Natural Language Processing (NLP)

(ML perspective)

Motivation

🔢 So far, we have been running Machine Learning algorithms with:

- *numerical inputs*
- *(encoded) categorical inputs*

🗣️ How can we incorporate **textual data** in these Machine Learning Algorithms?

🤔 What are the ML models dedicated to language-related tasks?

🚀 Thanks to the development of NLP libraries, NLP is finding applications on an industrial level

Examples:

- E-mail filtering (legitimate e-mail vs. spam)
- Sentiment analysis
- Chatbots
- Voice/speech recognition
- Smart assistants
- Language translation

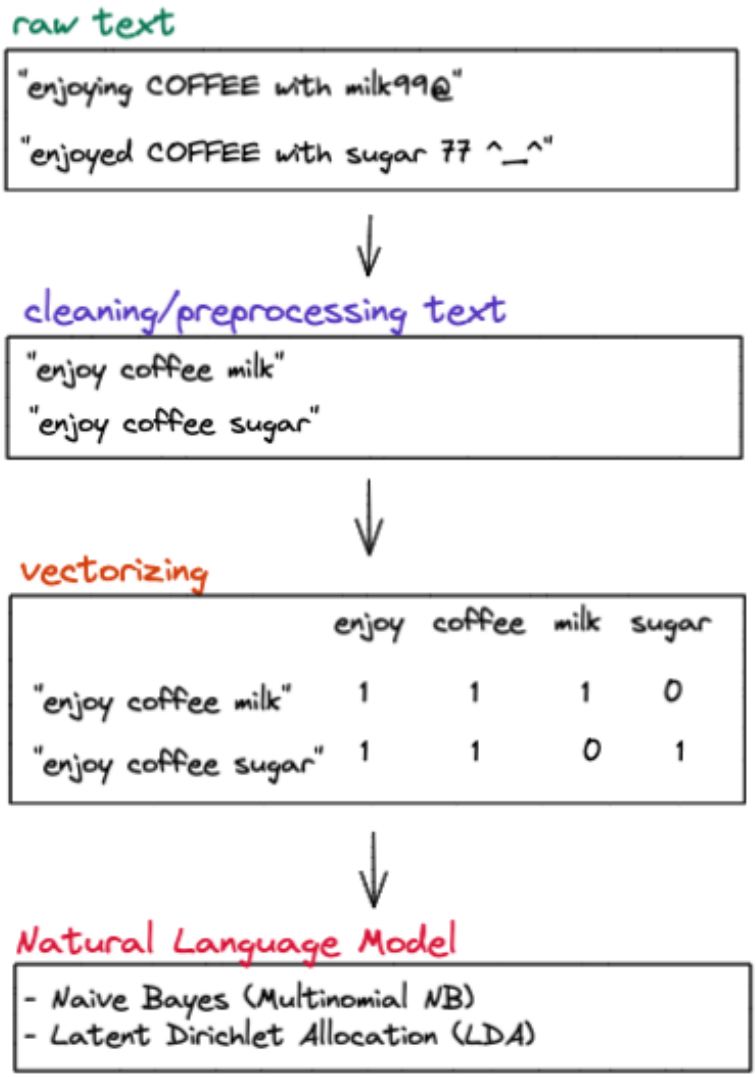
What is NLP?

> **Natural Language Processing (NLP)** is a subfield of linguistics, computer science and artificial intelligence concerned with the interactions between computers and human language - in particular how to program computers to process and analyze large amounts of natural language data (***speech*** AND ***text***).

Source: [Wikipedia (https://en.wikipedia.org/wiki/Natural_language_processing)](https://en.wikipedia.org/wiki/Natural_language_processing)

Plan

1. Text Preprocessing
2. Vectorizing
3. NLP Modeling: Naive Bayes Classifier
4. Topic Modeling: the Latent Dirichlet Allocation Algorithm (LDA) (Unsupervised)



1. Text Preprocessing

🧐 For any Machine Learning algorithm, data preprocessing is crucial, and this remains true for algorithms dealing with text

✍️ Text preprocessing is quite different from numerical preprocessing. The most common preprocessing tasks for textual data are:

