NLP Assignment 2

Names: Ali Ammar, Hafsa Azhar

Group: 10

Making your own tokenizer (0.5 pt)

For this assignment, make a simple tokenizer. Write 3 sentences and try the tokenizer out on them.
What to submit:

Question 2.1 Making your own tokenizer

In [5]:

```
def my_tokenizer(text):
    """
    The implementation of your own tokenizer

    :param text:  A string with a sentence (or paragraph, or document...)
    :return  A list of tokens
    """


    tokenized_text = re.findall(r'\w+|[^\w\s]', text)
    #Here I am using the regular expression for all the occurences, where it will split the text.
    #  \w+ are based on characters , | indicates OR expression and  [^\w\s] are for non-words such as punctuation.

    return tokenized_text

sample_string0 = "If you have the chance, watch it. Although, a warning, you'll cry your eyes out."
sample_string1 = "I wish life would be a bit easy"
sample_string2 = "I wish to go to Japan every once in a year. Wishes do come true. Right?"
sample_string3 = "Hello, world! How are you?"
print(my_tokenizer(sample_string0))
print(my_tokenizer(sample_string1))
print(my_tokenizer(sample_string2))
print(my_tokenizer(sample_string3))
```

```
['If', 'you', 'have', 'the', 'chance', ',', 'watch', 'it', '.', 'Although', ',', 'a', 'warning', ',', 'you', "'", 'll', 'cr
y', 'your', 'eyes', 'out', '.']
['I', 'wish', 'life', 'would', 'be', 'a', 'bit', 'easy']
['I', 'wish', 'to', 'go', 'to', 'Japan', 'every', 'once', 'in', 'a', 'year', '.', 'Wishes', 'do', 'come', 'true', '.', 'Righ
t', '?']
['Hello', ',', 'world', '!', 'How', 'are', 'you', '?']
```

• Provide a description of how your tokenizer works.

From the above, we can see that tokenizer works quite well, it is tokenizing each words such as it tokenized you'll into 'you' " ' " 'll' and tokenize the Symbols separately such as !, . , and ?.

• Report the tokens you obtain when using your tokenizer on your example sentences.

```
['If', 'you', 'have', 'the', 'chance', ',', 'watch', 'it', '.', 'Although'
, ',', 'a', 'warning', ',', 'you', "'", 'll', 'cry', 'your', 'eyes', 'out'
, '.']
['I', 'wish', 'life', 'would', 'be', 'a', 'bit', 'easy']
['I', 'wish', 'to', 'go', 'to', 'Japan', 'every', 'once', 'in', 'a', 'year
', '.', 'Wishes', 'do', 'come', 'true', '.', 'Right', '?']
['Hello', ',', 'world', '!', 'How', 'are', 'you', '?']
```

Using an off-the-shelf tokenizer (1 pt)

Compare the tokenizer you implemented in the previous question with one from NLTK, using the sentences provided in the Notebook.

What to submit: Reflect and answer these questions:

Question 2.2 Using an off-the-shelf tokenizer

In [6]:

```
#Now we are gonna compare the tokenizer you just wrote with the one from NLTK
#if you installed NLTK but never downloaded the 'punkt' tokenizer, uncomment the following lines:
import nltk
nltk.download('punkt')
from nltk.tokenize import word_tokenize

def nltk_tokenizer(text):
    """
    This function should apply the word_tokenize (punkt) tokenizer of nltk to the input text

    :param text:  A string with a sentence (or paragraph, or document...)
    :return  A list of tokens
    """


    tokenized_text= word_tokenize(text)

    return tokenized_text

test_sentences = ["I like this assignment because:\n-\tit is fun;\n-\tit helps me practice my Python skills.",
        "I won a prize, but I won't be able to attend the ceremony.",
        "“The strange case of Dr. Jekyll and Mr. Hyde” is a famous book... but I haven't read it.",
        "I work for the C.I.A.. And you?",
        "OMG #Twitter is sooooo coooool <3 :-) <-- lol...why do i write like this idk right? :) 🔥😂 🚔"]

for test_string in test_sentences:
    print("my_tokenizer =", my_tokenizer(test_string))
    print("nltk_tokenizer =", nltk_tokenizer(test_string))

    #print(my_tokenizer(test_string))

    #print(nltk_tokenizer(test_string))
    print("\n")
```

