Natural Language Processing (NLP)

NLP is a fascinating field at the intersection of computer science and linguistics, and it's a key component of many of the technologies we use every day, from search engines to virtual assistants.

In this Article, we'll dive into the core concepts of NLP, explore various techniques, and see how we can apply them to real-world problems. Whether you're a seasoned data scientist or just starting out, there's something here for you.

- LinkedIn
- YouTube
- gtihub
- Gmail
- discord

Introduction

Natural Language Processing (NLP) is a subfield of artificial intelligence that focuses on the interaction between computers and humans using natural language. It's a fascinating field that has seen rapid growth in recent years, thanks to advances in machine learning and deep learning.

Some Real World Applications of NLP are:

- `Chatbots`
- `CHAT GPT`
- `Copilot`

> Even 30% of this article was written by Copilot 😁

What is NLP

`Natural Language Processing (NLP)` is a very broad field that encompasses a wide range of tasks, from simple text processing to complex language understanding. It is concerned with the interaction between computers and humans using natural language.

Suppose You work for a `customer service department` and you receive hundreds of `emails` every day. It would be impossible to read and respond to each one manually.

Or you are a `doctor` and you have to go through hundreds of `medical records` to `diagnose` a patient. It would be very time-consuming and error-prone for you to do this manually.

There are Hundreds of such examples where if it's done manually it would be very time-consuming and error-prone. This is where `NLP` comes in. It can help you `automate` these tasks and make your life easier.

`NLP` can help you `extract information` from `text`, `classify` text into different categories, `summarize` text, `translate` text from one language to another, and much more.

So, think of the first scenario where you receive hundreds of emails every day. You can use `NLP` to `automatically read` and `classify` these emails into different categories. This way, you can `prioritize` which emails to respond to first and which ones to respond to later. How this happens:

- `Compile` all the emails into a single document.
- `Featurize` the text data, meaning you would want to convert the text data into a format that can be used by a machine learning model.
- `Compare` the Features of the text data to a set of predefined categories.

These are the basic steps involved in `NLP` but there are many more advanced techniques that can be used to `extract information` from text data.

How does NLP work?

Here's a simple example to illustrate how `NLP` works:

Suppose you have two `documents`:

- Document 1: "Bob Likes Apples"
- Document 2: "Sam Likes Oranges"

You want to `compare` these two documents to see if they are `similar` or `different`. You can:

- `Tokenize` the documents, meaning we would split the documents into individual words. So, the tokenized version of the documents would be:
    - Document 1: ["Bob", "Likes", "Apples"]
    - Document 2: ["Sam", "Likes", "Oranges"]
- `Vectorize` the documents, meaning we would convert the words into numbers. We can use a technique called `Bag of Words` to do this.

    > `Bag of Words` is a simple technique that converts text data into a matrix of word counts. Each row in the matrix represents a document, and each column represents a word. The value in each cell represents the count of the word in the document.

So, we compile all the words in the documents into a single list:

`["Bob", "Sam", "Likes", "Apples", "Oranges"]`

Now, we can convert the documents into vectors:

- Document 1:

```
"Bob Likes Apples"

->  ["Bob": 1,
    "Sam": 0,
    "Likes": 1,
    "Apples": 1,
    "Oranges": 0]

-> [1, 0, 1, 1, 0]
```

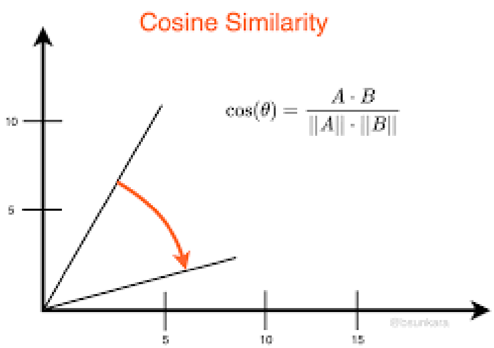
- Document 2:

```
"Sam Likes Oranges"

-> ["Bob": 0,
    "Sam": 1,
    "Likes": 1,
    "Apples": 0,
    "Oranges": 1]

-> [0, 1, 1, 0, 1]
```

Now, We have a fully `vectorized` version of each document. We can now `compare` these vectors to see if they are `similar` or `different`. This is very useful for `document classification` because we are treating the documents as `vectors` of `features`. SO, we can perform `mathematical operations` like `dot products` and `cosine similarity` to compare the documents.

Now, I'm not going to go deep into the `mathematical details` of how these operations work, GO DO YOUR OWN RESEARCH 😁

> `COSINE SIMILARITY` is a the `dot product` of two vectors `divided` by the `product` of the `magnitude` or `length` of the two vectors from the `origin`.



The `equation` for `cosine similarity` is:

$$\text{Cosine Similarity} = \frac{A \cdot B}{||A|| \times ||B||}$$

Where `A` and `B` are the two vectors(vectorized documents) and `||A||` and `||B||` are the magnitudes of the two vectors.

I can also re-write the `equation` as:

$$\text{Cosine Similarity} = \frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \times \sqrt{\sum_{i=1}^{n} B_i^2}}$$

We can use `cosine similarity` to `compare` the `similarity` of two `documents`. If the `cosine similarity` is `close to 1`, then the documents are `similar`. If the `cosine similarity` is `close to 0`, then the documents are `different`.