- lowercase
- dealing with numbers, punctuation, and symbols
- splitting
- tokenizing
- removing "stopwords"
- lemmatizing

💻 🧹 Basic cleaning with Python core string operations

When you have some unstructured text, you can already clean it with some **Python built-in string operations**

💻 ✂️ strip (https://docs.python.org/3/library/stdtypes.html?highlight=strip#str.strip) (1/2)

`strip` removes all the whitespaces at the beginning and the end of a string

In [ ]:

```
texts = [
    '   Bonjour, comment ca va ?     ',
    '    Heyyyyy, how are you doing ?   ',
    '        Hallo, wie gehts ?     '
]
texts
```

Out[ ]:

```
['   Bonjour, comment ca va ?     ',
 '    Heyyyyy, how are you doing ?   ',
 '        Hallo, wie gehts ?     ']
```

In [ ]:

```
[text.strip() for text in texts]
```

Out[ ]:

```
['Bonjour, comment ca va ?',
 'Heyyyyy, how are you doing ?',
 'Hallo, wie gehts ?']
```

💻 ✂️ strip (https://docs.python.org/3/library/stdtypes.html?highlight=strip#str.strip) (2/2)

You can also specify a "list" of characters (in the form of a *single and unordered string*) to be removed at the beginning and at the end of a string

In [ ]:

```
text = "abcd Who is abcd ? That's not a real name!!! abcd"
text
```

Out[ ]:

```
"abcd Who is abcd ? That's not a real name!!! abcd"
```

In [ ]:

```
text.strip('bdac')
```

Out[ ]:

```
" Who is abcd ? That's not a real name!!! "
```

💻 👥 replace (https://docs.python.org/3/library/stdtypes.html?highlight=str%20replace#str.replace)

In [ ]:

```
text = "I love koalas, koalas are the cutest animals on Earth."
text
```

Out[ ]:

```
'I love koalas, koalas are the cutest animals on Earth.'
```

In [ ]:

```
text.replace("koala", "panda")
```

Out[ ]:

```
'I love pandas, pandas are the cutest animals on Earth.'
```

💻 🪛 split (https://docs.python.org/3/library/stdtypes.html?highlight=str%20split#str.split)

In [ ]:

```
text = "linkin park / metallica /red hot chili peppers"
```

In [ ]:

```
text.split("/")
```

Out[ ]:

```
['linkin park ', ' metallica ', 'red hot chili peppers']
```

💻 🔡 Lowercase

Text modeling algorithms are ***case-sensitive***. Two words need to have the same casing to be considered equal.

In [ ]:

```
text = "i LOVE football sO mUch. FOOTBALL is my passion. Who else loves fOOtBaLL ?"
text
```

Out[ ]:

```
'i LOVE football sO mUch. FOOTBALL is my passion. Who else loves fOO
tBaLL ?'
```

In [ ]:

```
text.lower()
```

Out[ ]:

```
'i love football so much. football is my passion. who else loves foo
tball ?'
```

💻 🔢 Numbers

✅ We can (and often should) remove numbers during the text preprocessing steps, especially for:

- text clustering
- collecting keyphrases

In [ ]:

```
text = "i do not recommend this restaurant, we waited for so long, like 30 minut
es, this is ridiculous"
text
```

Out[ ]:

```
'i do not recommend this restaurant, we waited for so long, like 30
minutes, this is ridiculous'
```

In [ ]:

```
cleaned_text = ''.join(char for char in text if not char.isdigit())
cleaned_text
```

Out[ ]:

```
'i do not recommend this restaurant, we waited for so long, like  mi
nutes, this is ridiculous'
```

💻 ❗ ❓ Punctuation and Symbols

- Punctuation like ".?!" and symbols like "@#$" are not useful for topic modeling.
- Punctuation is barely used properly on social media platforms.

Warning: you might want to keep punctuation and symbols for authorship attribution!

