles from the given text:
    updated_text = re.sub(twitterusername_handle_pattern, '', text)

    return updated_text
text="Great meeting with @JohnDoe and @JaneSmith today! Looking
forward to our next project. Thanks for the insights @TechGuru 🚀
#innovation #teamwork"
updated_text = remove_twitterusername_handles(text)
print(updated_text)
```

```
Great meeting with  and  today! Looking forward to our next project.
Thanks for the insights  🚀 #innovation #teamwork
```

Code Explanation:In the above task, I have imported the regular expression and I have tried to implement a python script to remove twitter username handles by creating a function remove_twitterusername_handles and using the regular expressions pattern I tried to match the twitter username handles. In the next step, I have tried to remove username handles from the text. Now I ahve implemented the code using the given text and achieved the usernames less sentence.

##Question- 2

##Implement a Python program to find URLs in the given string.

```
text= '<p>Contents :</p><a href="https://w3resource.com">Python
Examples</a><a href="http://github.com">Even More Examples</a><a
href="https://openai.com">OpenAI Homepage</a><a
href="https://docs.python.org">Python Documentation</a>'
##Your code here

import re
# Given Text:
text = '<p>Contents :</p><a href="https://w3resource.com">Python
Examples</a><a href="http://github.com">Even More Examples</a><a
href="https://openai.com">OpenAI Homepage</a><a
href="https://docs.python.org">Python Documentation</a>'

#Here I am matching Regular expression pattern to URLs:
pattern = r'https?://[^\s<>"]+|www\.[^\s<>"]+'

#Here I am Finding all URLs using the above pattern:
url_traced = re.findall(pattern, text)

# Printing the traced URLs:
for urls in url_traced:
    print("urls_traced:",urls)
```

```
urls_traced: https://w3resource.com
urls_traced: http://github.com
urls_traced: https://openai.com
urls_traced: https://docs.python.org
```

In the above Regular expression task, I have tried to import the regular expression as "re". Then using teh given text I have tried to match the regular expression using a pattern "r'https?://[^\s<>"]+|www.[^\s<>"]+'". The expression is all about matching/tracing the URLs using r'https?:// which explains that URls start with either http:// or https://. This part "s?:" makes sure taht there is no space betweent he URLs.The "|" acts as the OR operator. This part "www.[^\s<>"]+" gives information about tracing the word that starts with "www" followed by not white spaces.

Using regular expressions based pattern matching on real world text

For the purposes of demonstration, here's a dummy paragraph of text. A few observations here:

- The text has multiple paragraphs with each paragraph having more than one sentence.
- Some of the words are capitalized (first letter is in uppercase followed by lowercase letters).

```
text = """Here is the First Paragraph and this is the First Sentence.
here is the Second Sentence. now is the Third Sentence. this is the
```

```
Fourth Sentence of the first paragaraph. this paragraph is ending now
with a Fifth Sentence.
Now, it is the Second Paragraph and its First Sentence. here is the
Second Sentence. now is the Third Sentence. this is the Fourth
Sentence of the second paragraph. this paragraph is ending now with a
Fifth Sentence.
Finally, this is the Third Paragraph and is the First Sentence of this
paragraph. here is the Second Sentence. now is the Third Sentence.
this is the Fourth Sentence of the third paragaraph. this paragraph is
ending now with a Fifth Sentence.
4th paragraph is not going to be detected by either of the regex
patterns below.
"""

print(text)
```

```
Here is the First Paragraph and this is the First Sentence. here is
the Second Sentence. now is the Third Sentence. this is the Fourth
Sentence of the first paragaraph. this paragraph is ending now with a
Fifth Sentence.
Now, it is the Second Paragraph and its First Sentence. here is the
Second Sentence. now is the Third Sentence. this is the Fourth
Sentence of the second paragraph. this paragraph is ending now with a
Fifth Sentence.
Finally, this is the Third Paragraph and is the First Sentence of this
paragraph. here is the Second Sentence. now is the Third Sentence.
this is the Fourth Sentence of the third paragaraph. this paragraph is
ending now with a Fifth Sentence.
4th paragraph is not going to be detected by either of the regex
patterns below.
```

The following code block shows a regular expression that matches only those strings that:

1. are at the start of a line and
2. the string does not start with a number or a whitespace

`re.findall()` finds all matches of the pattern in the text under consideration. The output is a list of strings that matched.

Further, the regular expression defined below matches the words that are capitalized.