We can also imporve the `Bag of Words` model by adjusting the `word counts` based on the `frequency` of the words in the `corpus`.(A `corpus` is a collection of `documents`)

- `TF-IDF` (Term Frequency-Inverse Document Frequency) is a technique that does this. It `weights` the `word counts` based on the `frequency` of the words in the `corpus`.

`TF(Term Frequency)` is the `importance` of the `term` or `word` in the `document`. It is calculated as the `number of times` the `term` appears in the `document`. We represent it as:

$$\text{TF(d, t)} = \frac{\text{Number of times t appears in d}}{\text{Total number of words in d}}$$

> `t` stands for `term` and `d` stands for `document`.

`IDF(Inverse Document Frequency)` is the `importance` of the `term` or `word` in the `corpus` meaning `all the documents`. It also means how `rare` the `term` is in the `corpus`. It is calculated as the `logarithm` of the `total number of documents` divided by the `number of documents` that contain the `term`. We represent it as:

$$\text{IDF(t)} = \log(\frac{\text{D}}{\text{dt}})$$

Where `D` is the `total number of documents` and `dt` is the `number of documents` that contain the `term`.

`TF-IDF` is calculated as the `product` of `TF` and `IDF`. It `weights` the `word counts` based on the `frequency` of the words in the `corpus`.

$$\text{W(x, y)} = \text{TF(x, y)} \times \text{IDF(x)}$$

`TF-IDF` is a very powerful technique that can help you `extract important information` from `text data`. We do this to get, not just the `word counts`, but the `importance` of the `words` in the `document`.

This is just a `brief overview` of how `NLP` works. There are many more `advanced techniques` that can be used to `extract information` from `text data`.

Natural Language Processing using Python

Now that we have a basic understanding of `NLP`, let's see how we can use `Python` to `perform NLP` tasks. We'll use the `Natural Language Toolkit (NLTK)` library, which is a `popular library` for `NLP` in `Python`.

We have to `install` the `NLTK` library first. We can do this using the following command:

```
conda install nltk # If you are using Anaconda

pip install nltk # If you are using pip
```

In this Article, I'll show you the workings of `NLP` using the `NLTK` library and build a `spam filter` using. In this process, we'll learn about `tokenization`, `stemming`, `lemmatization`, and `TF-IDF`.

Let's get started!

NLTK Basics

The `Natural Language Toolkit (NLTK)` is a `popular library` for `NLP` in `Python`. It provides a wide range of tools and resources for `text processing` and `analysis`. I hope you have already installed the `NLTK` library.

Lets import the `NLTK` library and `download` some `resources`:

In [1]:
```python
import nltk
```

Before going to the code, let me give you and overview. I'll use `nltk.download_shell()` to show you how to `download` the `resources`. You can `download` the `resources` and `corpora` that you need for your `NLP` tasks.

This method will open a `shell` where you can `download` the `resources` and `corpora` that you need. You can `download` the `resources` by `selecting` the `number` of the `resource` you want to `download`.

The shell will give you choices like:

- `d` to `download` the `resource`
- `q` to `quit` the `shell`

- `l` to `list` the `resources`
- `u` to `update` the `resources`

SO, let's download the `resources` and `corpora` named `stopwords`.

In [2]:
```
# nltk.download_shell()
```

Here I have downloaded the `stopwords` `corpora` using the `nltk.download_shell()` method. You can `download` the `resources` and `corpora` that you need for your `NLP` tasks too.

Now, for info I'll use a dataset from `UCI Machine Learning Repository` named `SMS Spam Collection`. This dataset contains `SMS` messages that are `labeled` as `spam` or `ham` (not spam). We'll use this dataset to build a `spam filter` using `NLP`.

You can `download` the `dataset` from this link (https://archive.ics.uci.edu/dataset/228/sms+spam+collection).

And I also have the dataset in my `GitHub` repository.

Now, let's start making the `spam filter` using `NLP`.

What Kind of Data are we dealing with?

So, we have a `dataset` that contains `SMS` messages that are `labeled` as `spam` or `ham` (not spam). But we need to `explore` the `data` first to see what kind of `data` we are dealing with and how we can `process` it.

So, I'll simply `read` the `dataset` using `open()` method and `print` the `first few lines` just to get a `glimpse` of the `data` and I hope we can `understand` the `data` better.

In [3]:
```
# taking a line from the file and adding it to a list

with open('SMSSpamCollection') as f:
    messages = f.readlines()
    messages = [line.rstrip() for line in messages]
```

In [4]:
```
len(messages)
```

Out[4]:
```
5574
```

So, you can see that the dataset has `5574` `SMS` messages that are `labeled` as `spam` or `ham` (not spam). Let's see the `first few lines` of the `data` to get a `glimpse` of the `data`.

In [5]:
```
messages[0]
```

Out[5]:
```
'ham\tGo until jurong point, crazy.. Available only in bugis n great worl
d la e buffet... Cine there got amore wat...'
```

Well this does not look good. there is a `\t` in the `data`, what does that mean? Let's `explore` the `data` further to see what's going on.

> I'm dumbing down the code here, so that anyone can understand it.

let's print the `first few lines` of the `data` to see what's going on.