• What are the differences in the two tokenizer outputs?

```
my_tokenizer = ['I', 'like', 'this', 'assignment', 'because', ':', '-', 'it', 'is', 'fun', ';', '-', 'it', 'helps', 'me', 'p
ractice', 'my', 'Python', 'skills', '.']
nltk_tokenizer = ['I', 'like', 'this', 'assignment', 'because', ':', '-', 'it', 'is', 'fun', ';', '-', 'it', 'helps', 'me',
'practice', 'my', 'Python', 'skills', '.']


my_tokenizer = ['I', 'won', 'a', 'prize', ',', 'but', 'I', 'won', "'", 't', 'be', 'able', 'to', 'attend', 'the', 'ceremony',
'.']
nltk_tokenizer = ['I', 'won', 'a', 'prize', ',', 'but', 'I', 'wo', "n't", 'be', 'able', 'to', 'attend', 'the', 'ceremony',
'.']


my_tokenizer = ['“', 'The', 'strange', 'case', 'of', 'Dr', '.', 'Jekyll', 'and', 'Mr', '.', 'Hyde', '”', 'is', 'a', 'famou
s', 'book', '.', '.', '.', 'but', 'I', 'haven', "'", 't', 'read', 'it', '.']
nltk_tokenizer = ['“', 'The', 'strange', 'case', 'of', 'Dr.', 'Jekyll', 'and', 'Mr.', 'Hyde', '”', 'is', 'a', 'famous', 'boo
k', '...', 'but', 'I', 'have', "n't", 'read', 'it', '.']


my_tokenizer = ['I', 'work', 'for', 'the', 'C', '.', 'I', '.', 'A', '.', '.', 'And', 'you', '?']
nltk_tokenizer = ['I', 'work', 'for', 'the', 'C.I.A', '..', 'And', 'you', '?']


my_tokenizer = ['OMG', '#', 'Twitter', 'is', 'sooooo', 'coooool', '<', '3', ':', '-', ')', '<', '-', '-', 'lol', '.', '.',
'.', 'why', 'do', 'i', 'write', 'like', 'this', 'idk', 'right', '?', ':', ')', '🔥', '😂', '🚔']
nltk_tokenizer = ['OMG', '#', 'Twitter', 'is', 'sooooo', 'coooool', '<', '3', ':', '-', ')', '<', '--', 'lol', '...', 'why',
'do', 'i', 'write', 'like', 'this', 'idk', 'right', '?', ':', ')', '🔥😂', '🚔']
```

From above we can see :

1. There is not much difference when it comes to first sentence.
2. my_tokenizer tokenize won't to 'won', " ' ", 't' whereas nltk_tokenizer tokenize to 'wo' , "n't".
3. my_tokenizer tokenize Dr.Jekyll to 'Dr', " . ", 'Jekyll' whereas nltk_tokenizer tokenize to 'Dr.' , 'Jekyll' and same with Mr.Hyde. Furthermore, my_tokenizer tokenize … to '.', " . ", '.' whereas nltk_tokenizer tokenize to '…' and for haven't my_tokenizer tokenize 'haven', " ' ", 't' whereas nltk_tokenizer tokenize to 'have' , 'n't'
4. my_tokenizer tokenize C.I.A to 'C' , " . ", 'I' , " . ", 'A' whereas nltk_tokenizer tokenize to 'C.I.A'.
5. my_tokenizer tokenize every single emoji separately whereas nltk_tokenizer tokenize some together and some separately.

From this we can say my_tokenizer tokenize too much whereas nltk_tokenizer tokenize less.

• While coding your tokenizer, did you foresee all these inputs?

While coding our tokenizer, we did foresee all these inputs.

• Is there a single 'perfect tokenizer'?