```
##code block - 3
re_pattern2 = r'[A-Z][a-z]+'
print(re.findall(re_pattern2, text))
```

```
['Here', 'First', 'Paragraph', 'First', 'Sentence', 'Second',
'Sentence', 'Third', 'Sentence', 'Fourth', 'Sentence', 'Fifth',
'Sentence', 'Now', 'Second', 'Paragraph', 'First', 'Sentence',
'Second', 'Sentence', 'Third', 'Sentence', 'Fourth', 'Sentence',
```

```
'Fifth', 'Sentence', 'Finally', 'Third', 'Paragraph', 'First',
'Sentence', 'Second', 'Sentence', 'Third', 'Sentence', 'Fourth',
'Sentence', 'Fifth', 'Sentence']
```

Following is a text excerpt on "Inaugural Address" taken from the website of the Joint Congressional Committee on Inaugural Ceremonies:

```
inau_text="""The custom of delivering an address on Inauguration Day
started with the very first Inauguration—George Washington's—on April
30, 1789(04-30-1789). ex:-18.5. After taking his oath of office on the
balcony of Federal Hall in New York City, Washington proceeded to the
Senate chamber where he read a speech before members of Congress and
other dignitaries. His second Inauguration took place in Philadelphia
on March 4, 1793(03/04/1793), in the Senate chamber of Congress Hall.
There, Washington gave the shortest Inaugural address on record—just
135 words —before repeating the oath of office.
Every President since Washington has delivered an Inaugural address.
While many of the early Presidents read their addresses before taking
the oath, current custom dictates that the Chief Justice of the
Supreme Court administer the oath first, followed by the President's
speech.
William Henry Harrison delivered the longest Inaugural address, at
8,445 words, on March 4, 1841—a bitterly cold, wet day. He died one
month later of pneumonia, believed to have been brought on by
prolonged exposure to the elements on his Inauguration Day. John
Adams' Inaugural address, which totaled 2,308 words, contained the
longest sentence, at 737 words. After Washington's second Inaugural
address, the next shortest was Franklin D. Roosevelt's fourth address
on January 20, 1945(01-20-1945), at just 559.0 words. Roosevelt had
chosen to have a simple Inauguration at the White House in light of
the nation's involvement in World War II.
In 1921, Warren G. Harding became the first President to take his oath
and deliver his Inaugural address through loud speakers. In 1925,
Calvin Coolidge's Inaugural address was the first to be broadcast
nationally by radio. And in 1949, Harry S. Truman became the first
President to deliver his Inaugural address over television airwaves.
Most Presidents use their Inaugural address to present their vision of
America and to set forth their goals for the nation. Some of the most
eloquent and powerful speeches are still quoted today. In 1865, in the
waning days of the Civil War, Abraham Lincoln stated, "With malice
toward none, with charity for all, with firmness in the right as God
gives us to see the right, let us strive on to finish the work we are
in, to bind up the nation's wounds, to care for him who shall have
borne the battle and for his widow and his orphan, to do all which may
achieve and cherish a just and lasting peace among ourselves and with
all nations." In 1933, Franklin D. Roosevelt avowed, "we have nothing
to fear but fear itself." And in 1961, John F. Kennedy declared, "And
so my fellow Americans: ask not what your country can do for you—ask
what you can do for your country."
```

```
Today, Presidents deliver their Inaugural address on the West Front of
the Capitol, but this has not always been the case. Until Andrew
Jackson’s first Inauguration in 1829, most Presidents spoke in either
the House or Senate chambers. Jackson became the first President to
take his oath of office and deliver his address on the East Front
Portico of the U.S. Capitol in 1829. With few exceptions, the next
37.0 Inaugurations took place there, until 1981, when Ronald Reagan’s
Swearing-In Ceremony and Inaugural address occurred on the West Front
Terrace of the Capitol. The West Front has been used ever since. You
should also need to extract the floating numbers such as -55.5, 20.8%,
-3.0 using your regular expression"""
```

Refer to above code block -3 for the following questions

##Questions-3.A

Identify all the positive and negative numbers with type of both intergers,float in the "Inaugural Address" excerpt and write a regular expression that finds all occurrences of such words in the text. Then, run the Python code snippet to automatically display the matched strings according to the pattern.*.

NOTE: You can use the *re.findall()* method as demonstrated in the example before this exercise.