In [ ]:

```
text = "I love bubble tea! OMG so #tasty @channel XOXO @$ ^_^ "
text
```

Out[ ]:

```
'I love bubble tea! OMG so #tasty @channel XOXO @$ ^_^ '
```

In [ ]:

```
import string # "string" module is already installed with Python
string.punctuation
```

Out[ ]:

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

In [ ]:

```
for punctuation in string.punctuation:
    text = text.replace(punctuation, '')

text
```

Out[ ]:

```
'I love bubble tea OMG so tasty channel XOXO   '
```

💻 💪 Combo: `strip` + `lowercase` + `numbers` + `punctuation/symbols`

In [ ]:

```
sentences = [
    "   I LOVE Pizza 999 @^_^",
    "  Le Wagon is amazing, take care - 666"
]
```

In [ ]:

```
def basic_cleaning(sentence):
    sentence = sentence.lower()
    sentence = ''.join(char for char in sentence if not char.isdigit())

    for punctuation in string.punctuation:
        sentence = sentence.replace(punctuation, '')

    sentence = sentence.strip()

    return sentence
```

In [ ]:

```
cleaned = [basic_cleaning(sentence) for sentence in sentences]
cleaned
```

Out[ ]:

```
['i love pizza', 'le wagon is amazing take care']
```

💻 🔍 Removing Tags with RegEx

We can remove `HTML` tags using RegEx (https://regexr.com/):

In [ ]:

```
import re

text = """<head><body>Hello Le Wagon!</body></head>"""
cleaned_text = re.sub('<[^<]+?>','', text)

print (cleaned_text)
```

```
Hello Le Wagon!
```

We can also extract e-mail addresses from a text:

In [ ]:

```
import re

txt = '''
    This is a random text, authored by darkvador@gmail.com
    and batman@outlook.com, WOW!
'''

re.findall('[\w.+-]+@[\w-]+\.[\w.-]+', txt)
```

Out[ ]:

```
['darkvador@gmail.com', 'batman@outlook.com']
```

💻 Cleaning with NLTK

Natural Language Toolkit (NLTK) is an **NLP library** that provides preprocessing and modeling tools for text data

📚 NLTK official website (https://www.nltk.org/)

🛠️ Installation Documentation (https://www.nltk.org/install.html)

💻 🎄 Tokenizing

- Tokenizing is essentially **splitting** a sentence, a paragraph, or even an entire piece of text into smaller chunks such as **individual words** called **tokens**.
  - *"Natural Language Processing"* → *["Natural","Language","Processing"]*

📚 **nltk.tokenize** (https://www.nltk.org/api/nltk.tokenize.html)

🔅 Here is a quote from Aristotle:

In [ ]:

```
text = 'It is during our darkest moments that we must focus to see the light'

text
```

Out[ ]:

```
'It is during our darkest moments that we must focus to see the ligh
t'
```

In [ ]:

```
from nltk.tokenize import word_tokenize

word_tokens = word_tokenize(text)
print(word_tokens) # print displays the words in one line
```

```
['It', 'is', 'during', 'our', 'darkest', 'moments', 'that', 'we', 'm
ust', 'focus', 'to', 'see', 'the', 'light']
```

💻🛑 Stopwords

- **Stopwords** are words that are used so frequently that they don't carry much information, especially for topic modeling
- **NLTK** has a built-in corpus of English stopwords that can be loaded and used

In [ ]:

```
from nltk.corpus import stopwords

stop_words = set(stopwords.words('english')) # you can also choose other languag
es
```

🕺 Here is an example of a tokenized sentence:

In [ ]:

```
tokens = ["i", "am", "going", "to", "go", "to", "the",
          "club", "and", "party", "all", "night", "long"]
```

❓ What stopwords could be removed ❓

In [ ]:

```
stopwords_removed = [w for w in tokens if w in stop_words]
stopwords_removed
```

Out[ ]:

```
['i', 'am', 'to', 'to', 'the', 'and', 'all']
```

❓ What are the meaningful words in this sentence ❓

In [ ]:

```
tokens_cleaned = [w for w in tokens if not w in stop_words]
tokens_cleaned
```

Out[ ]:

```
['going', 'go', 'club', 'party', 'night', 'long']
```

👉 What if you are *not* going to the party?