The efficiency of a tokenizer depends on the precise task, language, and text data being processed.it does vary across languages and scripts. The domain and the nature of the data also impact tokenization as tokenization on articles might be different from tokenization for social media text. There is no generally perfect tokenizer but rather a range of tokenization tools that can be selected and customized based on the requirements of a particular project.

Text classification with a unigram language model

Theory

What to submit:

• The probability of P(boring|Pos), using Laplace smoothing, showing the formula you used and intermediate calculations.

Smoothing We use smoothing to avoid zero probabilities:

$$P(w_i|Pos) \approx \frac{C(w_i, Pos) + k}{\sum_{w \in V} C(w, Pos) + kV}$$

Where V is the size of your vocabulary. With $k = 1$ we are using Laplace smoothing.

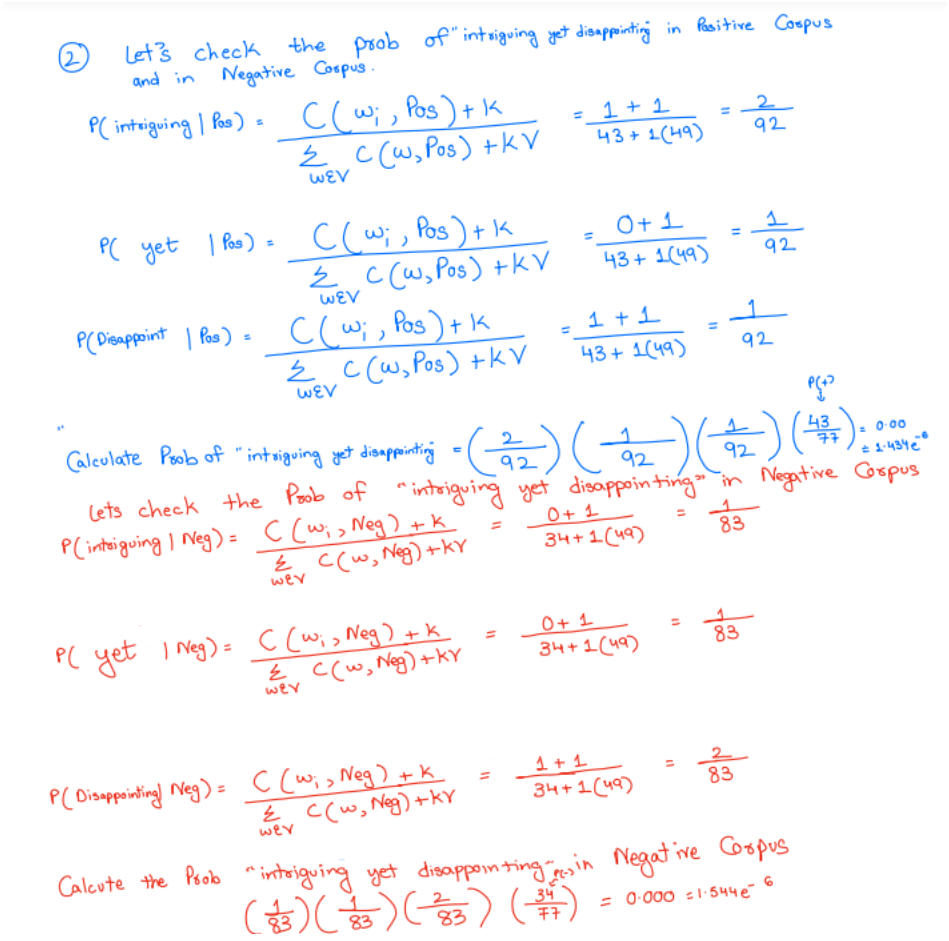
$P(\text{Boring}|\text{Pos}) \approx \frac{C(\text{Boring}, \text{Pos}) + k}{\sum_{w \in V} C(w, \text{Pos}) + kV} \Rightarrow \frac{0+1}{43+49} = \frac{1}{43+49} = \frac{1}{92}$ $k = 1$

0 : There is no boring word in Positive Review.