In [6]:
```
for mess_no, msg in enumerate(messages[:10]):
    print(mess_no, msg)
```

```
0 ham	Go until jurong point, crazy.. Available only in bugis n great worl
d la e buffet... Cine there got amore wat...
1 ham	Ok lar... Joking wif u oni...
2 spam	Free entry in 2 a wkly comp to win FA Cup final tkts 21st May 2005.
Text FA to 87121 to receive entry question(std txt rate)T&C's apply 0845281
0075over18's
3 ham	U dun say so early hor... U c already then say...
4 ham	Nah I don't think he goes to usf, he lives around here though
5 spam	FreeMsg Hey there darling it's been 3 week's now and no word back!
I'd like some fun you up for it still? Tb ok! XxX std chgs to send, £1.50 t
o rcv
6 ham	Even my brother is not like to speak with me. They treat me like ai
ds patent.
7 ham	As per your request 'Melle Melle (Oru Minnaminunginte Nurungu Vetta
m)' has been set as your callertune for all Callers. Press *9 to copy your
friends Callertune
8 spam	WINNER!! As a valued network customer you have been selected to rec
eivea £900 prize reward! To claim call 09061701461. Claim code KL341. Valid
12 hours only.
9 spam	Had your mobile 11 months or more? U R entitled to Update to the la
test colour mobiles with camera for Free! Call The Mobile Update Co FREE on
08002986030
```

Well that is interesting. The `print` statement is `splitting` the `data` into `two columns` based on the `tab` character. SO, now we know what the `\t` means. It is the `tab` character that `separates` the `label` from the `message`.

Which means if we make a `dataframe` using this `data`, we have to `split` the `data` into `two columns` based on the `tab` character.

In [7]:
```
import numpy as np
import pandas as pd

data = pd.read_csv('SMSSpamCollection', sep='\t', names=['label', 'message
data.head()
```

Out[7]:

	label	message
0	ham	Go until jurong point, crazy.. Available only ...
1	ham	Ok lar... Joking wif u oni...
2	spam	Free entry in 2 a wkly comp to win FA Cup fina...
3	ham	U dun say so early hor... U c already then say...
4	ham	Nah I don't think he goes to usf, he lives aro...

Here we have a `dataframe` that contains `two columns` named `label` and `message`. The `label` column contains the `labels` of the `SMS` messages (spam or ham) and the `message` column contains the `text` of the `SMS` messages.

Now, time for further `exploration` of the `data`.

In [8]:
```
data.describe()
```

Out[8]:

	label	message
count	5572	5572
unique	2	5169
top	ham	Sorry, I'll call later
freq	4825	30

We can see that the dataset has `5574` `SMS` messages that are `labeled` as `spam` or `ham` (not spam).

Now, we can see that the `top` `message` is a `ham` message `sorry, I'll call later`. This is the `most frequent` `message` in the `data`.

I want to know the top `spam` message too. It's because I want to see what kind of `spam` messages are in the `data`.

In [9]:
```
data.groupby('label').describe()
```

Out[9]:

	message			
	count	**unique**	**top**	**freq**
label				
ham	4825	4516	Sorry, I'll call later	30
spam	747	653	Please call our customer service representativ...	4

Feature Engineering

A big part of `Machine Learning` is `feature engineering`. It's the process of `creating new features` from the `existing features` which can help the `machine learning model` to `learn` better.

Feature engineering is a very important step in the `machine learning pipeline`. It can help you `improve` the `performance` of your `machine learning model` and `extract important information` from the `data`.

The more domain knowledge you have, the better you can `engineer features` from the `data`. You can `create new features` from the `existing features` like I can `create` a `new feature` from the `message` column that contains the `length` of the `message`.

In [10]:
```
data['length'] = data['message'].apply(len)
```

In [11]:
```
data.head()
```

Out[11]:

	label	message	length
0	ham	Go until jurong point, crazy.. Available only ...	111
1	ham	Ok lar... Joking wif u oni...	29
2	spam	Free entry in 2 a wkly comp to win FA Cup fina...	155
3	ham	U dun say so early hor... U c already then say...	49
4	ham	Nah I don't think he goes to usf, he lives aro...	61

And now we have an extra column named `length` that contains the `length` of the `message`. This is a `new feature` that we have `engineered` from the `existing features`.

Now time to `visualize` the `data` to see if we can `extract` any `important information` from the `data`.