😱 "not" is also considered as a stopword

✅ Removing stopwords is useful for:

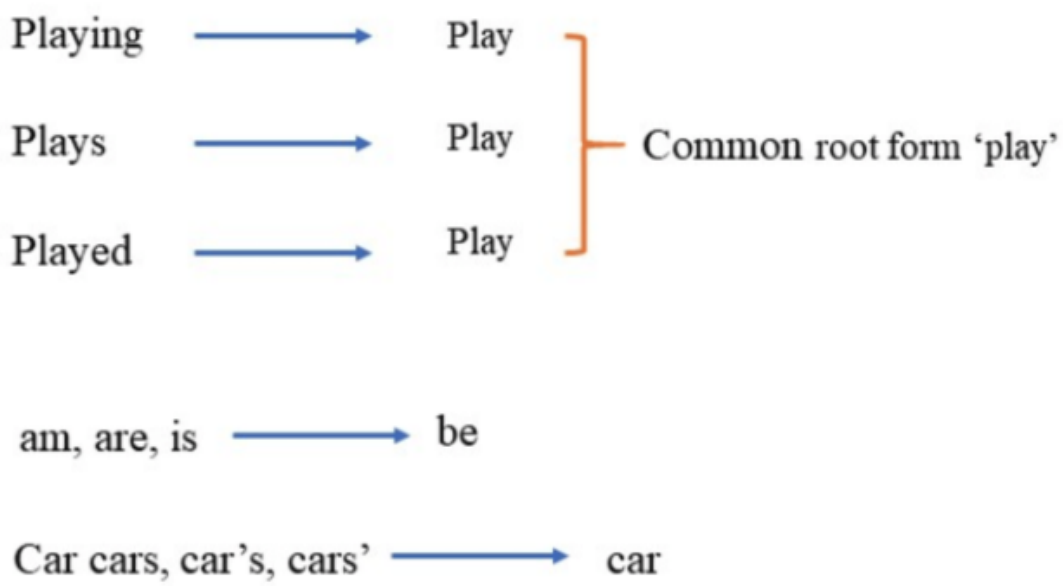
- topic modeling

❌ Dangerous for:

- sentiment analysis
- authorship attribution

💻 🌐 Lemmatizing

Lemmatizing is a technique used to find the **root** of words, in order to group them by their **meaning** rather than by their exact form



📚 **nltk.stem - WordNetLemmatizer** (https://www.nltk.org/_modules/nltk/stem/wordnet.html)

👇 Look at the following sentence:

In [ ]:

```
sentence
```

Out[ ]:

```
'He was RUNNING and EATING at the same time =[. He has a bad habit o
f swimming after playing 3 hours in the Sun =/'
```

🗒️ Let's apply the following steps:

1. Basic cleaning
2. Tokenizing
3. Removing stopwords
4. Lemmatizing

🧹 Step 1: Basic Cleaning

In [ ]:

```
sentence
```

Out[ ]:

```
'He was RUNNING and EATING at the same time =[. He has a bad habit o
f swimming after playing 3 hours in the Sun =/'
```

In [ ]:

```
cleaned_sentence = basic_cleaning(sentence)
cleaned_sentence
```

Out[ ]:

```
'he was running and eating at the same time  he has a bad habit of s
wimming after playing  hours in the sun'
```

🎄 Step 2 : Tokenize

Reminder: tokenizing means breaking a sentence down into a list of words, called "tokens")

In [ ]:

```
tokenized_sentence = word_tokenize(cleaned_sentence)
print(tokenized_sentence)
```

```
['he', 'was', 'running', 'and', 'eating', 'at', 'the', 'same', 'tim
e', 'he', 'has', 'a', 'bad', 'habit', 'of', 'swimming', 'after', 'pl
aying', 'hours', 'in', 'the', 'sun']
```

🛑 Step 3: Remove Stopwords

In [ ]:

```
tokenized_sentence_no_stopwords = [w for w in tokenized_sentence if not w in sto
p_words]
print(tokenized_sentence_no_stopwords)
```

```
['running', 'eating', 'time', 'bad', 'habit', 'swimming', 'playing',
'hours', 'sun']
```

🌐 Step 4: Lemmatizing

📚 **WordNetLemmatizer** (https://www.nltk.org/_modules/nltk/stem/wordnet.html)