```
##Your code here

import re

# Regular expression pattern to match positive and negative floating-
point numbers
re_pattern3 = r'-?\d+(?:\.\d+)?'

# Find all positive and negative floating-point numbers using the
pattern
numbers = re.findall(re_pattern3, inau_text)

# Printing the matched numbers:
for number in numbers:
    print("Number traced:",number)
```

```
Number traced: 30
Number traced: 1789
Number traced: 04
Number traced: -30
Number traced: -1789
```

```
Number traced: -18.5
Number traced: 4
Number traced: 1793
Number traced: 03
Number traced: 04
Number traced: 1793
Number traced: 135
Number traced: 8
Number traced: 445
Number traced: 4
Number traced: 1841
Number traced: 2
Number traced: 308
Number traced: 737
Number traced: 20
Number traced: 1945
Number traced: 01
Number traced: -20
Number traced: -1945
Number traced: 559.0
Number traced: 1921
Number traced: 1925
Number traced: 1949
Number traced: 1865
Number traced: 1933
Number traced: 1961
Number traced: 1829
Number traced: 1829
Number traced: 37.0
Number traced: 1981
Number traced: -55.5
Number traced: 20.8
Number traced: -3.0
```

##Your explanation:

In the above task, I have tried to implement a pattern which finds the positive and negative with both integer and float data type. For this I have imported regular expression using re. I have written a pattern to find the given as "r'-?\d+(?:.\d+)?'" where r is a raw string and the '-' indicates either positive or negative number. The 'd+' matches one or more digits. '?:.\d+' matches a floating number with one or more digits. The .findall() function is used to return a list of the used pattern that occurs in given string.

##**Question-3.B**

##*Identify all the dates of all forms - text form(April 20, 1945) and digit form(xx-xx-xxxx, xx/xx/xxxx) in the "Inaugural Address" excerpt and write a regular expression that finds all occurrences of the dates in the text. Then, run the Python code snippet to automatically display a list of all such dates identified.*

NOTE: You can use the *re.findall()* method as demonstrated in the example before this exercise.

```
##Your code here
import re

#here i am giving a  Regular expression pattern to match dates in text
form and digit form:
date_pattern = r"(?:\b(?:\w+\s+\d{1,2},\s+\d{4}|\d{2}[/-]\d{2}[/-]\
d{4})\b)"

#Here i am Finding all occurrences of dates in the text using the
pattern:
dates = re.findall(date_pattern, inau_text)

#here i am Printing the identified dates:
for date in dates:
    print(date)
```

```
April 30, 1789
04-30-1789
March 4, 1793
03/04/1793
March 4, 1841
January 20, 1945
01-20-1945
```

##Your explanation

In the above task,we need to identify the dates of all forms i.e text form and digit form. I have built a date pattern where the '?:' is used to match two sub groups. The '\b' represents a word boundary where the pattern matches at the start or end of a word. The '?:\w+\s+\d{1,2},\s+\d{4}' subpattern matches the required "Month Day, Year" format. The another '\d{2}[/-]\d{2}[/-]\d{4}' subpattern matches the "MM/DD/YYYY" or "MM-DD-YYYY" format. In this way the pattern traces the date formats and displays us from given passage.

#Task 3: Lemmatization/Stemming(25%)

#Question -1:

##How does the morphology of a language (e.g., agglutinative vs. fusional) impact the suitability of stemming vs. lemmatization?

##Your answer here

The morphology of a language, specifically whether it is agglutinative or fusional, can impact the suitability of stemming and lemmatization techniques for text processing. Understanding different types of morphological structures:

1. Stemming: It is a process of reducing words to their base or root form, called a stem which involves removing prefixes, suffixes, and inflectional endings from words. Relatively regular and transparent word structureAgglutinative languages are well-

suited for stemming because of the morphological components in the agglutinative languages are often clearly separable, making it easier to identify and remove affixes.Stemming is particularly effective in agglutinative languages because the affixes are typically easily identifiable and separable.

2. Lemmatization: It involves determining the base or dictionary form of a word, known as the lemma. The word's context and grammatical features to produce the correct base form. Fusional languages, which often have complex and irregular word forms, benefit more from lemmatization. This is because lemmatization considers the syntactic and semantic information of a word to produce its base form, which can vary significantly in fusional languages.Lemmatization is more suitable for fusional languages because it considers the word's context, syntactic role, and grammatical features to determine the base or dictionary form of a word.

In conclusion, the choice between stemming and lemmatization depends on the morphology of the language being processed. Agglutinative languages with regular and transparent structures are more suitable for stemming, while fusional languages with complex and irregular word forms benefit from lemmatization. In practical text processing applications, it's often beneficial to consider both stemming and lemmatization approaches, depending on the requirements and linguistic characteristics of the language being analyzed. However, it's important to note that the effectiveness of these techniques can vary depending on the specific language and the context in which they are applied.

Question - 2 :

Create a Python function that takes a sentence as input, performs lemmatization using **StanfordNLP**, and removes stopwords from the lemmatized sentence. Use a list of stopwords. Return the cleaned and lemmatized sentence.

Reference:https://stanfordnlp.github.io/stanfordnlp/