C(w,Pos) 43 : 5+3+10+25 = 43 (Total no of words in PR)

|V| 49 : 3+5+6+35 = 49 (Total no of unique words in corpus)

Would “intriguing yet disappointing” be classified as a positive or negative review? Why (show the probabilities used to decide it)?



So, intriguing yet disappointing” is considered as Negative corpus.

Coding

1. The performance of your classifier (accuracy) when running with and without Laplace smoothing (k = 1 and k = 0 respectively).

Now train two more models: one without Laplace smoothing, and one where stopwords are removed. Then test them on the same test data, and compare the performance with the results you previously obtained.

In [9]:

```
classifier_with_smoothing = MultinomialNB(alpha=1) #k=1
classifier_with_smoothing.fit(X_train_counts, data['Label'])

classifier_no_smoothing = MultinomialNB(alpha=0) #k=0
classifier_no_smoothing.fit(X_train_counts, data['Label'])

# Calculate predictions with smoothing
predictions_with_smoothing = classifier_with_smoothing.predict(X_test_counts)
accuracy_with_smoothing = accuracy_score(test_data['Label'], predictions_with_smoothing)
print("Accuracy with Laplace Smoothing (k = 1):", accuracy_with_smoothing)

# Test the model without Laplace smoothing
predictions_no_smoothing = classifier_no_smoothing.predict(X_test_counts)
accuracy_no_smoothing = accuracy_score(test_data['Label'], predictions_no_smoothing)
print("Accuracy without Laplace Smoothing (k = 0):", accuracy_no_smoothing)
```

```
Accuracy with Laplace Smoothing (k = 1): 0.85
Accuracy without Laplace Smoothing (k = 0): 0.8
```

2. The performance removing stop words vs. without removing them. Are they different? Why is that? •

In [10]:

```
#Now we will check the perfomance with removing stop words vs without removing them
#Lets first train it
vectorizer_no_stopwords= CountVectorizer(stop_words='english',lowercase=True)
X_train_no_stopwords= data['Review']
X_train_counts_no_stopwords = vectorizer_no_stopwords.fit_transform(X_train_no_stopwords)

classifer_no_stopwords= MultinomialNB()
classifer_no_stopwords.fit(X_train_counts_no_stopwords, data['Label'])

#Now lets check the accuracy
# Test the model with stop words removed
X_test_counts_no_stopwords = vectorizer_no_stopwords.transform(test_data['Review'])
predictions_no_stopwords = classifer_no_stopwords.predict(X_test_counts_no_stopwords)
accuracy_no_stopwords = accuracy_score(test_data['Label'], predictions_no_stopwords)
print("Accuracy with Stop Words Removed:", accuracy_no_stopwords)

# Test the model without stop words removed
predictions = classifier.predict(X_test_counts)
accuracy = accuracy_score(test_data['Label'], predictions)
print("Accuracy without Stop Words Removed:", accuracy)
```

```
Accuracy with Stop Words Removed: 0.85
Accuracy without Stop Words Removed: 0.85
```

The accuracy with Stop Words removed and without stop words removed are the same which is 0.85. This means that the choice of removing stop words or doesn't have a significant impact on the classification task for this specific dataset.

3. The performance after disabling the default lowercase normalization (and without stop word removal). Is there a difference, and if so, why do you think there is one?

In [11]:
```
# Create a CountVectorizer without lowercase normalization and stop word removal
vectorizer_no_preprocessing = CountVectorizer(lowercase=False, stop_words=None)
X_train_no_preprocessing = data['Review']
X_train_counts_no_preprocessing = vectorizer_no_preprocessing.fit_transform(X_train_no_preprocessing)

# Train a classifier with the unprocessed text
classifier_no_preprocessing = MultinomialNB()
classifier_no_preprocessing.fit(X_train_counts_no_preprocessing, data['Label'])

#Now lets test the model
X_test_counts = vectorizer.transform(test_data['Review'])
predictions = classifier.predict(X_test_counts)
accuracy = accuracy_score(test_data['Label'], predictions)
print("Accuracy with Preprocessing:", accuracy)

X_test_counts_no_preprocessing = vectorizer_no_preprocessing.transform(test_data['Review'])
predictions_no_preprocessing = classifier_no_preprocessing.predict(X_test_counts_no_preprocessing)
accuracy_no_preprocessing = accuracy_score(test_data['Label'], predictions_no_preprocessing)
print("Accuracy without Preprocessing:", accuracy_no_preprocessing)
```

```
Accuracy with Preprocessing: 0.85
Accuracy without Preprocessing: 0.81
```