In [ ]:

```
# Lemmatizing the verbs
verb_lemmatized = [
    WordNetLemmatizer().lemmatize(word, pos = "v") # v --> verbs
    for word in tokenized_sentence_no_stopwords
]

# 2 - Lemmatizing the nouns
noun_lemmatized = [
    WordNetLemmatizer().lemmatize(word, pos = "n") # n --> nouns
    for word in verb_lemmatized
]
```

In [ ]:

```
original_vs_lemmatized.style.hide(axis='index')
```

Out[ ]:

original word	lemmatized verbs	lemmatized nouns
running	run	run
eating	eat	eat
time	time	time
bad	bad	bad
habit	habit	habit
swimming	swim	swim
playing	play	play
hours	hours	hour
sun	sun	sun

✅ Lemmatizing is useful for:

- topic modeling
- sentiment analysis

🗑️ Preprocessing Text - Takeaways

- First of all, we can perform some **pre-cleaning operations** on the pieces of text of a corpus using Python built-in functions such as:
  - ✂️ `strip`
  - 👥 `replace`
  - 🪓 `split`
  - 🔡 `lowercase`
  - 🔢 removing numbers
  - ❗ removing punctuation and symbols

- Next, we can apply **preprocessing techniques** to prepare the pieces of text for NLP algorithms
  - 🎄 Tokenizing
  - 🛑 Removing stopwords
  - 🌐 Lemmatizing

🤔 Now that the text is preprocessed, how can it be analyzed by Machine Learning algorithms?