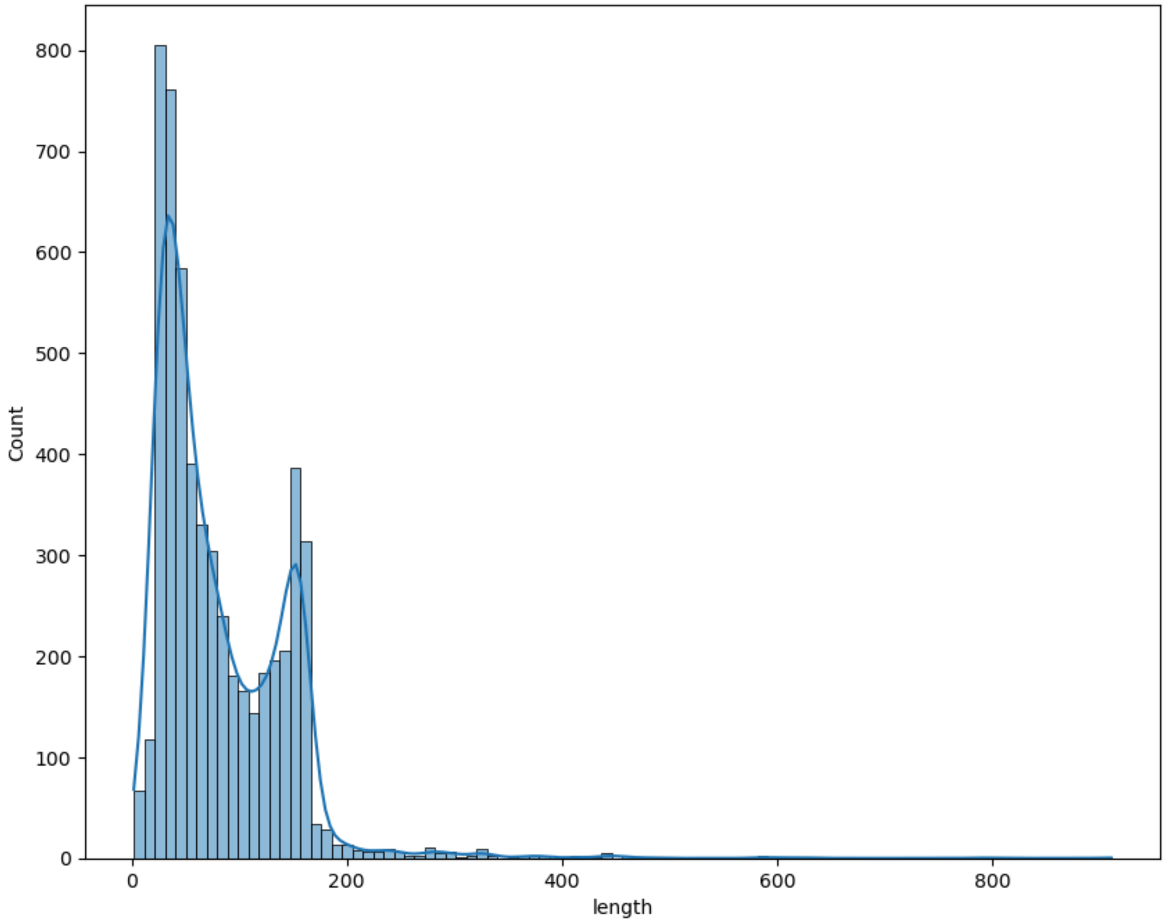
To do that I'll start with importing the `famous` `plotting` library `matplotlib` and `seaborn` for `data visualization`.

In [12]:
```
import matplotlib.pyplot as plt
import seaborn as sns
```

Now we can do some `data visualization`.

In [13]:
```
plt.figure(figsize=(10, 8))
sns.histplot(data=data, x='length', kde=True)
```

Out[13]:
```
<Axes: xlabel='length', ylabel='Count'>
```



> I suggest you play with the `bins` and `kde` values to see how the `distribution` of the `length` of the `messages` changes.

In my `plot` you can see that the destribution of the `length` of the `messages` is `right-skewed`. This means that most of the `messages` are `short` and only a few `messages` are `long` but there is an exeption too. The has a `bi-modal` distribution going on.

Meaning there are text massages that are quite `long`. Let's see some details about the `long` `messages`.

In [14]:
```
data['length'].describe()
```

Out[14]:
```
count    5572.000000
mean       80.489950
std        59.942907
min         2.000000
25%        36.000000
50%        62.000000
75%       122.000000
max       910.000000
Name: length, dtype: float64
```

Well, that's interesting. The `longest` `message` in the `data` is `910` `characters` long, that's a very long `message`, I wonder if it was from a girl, if that was the case, then I feel sorry for the guy 😁

You know what? I really want to see the `longest` `message` in the `data`.

