# Tokenization

* 1. 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:



• 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']

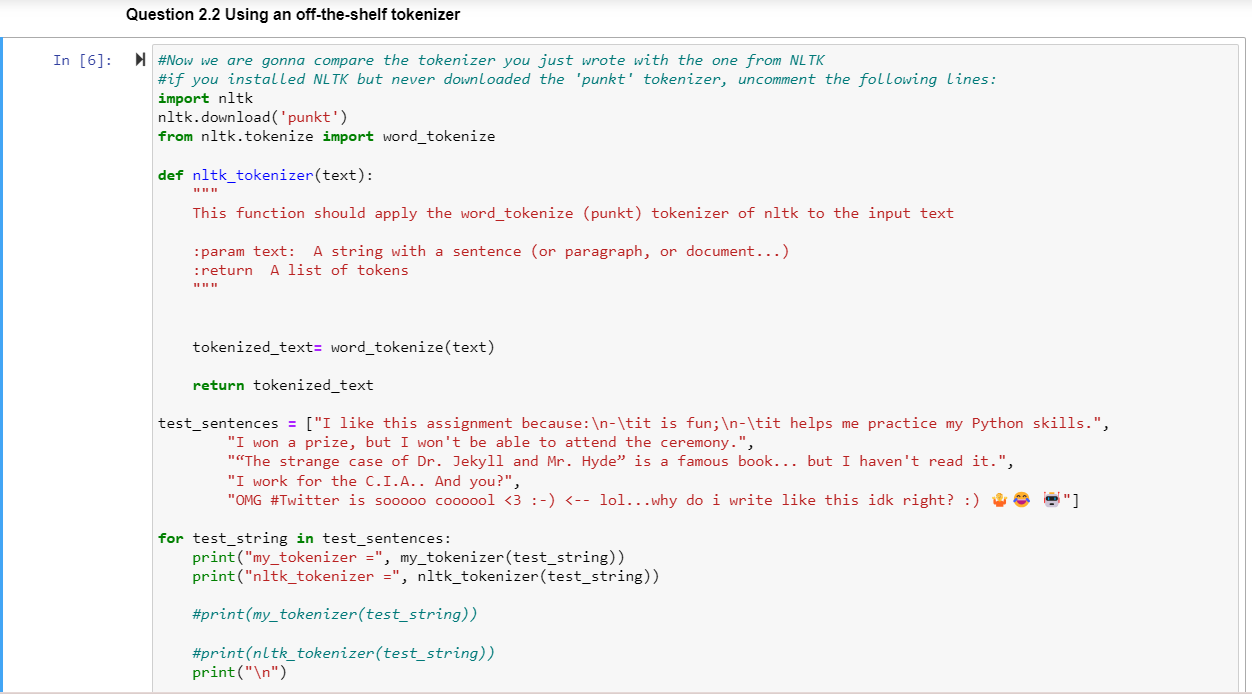
['I', 'wish', 'to', 'go', 'to', 'Japan', 'every', 'once', 'in', 'a', 'year', '.', 'Wishes', 'do', 'come', 'true', '.', 'Right', '?']

['Hello', ',', 'world', '!', 'How', 'are', 'you', '?']

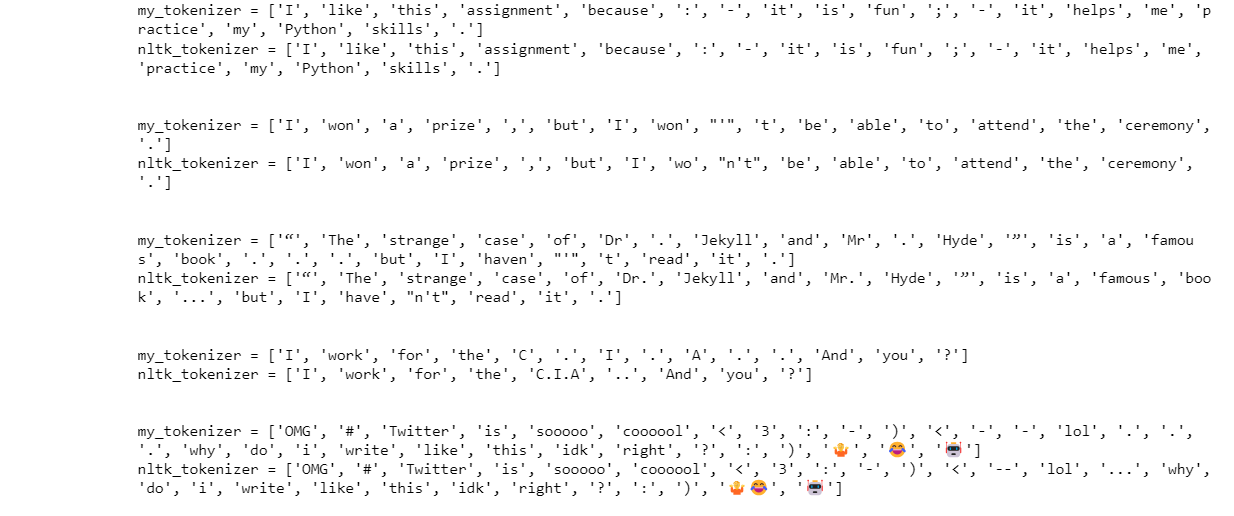
# Using an off-the-shelf tokenizer (1 pt)