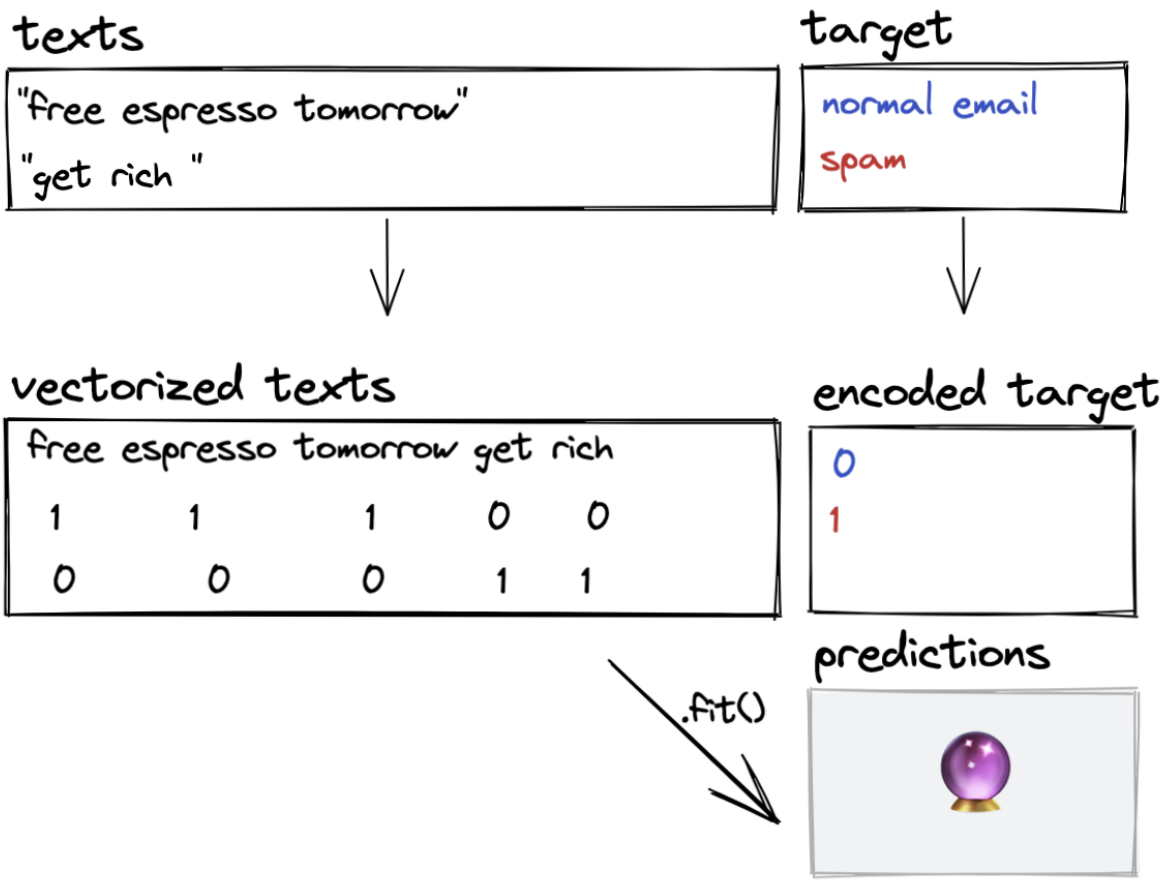
2. Vectorizing

🤖 Machine Learning algorithms cannot process *raw text*, as it needs to be *converted into numbers* first

🤦🏻 **`Vectorizing`** = the process of converting raw text into a numerical representation

There are multiple vectorizing techniques. Among them, we will present:

- Bag-of-Words
- Tf_idf
- N-grams



2.1. `Bag-of-Words` representation

💁🏻‍♀️ ***Bag-of-Words representation(BoW)*** is one of the most simple and effective ways to represent text for ML models.

- When using this representation, we are simply counting **how often** each **word** appears in each **document** of a **corpus**
- The count for each word becomes a feature:

	cat	dog	for	good	health	is	running	the	with	young	your
the young dog is running with the cat	1	1	0	0	0	1	1	2	1	1	0
running is good for your health	0	0	1	1	1	1	1	0	0	0	1
your cat is young	1	0	0	0	0	1	0	0	0	1	1
young young young young young cat cat cat	3	0	0	0	0	0	0	0	0	5	0

💻 `CountVectorizer`

💪 In Scikit-Learn, there is a tool called `CountVectorizer` to generate `bag-of-words` representations of a set of texts

👉 `CountVectorizer` converts a collection of text documents into a matrix of token counts

📚 **CountVectorizer** (https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html)

👇 Look at the following sentences:

In [ ]:

```
texts = [
    'the young dog is running with the cat',
    'running is good for your health',
    'your cat is young',
    'young young young young young cat cat cat'
]
```

💁🏻‍♀️ Let's apply the `CountVectorizer` to generate a `Bag-of-Words` representation of these four sentences

In [ ]:

```
from sklearn.feature_extraction.text import CountVectorizer

count_vectorizer = CountVectorizer()
X = count_vectorizer.fit_transform(texts)
X.toarray()
```

Out[ ]:

```
array([[1, 1, 0, 0, 0, 1, 1, 2, 1, 1, 0],
       [0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1],
       [1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1],
       [3, 0, 0, 0, 0, 0, 0, 0, 0, 5, 0]])
```

🤔 Can you guess which column represents which word?

🔥 As soon as the `CountVectorizer` is fitted to the text, you can retrieve all the words seen with `get_feature_names_out()` :

In [ ]:

```
count_vectorizer.get_feature_names_out()
```

Out[ ]:

```
array(['cat', 'dog', 'for', 'good', 'health', 'is', 'running', 'th
e',
       'with', 'young', 'your'], dtype=object)
```

In [ ]:

```
import pandas as pd

vectorized_texts = pd.DataFrame(
    X.toarray(),
    columns = count_vectorizer.get_feature_names_out(),
    index = texts
)

vectorized_texts
```

Out[ ]:

	cat	dog	for	good	health	is	running	the	with	young	your
the young dog is running with the cat	1	1	0	0	0	1	1	2	1	1	0
running is good for your health	0	0	1	1	1	1	1	0	0	0	1
your cat is young	1	0	0	0	0	1	0	0	0	1	1
young young young young young cat cat cat	3	0	0	0	0	0	0	0	0	5	0

Be aware that there are some limitations when it comes to the bag-of-words representation:

- ❌ A BoW does *NOT* take into account the **order of the words** $\rightarrow$ hence the name "Bag of Words"
- ❌ A BoW does *NOT* take into account a **document's length** $\rightarrow$ `Tf-idf` to the rescue
- ❌ A BoW does *NOT* capture **document context** $\rightarrow$ `N-gram` to the rescue

2.2. `Tf-idf` Representation

Term Frequency (tf) & CountVectorizer

> The more often a word appears in a document, the more likely it is that it will be important to this document

Example: if the word *elections* appears frequently in a document, it is obvious that this document deals with *politics*.