In [15]:
```
print(data[data['length'] == 910]['message'].iloc[0])
```

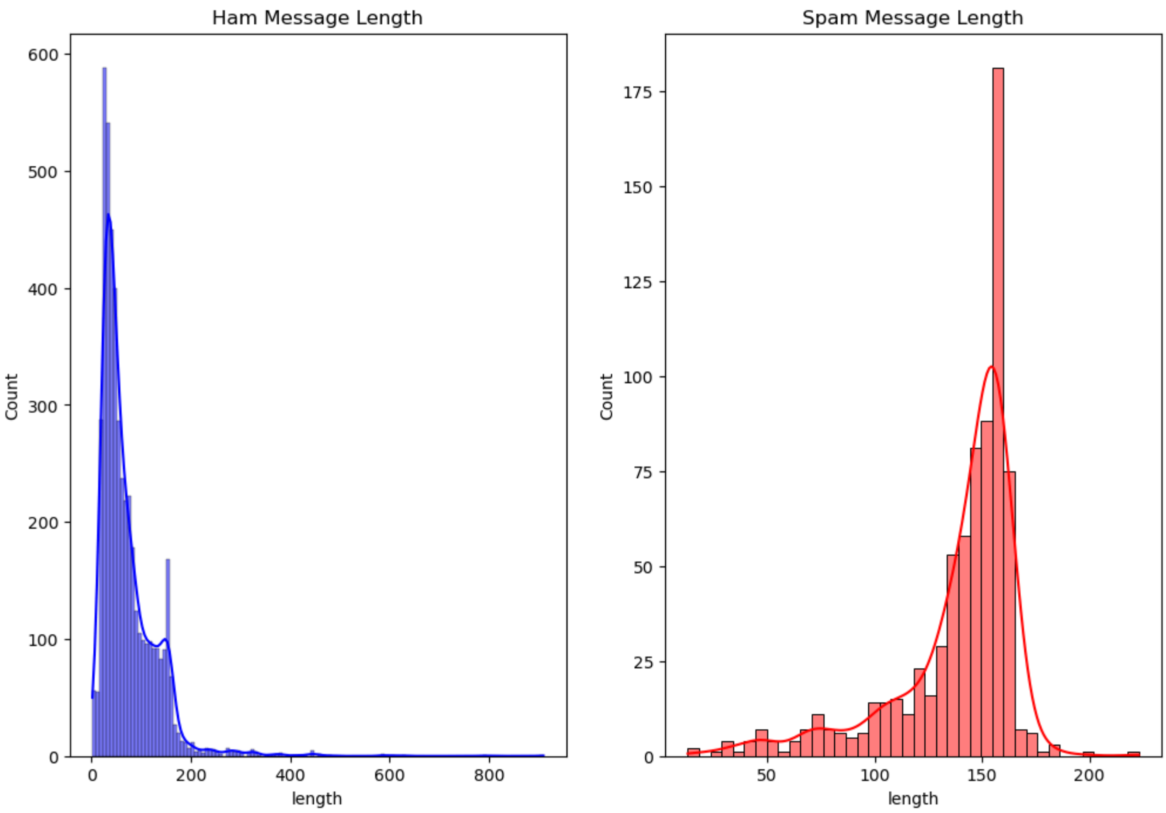
```
For me the love should start with attraction.i should feel that I need her
every time around me.she should be the first thing which comes in my though
ts.I would start the day and end it with her.she should be there every time
I dream.love will be then when my every breath has her name.my life should
happen around her.my life will be named to her.I would cry for her.will giv
e all my happiness and take all her sorrows.I will be ready to fight with a
nyone for her.I will be in love when I will be doing the craziest things fo
r her.love will be when I don't have to proove anyone that my girl is the m
ost beautiful lady on the whole planet.I will always be singing praises for
her.love will be when I start up making chicken curry and end up makiing sa
mbar.life will be the most beautiful then.will get every morning and thank
god for the day because she is with me.I would like to say a lot..will tell
later..
```

I couldn't have been more wrong. The `longest` `message` is a `monologue` from `a random guy` about `love` and how he `feels` about it. 🥴

With that out of the way, Let's see another `histogram` of the `length` of the `messages` based on the `label` column. Let's see if we can get some `insights` from the `data`.

In [16]:
```
plt.figure(figsize=(12, 8))

plt.subplot(1, 2, 1)
sns.histplot(data[data['label'] == 'ham'], x='length', kde=True, color='bl
plt.title('Ham Message Length')

plt.subplot(1, 2, 2)
sns.histplot(data[data['label'] == 'spam'], x='length', kde=True, color='r
plt.title('Spam Message Length')
```

Out[16]:
```
Text(0.5, 1.0, 'Spam Message Length')
```



SO, we can see that the `spam` messages are `longer` than the `ham` messages. `Ham` messages are `short` almost at the `5-100` characters range. But the spam messages are long and are in the `120-180` characters range.

Now, time for the next step,

Text Preprocessing

`Text preprocessing` is a very important step in `NLP`. It involves `cleaning` and `transforming` the `text data` into a `format` that can be used by a machine learning model. As we have seen before(i'm refering to my other articles), `machine learning models` cannot work with `text` data `directly` because they are `mathematical models` that require `numerical input`.

But in our hand we have `string` text data. So, we have to `convert` the `text` data into a `format` that can be used by a machine learning model. This is where `text preprocessing` comes in.

In the theory section, I have talked about the `bag of words` model and the `TF-IDF` technique. These are the `techniques` that we can use to `convert` the `text data` into a `format` that can be used by a `machine learning model`.

Tokenization

As I have said before, `tokenization` is the process of `splitting` the text data into `individual words`. SO, let's `split` the `data`.

To do all that we will use the `built-in` `string` `methods` in `Python`. We will also use the `split()` method to `split` the `data` into `individual words`.

Let's me first demonstrate the `string` `methods` that we are going to use to `split` the `data`.

In [17]:
```
import string
```

In [18]:
```
# a sample message
mess = 'Sample message! Notice: it has punctuation.'
```

I have taken a sample string and I'll remove the `punctuation` from the `string` by using the `string.punctuation` which will return a `string` containing all the `punctuation` characters.