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:



• What are the differences in the two tokenizer outputs?



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’?

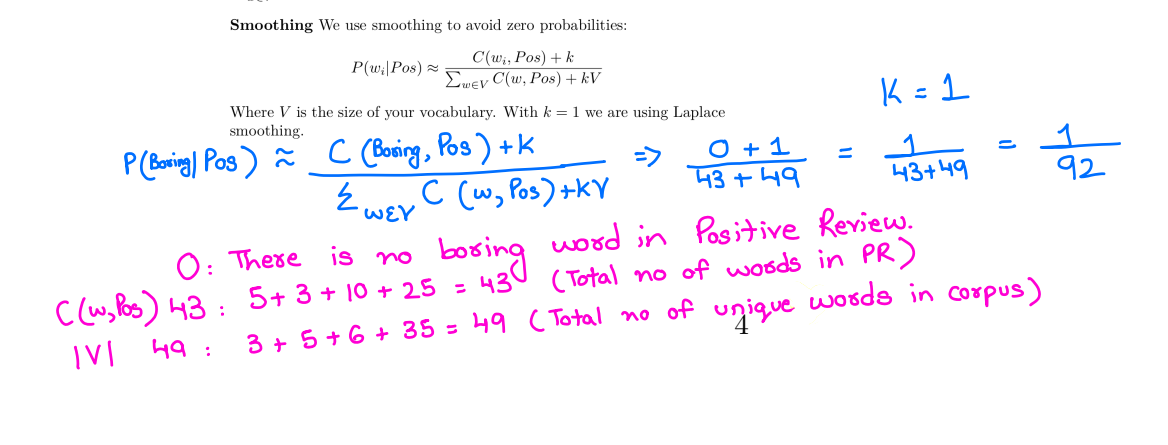
The efficiency of a tokenizer depends on the precise task, language, and text data being processed.it does vary across languages and scripts. The domain and the nature of the data also impact tokenization as tokenization on articles might be different from tokenization for social media text. There is 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

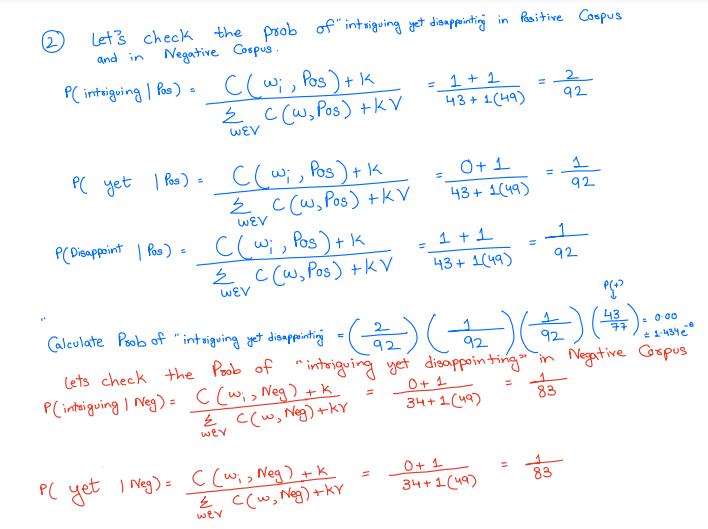
## Theory

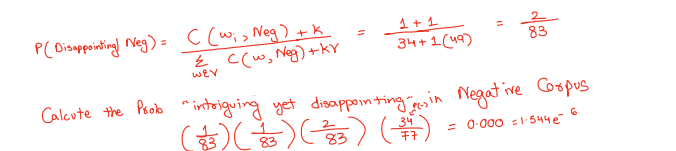
What to submit:

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



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

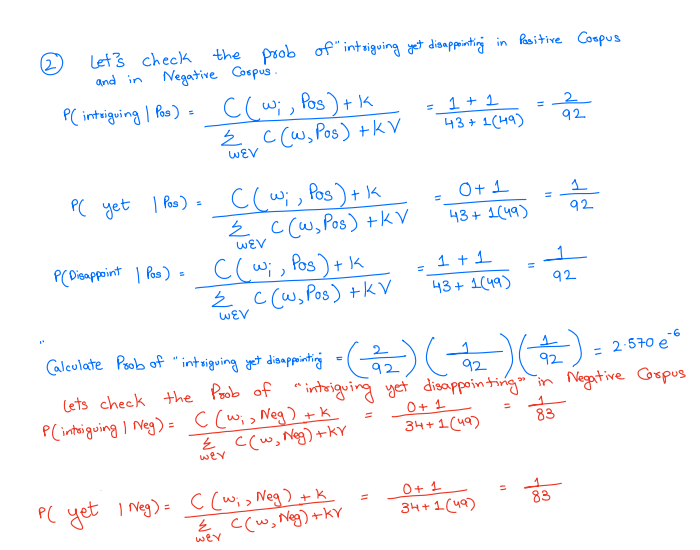


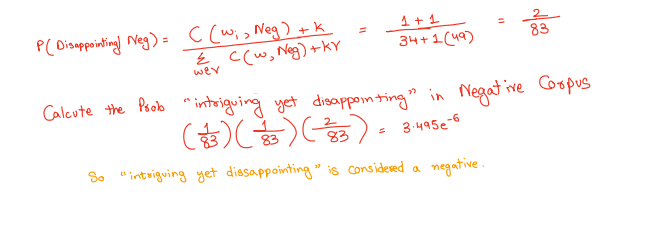


So, intriguing yet disappointing” is considered as Negative corpus.

##Here I added P(+) and P(-)