```
!pip install stanfordnlp
```

```
Requirement already satisfied: stanfordnlp in c:\users\19408\
anaconda3\lib\site-packages (0.2.0)
Requirement already satisfied: requests in c:\users\19408\anaconda3\
lib\site-packages (from stanfordnlp) (2.28.1)
Requirement already satisfied: protobuf in c:\users\19408\anaconda3\
lib\site-packages (from stanfordnlp) (4.25.3)
Requirement already satisfied: tqdm in c:\users\19408\anaconda3\lib\
site-packages (from stanfordnlp) (4.64.1)
Requirement already satisfied: numpy in c:\users\19408\anaconda3\lib\
site-packages (from stanfordnlp) (1.21.5)
Requirement already satisfied: torch>=1.0.0 in c:\users\19408\
anaconda3\lib\site-packages (from stanfordnlp) (2.1.0)
Requirement already satisfied: filelock in c:\users\19408\anaconda3\
```

```
lib\site-packages (from torch>=1.0.0->stanfordnlp) (3.6.0)
Requirement already satisfied: typing-extensions in c:\users\19408\
anaconda3\lib\site-packages (from torch>=1.0.0->stanfordnlp) (4.9.0)
Requirement already satisfied: sympy in c:\users\19408\anaconda3\lib\
site-packages (from torch>=1.0.0->stanfordnlp) (1.10.1)
Requirement already satisfied: networkx in c:\users\19408\anaconda3\
lib\site-packages (from torch>=1.0.0->stanfordnlp) (2.8.4)
Requirement already satisfied: jinja2 in c:\users\19408\anaconda3\lib\
site-packages (from torch>=1.0.0->stanfordnlp) (2.11.3)
Requirement already satisfied: fsspec in c:\users\19408\anaconda3\lib\
site-packages (from torch>=1.0.0->stanfordnlp) (2024.2.0)
Requirement already satisfied: certifi>=2017.4.17 in c:\users\19408\
anaconda3\lib\site-packages (from requests->stanfordnlp) (2022.9.14)
Requirement already satisfied: idna<4,>=2.5 in c:\users\19408\
anaconda3\lib\site-packages (from requests->stanfordnlp) (3.3)
Requirement already satisfied: urllib3<1.27,>=1.21.1 in c:\users\
19408\anaconda3\lib\site-packages (from requests->stanfordnlp)
(1.26.11)
Requirement already satisfied: charset-normalizer<3,>=2 in c:\users\
19408\anaconda3\lib\site-packages (from requests->stanfordnlp) (2.0.4)
Requirement already satisfied: colorama in c:\users\19408\anaconda3\
lib\site-packages (from tqdm->stanfordnlp) (0.4.6)
Requirement already satisfied: MarkupSafe>=0.23 in c:\users\19408\
anaconda3\lib\site-packages (from jinja2->torch>=1.0.0->stanfordnlp)
(2.0.1)
Requirement already satisfied: mpmath>=0.19 in c:\users\19408\
anaconda3\lib\site-packages (from sympy->torch>=1.0.0->stanfordnlp)
(1.2.1)
```