In [19]:
```
string.punctuation
```

Out[19]:
```
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

Now We use string comprehehension to `remove` the `punctuation` from the `string` and then use `join()` method to `join` the `characters` into a `single string` again.

In [20]:
```
no_punc = [char for char in mess if char not in string.punctuation]
# if the character is not in the string.punctuation then add it to the lis

print(f'Original message: {mess}')
print(f'After removing punctuation: {no_punc}')

# joining the list of characters to form a string
no_punc = ''.join(no_punc) # this is the way to join a list of characters

print(f'Complete message: {no_punc}')
```

```
Original message: Sample message! Notice: it has punctuation.
After removing punctuation: ['S', 'a', 'm', 'p', 'l', 'e', ' ', 'm', 'e',
's', 's', 'a', 'g', 'e', ' ', 'N', 'o', 't', 'i', 'c', 'e', ' ', 'i', 't',
' ', 'h', 'a', 's', ' ', 'p', 'u', 'n', 'c', 't', 'u', 'a', 't', 'i', 'o',
'n']
Complete message: Sample message Notice it has punctuation
```

Now we have a string that does not contain any punctuation.

Now, what we can do is split the string into individual words using the `split()` method. This will `split` the `string` into individual words `based on the` space` character.

In [21]:
```
no_punc.split()
```

Out[21]:
```
['Sample', 'message', 'Notice', 'it', 'has', 'punctuation']
```

Wel, that was easy right??? Wrong, this is just the `beginning`. We cannot just `split` the `data` into `individual words` and call it a day. We have to `clean` the `data` first.

By cleaning the data I mean removing the stopwords, lowercasing the data, and stemming the data.

So, we start by removing the `stopwords` from the `data`.

We downloaded the `stopwords` `corpora` before. Now we can use the `stopwords` to `remove` the `stopwords` from the `data`.

In [22]:
```
from nltk.corpus import stopwords

stopwords.words('english')[:10]
```

Out[22]:
```
['a', 'about', 'above', 'after', 'again', 'against', 'ain', 'all', 'am',
'an']
```

Here are the first few stopwords in the corpora. `stopwords` are `most common` words that do not `contribute` much to the `meaning` of the `text`. So, we can `remove` the `stopwords` from the `data` to `clean` the `data`.

Now, we make another list that will contain the `words` that are `not` in the `stopwords` `corpora`.

> Note: The words that are in the stopwords corpora are in lowercase, so we have to lowercase every word in the data to compare the words with the stopwords.

In [23]:
```
clean_msg = [word for word in no_punc.split() if word.lower() not in stopw

print(f'Original msg with stopwords: {no_punc.split()}')
print(f'Clean msg: {clean_msg}')
```

```
Original msg with stopwords: ['Sample', 'message', 'Notice', 'it', 'has',
'punctuation']
Clean msg: ['Sample', 'message', 'Notice', 'punctuation']
```

As you can see that `"it"` and `"has"` are `removed` from the `data` because they are `stopwords`.

We won't lowercase the `data` yet. Because we are not `vectorizing` the `data` yet. SO, we have to `lowercase` the `data` before `vectorizing` the `data`.

But now we have to think about how we can apply this cleaning to the whole dataset. We can use the `apply()` method to apply the cleaning to the whole dataset.

In [24]:
```
def text_process(msg):
    """
    1. Remove punctuation
    2. Remove stopwords
    3. Return list of clean text words
    """
    no_punc = [char for char in msg if char not in string.punctuation]
    no_punc = ''.join(no_punc)
    return [word for word in no_punc.split() if word.lower() not in stopwo
```

Now we apply!

In [25]:
```
data['message'].head(5).apply(text_process)
```

Out[25]:
```
0    [Go, jurong, point, crazy, Available, bugis, n...
1                        [Ok, lar, Joking, wif, u, oni]
2    [Free, entry, 2, wkly, comp, win, FA, Cup, fin...
3        [U, dun, say, early, hor, U, c, already, say]
4    [Nah, dont, think, goes, usf, lives, around, t...
Name: message, dtype: object
```

So, the `apply()` method `applies` the `cleaning` `function` to the `whole` `dataset` and in the end we have a `cleaned` `dataset`.

Now, for the `stemming` part...

`Stemming` is the process of `reducing` the `words` to their `root` or `base` form. For example, the `stem` of the word `running` is `run`. This can help us `reduce` the `dimensionality` of the data and improve the performance of the machine learning model.

But for our data set, `stemming` is not a good idea. Because the `stemmed` words might not make any sense in the context of the SMS messages and also in the messages we have words like `U`, `r`, `ok`, `lol` etc. which are not `real` words but `short forms` of real words. So, `stemming` will not be a good idea for our data.

> We will use the `text_process` function in the `vectorization` step.

Vectorization

Now, we have to `vectorize` the data using the bag of words model. We will use the `CountVectorizer` class from the `sklearn` library to `vectorize` the `data`.

Before diving into the vectorization part, let me give you a brief overview of the CountVectorizer class.

> The CountVectorizer class is a sklearn class that converts a collection of text documents into a matrix of token counts. Each row in the matrix represents a Word and each column represents a message. The value in each cell represents the count of the word in the message.

As the model transforms the `data` into a `matrix` of `word counts` and has a `column` for each `word` in the `data`, the `dimensionality` of the `data` will be `very high`. So, it makes a `sparse matrix` to `store the data`.

what is a `sparse matrix`?

A `sparse matrix` is a matrix where `most elements` are `zero`, making it efficient to `store and process` by only saving the `non-zero values`, unlike dense matrices where most elements are significant. this technique saves memory and computation time in fields like data science, scientific computing, and machine learning

> you can find a more detailed explanation here

In [26]:
```
from sklearn.feature_extraction.text import CountVectorizer
```

Now we can fit our messeges to the CountVectorizer class and `transform` the `messages` into a `matrix` of `word counts`.

> The CountVectorizer class has a lot of parameters that you can tune to improve the performance of the machine learning model but we will use the default parameters for now and pass our cleaner function to the CountVectorizer class as the analyzer parameter. This way the CountVectorizer class will clean the data before vectorizing the data using the cleaner function we built before.

In [27]:
```
#bag of words transformer
bow_transformer = CountVectorizer(analyzer=text_process).fit(data['message