> The frequency of a word x in a document d is called **`term frequency`** , and is denoted by $t_{x,d}$

❓ In our last example, could we compute $t_{young,3}$ ❓

In [ ]:

```
vectorized_texts
```

Out[ ]:

	cat	dog	for	good	health	is	running	the	with	young	your
the young dog is running with the cat	1	1	0	0	0	1	1	2	1	1	0
running is good for your health	0	0	1	1	1	1	1	0	0	0	1
your cat is young	1	0	0	0	0	1	0	0	0	1	1
young young young young young cat cat cat	3	0	0	0	0	0	0	0	0	5	0

$\rightarrow t_{young,3} = 5$

Document Frequency (df)

> If a word appears in many documents of a corpus, however, it shouldn't be that important to understand the corpus.

Example: on eurosport.com/football (https://www.eurosport.com/football/), the word "*football*" appears in every article, hence why the word *football* on this website is an unimportant word!

> The number of documents d in a corpus containing the word x is called **`document frequency (df)`** , and is denoted by df_x

> The **relative document frequency** of a word x can be computed as $\frac{df_x}{N}$
>
> $\rightarrow df_x$ = number of documents d containing the word x
> $\rightarrow N$ = total number of documents in a corpus

❓ In our last example, could we compute df_{cat}, df_{young} and df_{the} ❓

In [ ]:

```
vectorized_texts
```

Out[ ]:

	cat	dog	for	good	health	is	running	the	with	young	your
the young dog is running with the cat	1	1	0	0	0	1	1	2	1	1	0
running is good for your health	0	0	1	1	1	1	1	0	0	0	1
your cat is young	1	0	0	0	0	1	0	0	0	1	1
young young young young young cat cat cat	3	0	0	0	0	0	0	0	0	5	0

> If a word x appears in too many documents of a corpus - i.e. if the document frequency df_x is too high - the word x won't help us with topic modeling and should be considered irrelevant.

Example: on eurosport.com/football/ (https://www.eurosport.com/football/), the word "football" won't help us distinguish two articles, one dealing mainly with strategy and another one talking about referee best practices!

> 👩🏻 A word x in a corpus of texts will be considered important when its (relative) document frequency is low ⇔ its **`inverse document frequency`** $\frac{N}{df_x}$ is high.

Tf-idf Formula

> 💡 The intuition of the `term frequency - inverse document frequency` approach is to give a high weight to any term which appears frequently in a single document, but not in too many documents of the corpus.

👩🏻 ***Weight of a word x in a document d:***

$$w_{x,d} = \underbrace{tf_{x,d}}_{"tf"} \times \underbrace{\left[log\left(\frac{N+1}{df_x+1}\right)+1\right]}_{"idf"}$$

where:

- $tf_{x,d}$ = number of occurences of a word x in the document d
- df_x = number of documents d containing the word x
- N = total number of documents in a corpus

2.3. 💻 TfidfVectorizer

🧨 `raw documents` $\rightarrow$ `matrix of tf-idf features`

📚 **sklearn.feature_extraction.text.TfidfVectorizer** (https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html)

In [ ]:

```
texts
```

Out[ ]:

```
['the young dog is running with the cat',
 'running is good for your health',
 'your cat is young',
 'young young young young young cat cat cat']
```

In [ ]:

```
from sklearn.feature_extraction.text import TfidfVectorizer
```

In [ ]:

```
# Instantiating the TfidfVectorizer
tf_idf_vectorizer = TfidfVectorizer()

# Training it on the texts
weighted_words = pd.DataFrame(tf_idf_vectorizer.fit_transform(texts).toarray(),
                 columns = tf_idf_vectorizer.get_feature_names_out())

weighted_words
```

Out[ ]:

	cat	dog	for	good	health	is	running	the	with	
0	0.227904	0.357056	0.000000	0.000000	0.000000	0.227904	0.281507	0.714112	0.357056	(
1	0.000000	0.000000	0.463709	0.463709	0.463709	0.295980	0.365594	0.000000	0.000000	(
2	0.470063	0.000000	0.000000	0.000000	0.000000	0.470063	0.000000	0.000000	0.000000	(
3	0.514496	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	(

Controlling the vocabulary size:

In every language, there are many words used in everyday vocabulary:

- 🇬🇧 English: ~20,000 words
- 🇫🇷 French: ~20,000 words
- 🇩🇪 German: ~20,000 words

In a document, we can't afford to vectorize every word!