```
!pip install stanza
```

```
Collecting stanza
  Downloading stanza-1.8.0-py3-none-any.whl (970 kB)
     ------------------------------------ 970.4/970.4 kB 4.7 MB/s
eta 0:00:00
Requirement already satisfied: numpy in c:\users\19408\anaconda3\lib\
site-packages (from stanza) (1.21.5)
Requirement already satisfied: networkx in c:\users\19408\anaconda3\
lib\site-packages (from stanza) (2.8.4)
Requirement already satisfied: torch>=1.3.0 in c:\users\19408\
anaconda3\lib\site-packages (from stanza) (2.1.0)
Requirement already satisfied: protobuf>=3.15.0 in c:\users\19408\
anaconda3\lib\site-packages (from stanza) (4.25.3)
Requirement already satisfied: toml in c:\users\19408\anaconda3\lib\
site-packages (from stanza) (0.10.2)
Requirement already satisfied: tqdm in c:\users\19408\anaconda3\lib\
site-packages (from stanza) (4.64.1)
Collecting emoji
  Downloading emoji-2.10.1-py2.py3-none-any.whl (421 kB)
     ------------------------------------ 421.5/421.5 kB 27.4 MB/s
```

```
eta 0:00:00
Requirement already satisfied: requests in c:\users\19408\anaconda3\
lib\site-packages (from stanza) (2.28.1)
Requirement already satisfied: filelock in c:\users\19408\anaconda3\
lib\site-packages (from torch>=1.3.0->stanza) (3.6.0)
Requirement already satisfied: typing-extensions in c:\users\19408\
anaconda3\lib\site-packages (from torch>=1.3.0->stanza) (4.9.0)
Requirement already satisfied: sympy in c:\users\19408\anaconda3\lib\
site-packages (from torch>=1.3.0->stanza) (1.10.1)
Requirement already satisfied: jinja2 in c:\users\19408\anaconda3\lib\
site-packages (from torch>=1.3.0->stanza) (2.11.3)
Requirement already satisfied: fsspec in c:\users\19408\anaconda3\lib\
site-packages (from torch>=1.3.0->stanza) (2024.2.0)
Requirement already satisfied: certifi>=2017.4.17 in c:\users\19408\
anaconda3\lib\site-packages (from requests->stanza) (2022.9.14)
Requirement already satisfied: idna<4,>=2.5 in c:\users\19408\
anaconda3\lib\site-packages (from requests->stanza) (3.3)
Requirement already satisfied: charset-normalizer<3,>=2 in c:\users\
19408\anaconda3\lib\site-packages (from requests->stanza) (2.0.4)
Requirement already satisfied: urllib3<1.27,>=1.21.1 in c:\users\
19408\anaconda3\lib\site-packages (from requests->stanza) (1.26.11)
Requirement already satisfied: colorama in c:\users\19408\anaconda3\
lib\site-packages (from tqdm->stanza) (0.4.6)
Requirement already satisfied: MarkupSafe>=0.23 in c:\users\19408\
anaconda3\lib\site-packages (from jinja2->torch>=1.3.0->stanza)
(2.0.1)
Requirement already satisfied: mpmath>=0.19 in c:\users\19408\
anaconda3\lib\site-packages (from sympy->torch>=1.3.0->stanza) (1.2.1)
Installing collected packages: emoji, stanza
Successfully installed emoji-2.10.1 stanza-1.8.0
```

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

```
[nltk_data] Downloading package stopwords to
[nltk_data]     C:\Users\19408\AppData\Roaming\nltk_data...
[nltk_data]   Package stopwords is already up-to-date!

True
```

```
#CODE HERE
import stanza
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize

def clean_and_lemmatize_sentence(sentence):
    stanza.download('en')

#Here i have Loaded the English model for stanza:
    nlp = stanza.Pipeline('en')
```

```
# Here I Lemmatized the sentence using stanza:
    doc = nlp(sentence)
    lemmas = []
    for sentence in doc.sentences:
        for word in sentence.words:
            lemmas.append(word.lemma)

#Here I Removed the stopwords from the lemmatized sentence:
    stop_words = set(stopwords.words('english'))
    cleaned_lemmas = [lemma for lemma in lemmas if lemma.lower() not
in stop_words]

#Here I  Returned the cleaned and lemmatized sentence:
    cleaned_sentence = ' '.join(cleaned_lemmas)
    return cleaned_sentence
sentence = "The predicting of future is'nt magic, Its Artificial
Intelligence"
cleaned_lemmatized_sentence = clean_and_lemmatize_sentence(sentence)
print(cleaned_lemmatized_sentence)
```

```
{"model_id":"1137a87ac1eb4eaeb0b46c6024b5754c","version_major":2,"vers
ion_minor":0}
```