print(f"Total number of vocabularies in the bag of words: {len(bow_transfo
```

```
Total number of vocabularies in the bag of words: 11425
```

This might take a while to `transform` the data into a matrix of word counts. So, be patient.

My data is `transformed` into a `matrix` of `word counts` and It has `11425` unique words in the `bag of words` even after `cleaning` the `data`.

Let's now explore the `bow_transformer` to see what we have `achieved`. Let's get the `4th` `message` from the `data` and `transform` it into a `matrix` of `word counts`.

In [28]:
```
mess4 = data['message'][3]
print(f'Message 4: {mess4}')
```

```
Message 4: U dun say so early hor... U c already then say...
```

In [29]:
```
bow4 = bow_transformer.transform([mess4])
print(bow4)
```

```
<Compressed Sparse Row sparse matrix of dtype 'int64'
	with 7 stored elements and shape (1, 11425)>
  Coords	Values
  (0, 4068)	2
  (0, 4629)	1
  (0, 5261)	1
  (0, 6204)	1
  (0, 6222)	1
  (0, 7186)	1
  (0, 9554)	2
```

In [30]:
```
bow4.shape
```

Out[30]:
```
(1, 11425)
```

SO, we have successfully transformed the 4th message into a matrix of word counts. You might get confused by the `numbers` in the `matrix` and the `shape` of the `matrix`. But don't worry, I'll explain it to you.

the `shape` of the matrix is `(1, 11425)`. This means that the `matrix` has `1 row` and `11425 columns`. The row represents the `message` and the `columns` represent the `words` in the `data`.

We see the output in something like this.

```
(0, 4068)      2
(0, 4629)      1
(0, 5261)      1
(0, 6204)      1
(0, 6222)      1
(0, 7186)      1
(0, 9554)      2
```

This is a `sparse matrix` representation of the `message`. The `numbers` in the `matrix` represent the `count` of the `word` in the `message`. For example, the `word` at index `4068` appears `2` times in the `message`.

this index is the index of the word in the transformed data we vectorized before.

So, if you look carefully you can see that there are `7` `words` in the `message` and the `word` at `index` `4068` appears `2` times in the `message`.

So, let's see if it's true or not.

In [31]:
```
bow_transformer.get_feature_names_out()[4068]
```

Out[31]: 'U'

> We can use the `get_feature_names()` method to get the whole list of `words` in the `transformed` `data`.

There you go..

We got the word at index `4068` is `U` and it appears `2` times in the message.

So, we can be assured that the `CountVectorizer` class has transformed the data into a matrix of word counts successfully.

Well, we have done a lot of work so far. We have `cleaned the data`, `transformed the data` into a `matrix` of `word counts`, and `vectorized` the `data` (not complete yet). Now what we can do is `TF-IDF` the `data`.

TF-IDF

`TF-IDF` (Term Frequency-Inverse Document Frequency) is a `technique` that `weights` the `word counts` based on the `frequency` of the `words` in the `corpus`. It is a very `powerful technique` that can help you `extract important information` from `text data`.

I have already talked about the `TF-IDF` technique in the `theory` section. So, I'll not go into the details of the TF-IDF technique. But I'll show you how to TF-IDF the data using the `TfidfTransformer` class from the `sklearn` library.

But before that we need to get the `sparse matrix` of the `word counts` that we got from the `CountVectorizer` class. We can use the `bow_transformer` to `transform` the data into a sparse matrix of word counts.

In [32]:
```
msg_bow = bow_transformer.transform(data['message'])
print(f"shape of the matrix: {msg_bow.shape}")
```

```
shape of the matrix: (5572, 11425)
```

And we have the `sparse matrix` of the word counts that we got from the CountVectorizer class. The shape of the matrix is `(5574, 11425)`. This means that the `matrix` has `5574 rows` and `11425 columns`.

The `rows` represent the `messages` and the `columns` represent the `words` in the `data`.

As this is a `sparse matrix`, it does not `store` the `zero elements`. So, we have a lot of `non-zero elements` in the `matrix`. Let's see it

In [33]:
```
msg_bow.nnz # number of non-zero occurences
```

Out[33]:
```
50548
```

Well, that's a lot of non-zero elements in the `matrix`, almost `51k` non-zero elements. This is because the messages has a lot of unique words.

We can calculated the `sparsity` of the `matrix` by dividing the `number` of `non-zero elements` by the `total number of elements` in the `matrix`.

In [34]:
```
sparsity = (100.0 * msg_bow.nnz / (msg_bow.shape[0] * msg_bow.shape[1]))
print(f"Sparsity: {sparsity}")
```