OR



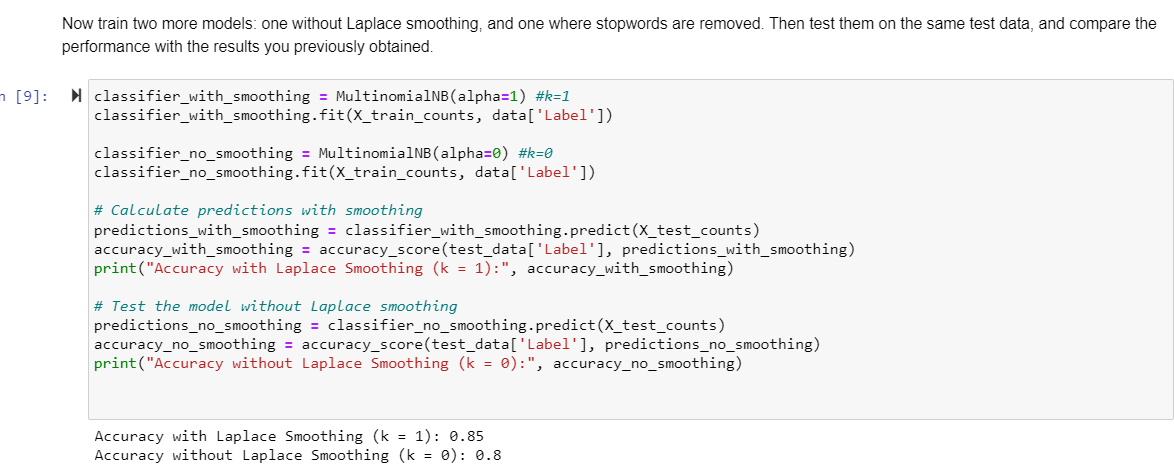


## Without P(-)

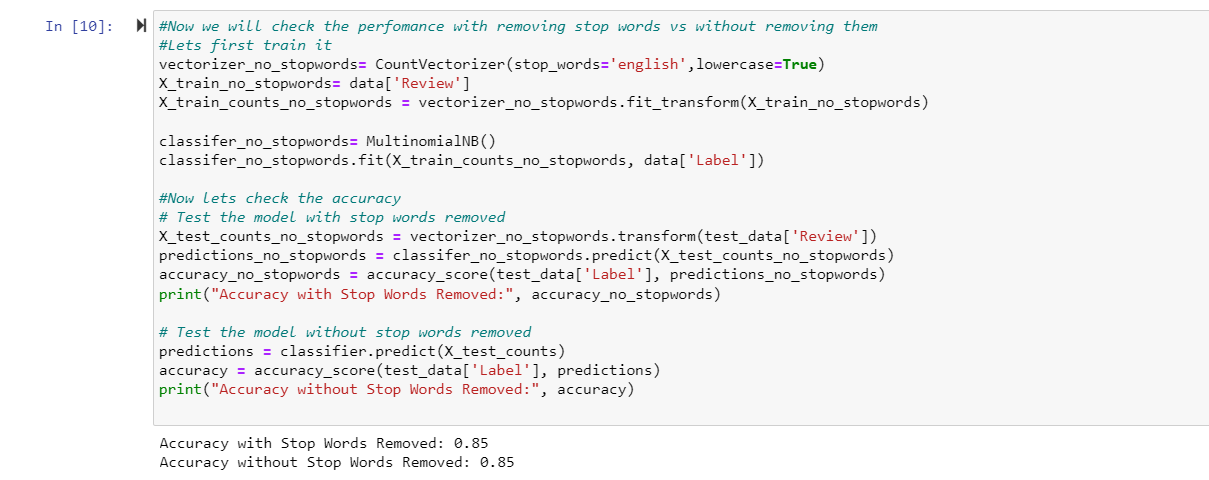
Kindly choose one

# Coding

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

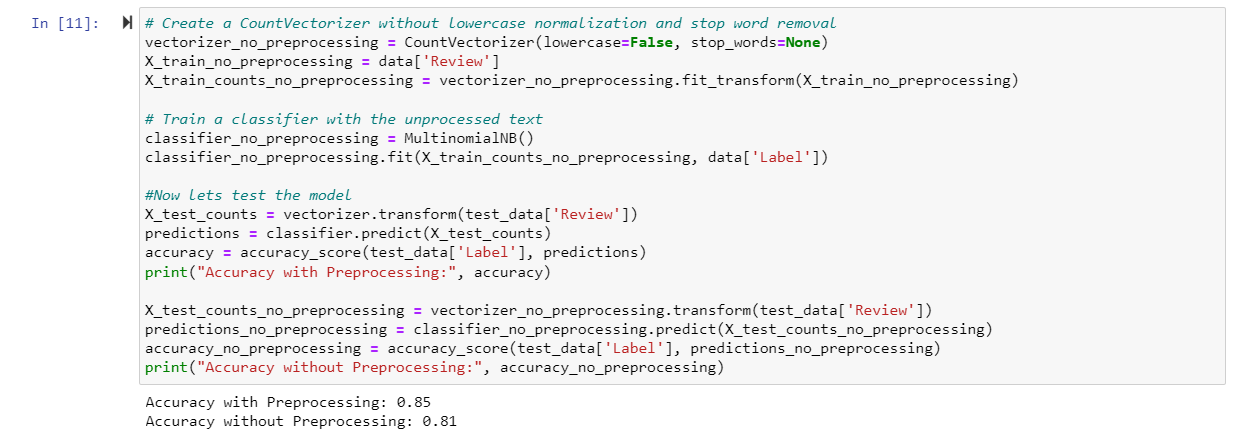


1. The performance removing stop words vs. without removing them. Are they different? Why is that? •



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.

1. 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?



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

* can be computed by counting the amount the bigram appears in the dataset and divide the amount by the amount . The final result after smoothing will look like that:
* 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:

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

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

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

Which equals:

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

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

The same is for “terrible failure”:

|  |  |
| --- | --- |
| Count(terrible failure) | 0 |
| Count(terrible) | 1 |
| Vocabulary (v) | 28 |

## 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.