```
2024-02-29 14:20:51 INFO: Downloaded file to C:\Users\19408\
stanza_resources\resources.json
2024-02-29 14:20:51 INFO: Downloading default packages for language:
en (English) ...
2024-02-29 14:20:53 INFO: File exists: C:\Users\19408\
stanza_resources\en\default.zip
2024-02-29 14:20:58 INFO: Finished downloading models and saved to C:\
Users\19408\stanza_resources
2024-02-29 14:20:58 INFO: Checking for updates to resources.json in
case models have been updated.  Note: this behavior can be turned off
with download_method=None or
download_method=DownloadMethod.REUSE_RESOURCES
```

```
{"model_id":"187a5b0ce7f84fd3a5de66929e22c23e","version_major":2,"vers
ion_minor":0}
```

```
2024-02-29 14:20:59 INFO: Downloaded file to C:\Users\19408\
stanza_resources\resources.json
2024-02-29 14:21:01 INFO: Loading these models for language: en
(English):
=============================================
| Processor    | Package                    |
---------------------------------------------
| tokenize     | combined                   |
| mwt          | combined                   |
| pos          | combined_charlm            |
```

```
| lemma        | combined_nocharlm          |
| constituency | ptb3-revised_charlm        |
| depparse     | combined_charlm            |
| sentiment    | sstplus_charlm             |
| ner          | ontonotes-ww-multi_charlm |
=============================================

2024-02-29 14:21:01 INFO: Using device: cpu
2024-02-29 14:21:01 INFO: Loading: tokenize
2024-02-29 14:21:01 INFO: Loading: mwt
2024-02-29 14:21:01 INFO: Loading: pos
2024-02-29 14:21:02 INFO: Loading: lemma
2024-02-29 14:21:02 INFO: Loading: constituency
2024-02-29 14:21:02 INFO: Loading: depparse
2024-02-29 14:21:03 INFO: Loading: sentiment
2024-02-29 14:21:03 INFO: Loading: ner
2024-02-29 14:21:04 INFO: Done loading processors!

predict future magic , artificial intelligence
```

##Your answer here

In the above task, I have tried to import stanfordnlp but given a note that All development, issues, ongoing maintenance, and support have been moved to new GitHub repository as the toolkit is being renamed as Stanza since version 1.0.0. It was given to visit new website for more information where we can still download stanfordnlp via pip, but newer versions of this package will be made available as stanza. I have used stanza for lemmatization, stopwords from nltk.corpus for stopwords, and word_tokenize from nltk.tokenize for tokenization Using NLTK library it provides a wide range of natural language processing functionalities and created a lemmatize_and_remove_stopwords function for removal of stop words. In this I have achieved lemmatization of sentence using different stopwords. The final words are printed as output which has no stopwords in it.

##**Question - 3**

(refer to the byte pair encoding concept which is explained in activity-2 at the Tutorial of Subword Tokenization using HuggingFace after the Question6 in activity-2)

Consider the following two sentences:

S1:I like yellow roses better than red ones.

S2: Looks like John is bettering the working conditions at his organization

Create a Python function that encodes two sentences using the custom BPE tokenizer and identifies common subword tokens (tokens that appear in both encodings). Return a list of these common subword tokens. Is/Are there any interesting observations when you compare the tokens between the two encodings? What do you think is causing what you observe as part of your comparison?

```
from tokenizers import ByteLevelBPETokenizer
import tempfile
import os

def encode_and_find_common_subwords(sentence1, sentence2):
#Here I have Created temporary text files:
    with tempfile.NamedTemporaryFile(mode="w", delete=False) as
temp_file1, tempfile.NamedTemporaryFile(mode="w", delete=False) as
temp_file2:
        temp_file1.write(sentence1)
        temp_file2.write(sentence2)
        temp_file1.flush()
        temp_file2.flush()
        temp_file_paths = [temp_file1.name, temp_file2.name]

#here I have Initialized a ByteLevelBPETokenizer:
        tokenizer = ByteLevelBPETokenizer()

#Here I have Trained the tokenizer on the temporary files:
        tokenizer.train(files=temp_file_paths, vocab_size=1000,
min_frequency=2, special_tokens=["[UNK]", "[CLS]", "[SEP]", "[PAD]",
"[MASK]"])