```
Sparsity: 0.07940295412668218
```

Sparsity tells us how `sparse` the `matrix` is.

In this case, the `sparsity` of the `matrix` is `0.0794`. This means that the `matrix` has `7.94%` `non-zero` `elements`. This is very `efficient` in terms of `memory` because it does not store the zero elements.

Now we use the `TfidfTransformer` class to `TF-IDF` the data. We will `fit` the `sparse matrix` of the word counts to the `TfidfTransformer` class and `transform` the `sparse matrix` into a matrix of `TF-IDF` values.

In [35]:
```
from sklearn.feature_extraction.text import TfidfTransformer

tfidf = TfidfTransformer().fit(msg_bow)
```

```python
# transforming the bag of words
tfidf4 = tfidf.transform(bow4) # passing the bag of words of message 4

print(f"TFIDF of message 4: \n{tfidf4}")

# we can also check the idf of a particular word
print(f"IDF of the word 'university': {tfidf.idf_[bow_transformer.vocabula
```

```
TFIDF of message 4:
<Compressed Sparse Row sparse matrix of dtype 'float64'
	with 7 stored elements and shape (1, 11425)>
  Coords	Values
  (0, 4068)	0.4083258993338407
  (0, 4629)	0.2661980190608719
  (0, 5261)	0.2972995740586873
  (0, 6204)	0.2995379972369742
  (0, 6222)	0.31872168929491496
  (0, 7186)	0.4389365653379858
  (0, 9554)	0.5385626262927565
IDF of the word 'university': 8.527076498901426
```

We can see the `Tf-Idf` `values` of the `4th` `message` in the `data`. The `Tf-Idf` `values` are `normalized` and `weighted` based on the `frequency` of the `words` in the `corpus`.

Let's transform the `whole` `data` into a `matrix` of `TF-IDF` `values`.

In [36]:
```python
msg_tfidf = tfidf.transform(msg_bow)
```

Now, we can use any machine learning model to classify the messages as `spam` or `ham` now.

But in this case I'll use `Naive Bayes classifier` to `classify` the messages as `spam` or `ham`. I'll use the `MultinomialNB` class from the `sklearn` library.

In [37]:
```python
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import MultinomialNB

X_train, X_test, y_train, y_test = train_test_split(data['message'], data[
```

We've split the data into training and testing sets.

Now we need to clean the data and transform it into a word count matrix using CountVectorizer. Then we'll apply TF-IDF transformation using TfidfTransformer.

Finally, we'll use MultinomialNB to classify the messages as spam or ham. To automate this process, we'll use sklearn's pipeline feature.

The pipeline takes a `list of tuples`, where each tuple contains the name of the step and the transformer or estimator to be chained in the pipeline. The pipeline will have three steps: **CountVectorizer**, **TfidfTransformer**, and **MultinomialNB**.

In [38]:
```python
from sklearn.pipeline import Pipeline
pipeline = Pipeline([
    ('bow', CountVectorizer(analyzer=text_process)), # bow is the name of
```

```
    ('tfidf', TfidfTransformer()), # tfidf is the name of the step and Tfi
    ('classifier', MultinomialNB()) # classifier is the name of the step a
])
```

Now, we can directly `fit` the `pipeline` to the `data` and `predict` the `output`.

In [39]:
```
pipeline.fit(X_train, y_train)
```

Out[39]:
```
▸ Pipeline
  ▸ CountVectorizer
  ▸ TfidfTransformer
  ▸ MultinomialNB
```

And we have `trained` the `pipeline` on the training data and predicted the output on the testing data using the multinomialNB classifier.

Now, we can use the `classification_report` and `confusion_matrix` to `evaluate` the `performance` of the `classifier`.

In [40]:
```
predictions = pipeline.predict(X_test)
```

In [41]:
```
from sklearn.metrics import classification_report, confusion_matrix

print(classification_report(y_test, predictions))
print('\n')
print(confusion_matrix(y_test, predictions))
```

```
              precision    recall  f1-score   support

         ham       0.96      1.00      0.98      1451
        spam       1.00      0.73      0.84       221

    accuracy                           0.96      1672
   macro avg       0.98      0.86      0.91      1672
weighted avg       0.97      0.96      0.96      1672



[[1451    0]
 [  60  161]]
```

Looks pretty good!

We can also use any other machine learning model. Let's say we want to use the `RandomForestClassifier`.

In [42]:
```
from sklearn.ensemble import RandomForestClassifier

rf = Pipeline([
    ('bow', CountVectorizer(analyzer=text_process)),
    ('tfidf', TfidfTransformer()),
    ('classifier', RandomForestClassifier(n_estimators=1000))
```

```
])

rf.fit(X_train, y_train)

rf_predictions = rf.predict(X_test)
```

In [43]:

```
print(classification_report(y_test, rf_predictions, digits=4))
print('\n')
print(confusion_matrix(y_test, rf_predictions, ))
```

```
              precision    recall  f1-score   support

         ham     0.9667    1.0000    0.9831      1451
        spam     1.0000    0.7738    0.8724       221

    accuracy                         0.9701      1672
   macro avg     0.9833    0.8869    0.9278      1672
weighted avg     0.9711    0.9701    0.9684      1672



[[1451    0]
 [  50  171]]
```

And the results are infront of you!

Here's a task for you.

> Use cross validation and grid search to find the best hyperparameters for the Random Forest classifier of the pipeline.

Final Words

Writting this article took a lot of time and I'm tired af.