The accuracy with Preprocessing is 0.85 whereas, the accuracy without Preprocessing is 0.81. I think it is based on the characteristics of our data. If we look at the work of preprocessing phase, all texts are converted into lower case that means it has reduced the dimensionality of the feature space while in without preprocessing phase words in different case are considered different. Indeed, the lowercase normalization and stop word removal seems to have a positive impact on the models performance.

Text classification with a bigram language model

Theory

- $P(w_i|w_{i-1})$ can be computed by counting the amount the bigram appears in the dataset and divide the amount by the amount w_{i-1}. The final result after smoothing will look like that:

$$P(w_i|w_{i-1}) = \frac{\text{Count}(w_{i-1}, w_i)}{\text{Count}(w_{i-1})}$$

- Using the following corpus

Positive	Negative
This was a great movie: the plot is intriguing, the plot twists are well-thoughtout and the actors are really delivering a great performance.	A terrible movie with a boring plot, once again a reminder that great actors are not enough to shoot an interesting movie.
I really like the movie, the director manages to tell a familiar story we can all identify with.	Disappointing, boring, uninspiring. I wish I had not wasted 10$ to see this.
a familiar, actors are, all identify, and the, are really, are well-thought-out, can all, delivering a, director manages, familiar story, great movie, great performance, i really, identify with, intriguing the, is intriguing, like the, manages to, plot is, plot twists, really delivering, really like,	

story we, tell a, the actors, the director, the movie, this was, to tell, twists are, was a, we can, well-thought-out and,	

To calculate the probability of "great movie" the following formula is used:

$$P(\text{movie}|\text{great}) = \frac{\text{Count}(\text{great movie}) + 1}{\text{Count}(\text{great}) + V}$$

Note that the formula is making use of Laplace smoothing.

Count(great movie)	2
Count(great)	4
Vocabulary (v)	31

Using the given example, the calculation will be $\frac{0+2}{4+31} = 0.05714$

To calculate the probability of "familiar enough" the same formula as the previous part is used:

$$P(\text{enough}|\text{familiar}) = \frac{\text{Count}(\text{familiar enough}) + 1}{\text{Count}(\text{familiar}) + V}$$

Count(familiar enough)	0
Count(familiar)	3
Vocabulary (v)	31

Which equals: $\text{P}(\text{enough}|\text{familiar}) = \frac{0+0}{3+31} = 0$

- To calculate the probability of "uninspiring plot", the same formula as the previous part will be used but without the Laplace smoothing.

$$\text{P}(\text{plot}|\text{uninspiring}) = \frac{\text{Count}(\text{uninspiring plot})}{\text{Count}(\text{uninspiring})}$$

Count(uninspiring plot)	0
Count(uninspiring)	1
Vocabulary (v)	28

$$P(\text{plot}|\text{uninspiring}) = \frac{0}{1} = 0$$

The same is for "terrible failure":

$$\text{P}(\text{failure}|\text{terrible}) = \frac{\text{Count}(\text{terrible failure})}{\text{Count}(\text{terrible})}$$

Count(terrible failure)	0

Count(terrible)	1
Vocabulary (v)	28

$$P(\text{failure}|\text{terrible}) = \frac{0}{1} = 0$$

Code

Model	n = 2	n = 3	n = 4
Accuracy	0.75	0.53	0.51

It looks like having higher ngrams does not improve the accuracy in our case. Possible reasons are:

- Having small dataset which makes it hard to capture patterns effectively.
- Using higher ngrams can increase sparsity.
- Overfitting the data.

Suggestions that can improve the Performance can be:

- Having a bigger dataset.
- Using TF-IDF instead of using raw term frequencies.
- Remove irrelevant ngrams.