# This is for Encoded sentence 1:
        encoded_sentence1 = tokenizer.encode(sentence1).tokens

#This is for Encode sentence 2:
        encoded_sentence2 = tokenizer.encode(sentence2).tokens

#This is to Find common subword tokens:
        common_tokens = list(set(encoded_sentence1) &
set(encoded_sentence2))
#To Remove temporary files:
    for temp_file_path in temp_file_paths:
        os.remove(temp_file_path)

    return common_tokens
sentence1 = "nlp is my favourite subject "
sentence2 = "most of the subjects I like are related to artificial
intelligence"

common_tokens = encode_and_find_common_subwords(sentence1, sentence2)
print(common_tokens)
```

```
['Ġ', 's', 'a', 'm', 'Ġi', 'te', 'r', 'i', 'Ġsubject', 'f', 'o', 'n',
'l']
```

Exlpanation for above code:

In the above task, I have imported the 'ByteLevelBPETokenizer' from the tokenizer and also I have imported the tempfile and os. I have used sentence-1 and sentence-2 which takes as inputs

for the encode_and_find_common_subwords function.The temporary files are pushed into contents to write on disk.The `tokenizer.encode(sentence1).tokens` encodes `sentence1` and `tokenizer.encode(sentence2).tokens` encodes `sentence2` using the trained tokenizer and returns a list of subword tokens. The `encode_and_find_common_subwords` function by providing two example sentences: `sentence1` and `sentence2`. I have printed the output, which shows the list of common subword tokens.

Task - 4 : Minimum Edit distance (25%)

##Minimum edit Distance Minimum Edit Distance (also known as Levenshtein Distance) is a measure of similarity between two strings by calculating the minimum number of single-character edits (insertions, deletions, substitutions) required to transform one string into the other. It has applications in various fields, including natural language processing, spell checking, DNA sequence alignment, and more.

Character Based Text Similarity

"As an example, this technology is used by information retrieval systems, search engines, automatic indexing systems, text summarizers, categorization systems, plagiarism checkers, speech recognition, rating systems, DNA analysis, and profiling algorithms (IR/AI programs to automatically link data between people and what they do)."

```
##code block -4
# A Naive recursive Python program to find minimum number
# operations to convert str1 to str2


def editDistance(str1, str2, m, n):

    # If first string is empty, the only option is to
    # insert all characters of second string into first
    if m == 0:
        return n

    # If second string is empty, the only option is to
    # remove all characters of first string
    if n == 0:
        return m

    # If last characters of two strings are same, nothing
    # much to do. Ignore last characters and get count for
    # remaining strings.
    if str1[m-1] == str2[n-1]:
        return editDistance(str1, str2, m-1, n-1)
```

```
    # If last characters are not same, consider all three
    # operations on last character of first string, recursively
    # compute minimum cost for all three operations and take
    # minimum of three values.
    return 1 + min(editDistance(str1, str2, m, n-1),    # Insert
                   editDistance(str1, str2, m-1, n),    # Remove
                   editDistance(str1, str2, m-1, n-1)    # Replace
                   )


# Driver code
str1 = "sunday"
str2 = "saturday"
print (editDistance(str1, str2, len(str1), len(str2)))
```

```
3
```

Refer above code block -4 for the following question

##Question-1

##Assuming case sensitivity where changing a letter's case has a cost of 1, calculate the minimum cost to transform "Imagine" into "imagination" with the following operation costs: insertions = 2, deletions = 2, substitutions = 3

```
##your code here

def editDistance(str1, str2, m, n, case_insensitive=False):
    if m == 0:
        return n

    if n == 0:
        return m

    if case_insensitive and str1[m-1].lower() == str2[n-1].lower():
        return editDistance(str1, str2, m-1, n-1, case_insensitive)

    if str1[m-1] == str2[n-1]:
        return editDistance(str1, str2, m-1, n-1, case_insensitive)

    if case_insensitive:
        cost = 1
    else:
        cost = 3

    return min(
        #This is used for Insertion:
        cost + editDistance(str1, str2, m, n-1, case_insensitive),
        #This is used for Removal:
```

```
        cost + editDistance(str1, str2, m-1, n, case_insensitive),
        #This is used for replacing:
        cost + editDistance(str1, str2, m-1, n-1, case_insensitive)
    )


str1 = "Imagine"
str2 = "imagination"
case_insensitive = True
print(editDistance(str1, str2, len(str1), len(str2),
case_insensitive))
```

```
5
```

##Your explanation

In the above task I have used edit distance function, in that string-1 and string-2 are two inputs and m,n are lengths of the strings. case-insensitive function gives whether the word is sensitive or not. If length of one string is 0 it returns to another string. The edit distance continues checking for low cost distance. The transformation happens between the strings imagine and imagination and it gives the final cost.

##Tutorial -2

Levenshtein Distance for Sentences

```
#code block - 5
# Simple Minimum Edit Distance
def levenshtein_distance(str1, str2):
    # Initialize a matrix to store edit distances
    m, n = len(str1), len(str2)
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    # Initialize the first row and column
    for i in range(m + 1):
        dp[i][0] = i
    for j in range(n + 1):
        dp[0][j] = j

    # Fill in the matrix using dynamic programming
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            cost = 0 if str1[i - 1] == str2[j - 1] else 2  #
Substitution cost is 2
            dp[i][j] = min(
                dp[i - 1][j] + 1,  # Deletion
                dp[i][j - 1] + 1,  # Insertion
                dp[i - 1][j - 1] + cost,  # Substitution
            )
```

```
    # The final value in the matrix represents the Levenshtein
distance
    return dp[m][n]

# Example usage
str1 = "This is a cat"
str2 = "That is a dog"
distance = levenshtein_distance(str1, str2)
print(f"The Levenshtein distance between '{str1}' and '{str2}' with
substitution cost 2 is {distance}")
```

```
The Levenshtein distance between 'This is a cat' and 'That is a dog'
with substitution cost 2 is 10
```

Refer to above code block -5 from tutorial - 2 for the following question

##Question:2

##Assign different costs to insertions, deletions, and substitutions to reflect varying penalties for different types of edits. Calculate the Levenshtein distance with these weighted costs.

String1 = ("Natural language processing")

String2 = ("Computer science department")

Provide your explanation in the tex block below

```
##ENTER YOUR CODE HERE
def levenshtein_distance(str1, str2, deletion_cost, insertion_cost,
substitution_cost):
    m, n = len(str1), len(str2)
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    for i in range(m + 1):
        dp[i][0] = i * deletion_cost

    for j in range(n + 1):
        dp[0][j] = j * insertion_cost

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            cost = 0 if str1[i - 1] == str2[j - 1] else
substitution_cost
            dp[i][j] = min(
                dp[i - 1][j] + deletion_cost,
                dp[i][j - 1] + insertion_cost,
                dp[i - 1][j - 1] + cost,
            )

    return dp[m][n]
```

```
str1 = "Natural language processing"
str2 = "Computer science department"
deletion_cost = 1
insertion_cost = 3
substitution_cost = 2

distance = levenshtein_distance(str1, str2, deletion_cost,
insertion_cost, substitution_cost)
print(f"The Levenshtein distance between '{str1}' and '{str2}' with
weighted costs is {distance}")
```

```
The Levenshtein distance between 'Natural language processing' and
'Computer science department' with weighted costs is 46
```

##Your explanation

In the above code, I have modified and added 3 parameter to the provided code which represents deletions, insertions and substitutions. I have set deletion_cost to 1 substitution_cost to 2 and insertion_cost to 3 for the different types of edits. The Levenshtein distance between the two strings with the weighted costs is then calculated using the modified levenshtein_distance function, and the result is printed to the console. The output for the Levenshtein distance between "Natural language processing" and "Computer science department" considering the specified weighted costs for different types of edits.

##Question-3

##Describe how MED is used in various NLP tasks, such as spell checking, speech recognition, text summarization, and machine translation. How does its effectiveness vary across these tasks?

##Your Explanation Here -->

The Minimum edit distance(MED) is used in various NLP tasks in finding the similarity and dissimilarity in between two strings. In spell checking is used to identify and correct the misspelled words. It also calculates the MED between misspelled and correct one. The speech recognition in context of MED is used for translation of word or phrase. comparing features of input helps to identify closest map. When it comes to text summarization MED can be used for finding the similarities among sentences or docs. Machine translation used MED for translation of words. It gives between source and targeted values. On conclusion the effectiveness of MED not only depend on NLP tasks but also on different characteristics and complexties of the given task. It aslo depends on quality of training, availability, algorthms required, advance techniques and challenges to each task.