We can, however, control the number of words to be vectorized (*curse of dimensionality (https://www.analyticsvidhya.com/blog/2021/04/the-curse-of-dimensionality-in-machine-learning/)!*):

👉 Scikit-Learn allows us to customize the *CountVectorizer* and *TfidVecdtorizer* with key parameters to control vocabulary size.

💻 **Key parameters of `TfidfVectorizer` (https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html) (and `CountVectorizer` (https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html))**

- `max_df` / `min_df`
- `max_features`

💻 **`max_df`** (resp. **`min_df`**)

> When building the vocabulary, CountVectorizer and TfidfVectorizer will remove terms which have a document frequency strictly higher (resp. lower) than the given threshold. *max_df* and *min_df* help us building **corpus-specific stopwords**.

Example: when classifying pieces of text into "basketball" or "football", the word "ball" would appear too often and would be useless for this classification, it would be better to filter it out using `max_df`

How to use these parameters in practice?

- `max_df` (`min_df`) can be either a `float` between 0.0 and 1.0 or an `integer`
  - `max_df (min_df) = 0.5` ⇔ "ignore terms that appear in more (less) than 50% of the documents"
  - `max_df (min_df) = 20` ⇔ "ignore terms that appear in more (less) than 20 documents"
- By default, `max_df = 1.0` ⇔ no "frequent" word will be removed
- By default, `min_df = 0.0` ⇔ no "infrequent" word will be removed

In [ ]:

```
# Number of occurences of each word
document_frequency
```

Out[ ]:

	cat	dog	for	good	health	is	running	the	with	young	your
document_frequency	3	1	1	1	1	3	2	1	1	3	2

In [ ]:

```
# Instantiate the CountVectorizer with max_df = 2
count_vectorizer = CountVectorizer(max_df = 2) # removing "cat", "is", "young"

# Train it
X = count_vectorizer.fit_transform(texts)
X = pd.DataFrame(
    X.toarray(),
    columns = count_vectorizer.get_feature_names_out(),
    index = texts
)

X
```

Out[ ]:

	dog	for	good	health	running	the	with	your
the young dog is running with the cat	1	0	0	0	1	2	1	0
running is good for your health	0	1	1	1	1	0	0	1
your cat is young	0	0	0	0	0	0	0	1
young young young young young cat cat cat	0	0	0	0	0	0	0	0

`max_features`

> By specifying `max_features = k` (k being an integer), the CountVectorizer (or the TfidfVectorizer) will build a vocabulary that only considers the top k tokens ordered by term frequency across the corpus.

How to use "max_features" in practice?

In [ ]:

```
# CountVectorizer with the 3 most frequent words
count_vectorizer = CountVectorizer(max_features = 3)

X = count_vectorizer.fit_transform(texts)
X = pd.DataFrame(
    X.toarray(),
     columns = count_vectorizer.get_feature_names_out(),
     index = texts
)

X
```

Out[ ]:

	cat	is	young
the young dog is running with the cat	1	1	1
running is good for your health	0	1	0
your cat is young	1	1	1
young young young young young cat cat cat	3	0	5

✅ Advantages of the `Tf-idf` representation:

- Using relative frequency rather than count is robust to document length
- Takes into account the context of the whole corpus

❌ Disadvantages of the `Tf-idf` representation:

- Like the `BoW`, `Tf-idf` does *NOT* capture the **within-document context** → `N-gram` helps here
- Like the `BoW`, the word order is completely disregarded

2.4. `N-grams`

Example: the two following sentences have the exact same representation:

In [ ]:

```
actors_movie = [
    "I like the movie but NOT the actors",
    "I like the actors but NOT the movie"
]
```

In [ ]:

```
# Vectorize the sentences
count_vectorizer = CountVectorizer()
actors_movie_vectorized = count_vectorizer.fit_transform(actors_movie)

# Show the representations in a nice DataFrame
actors_movie_vectorized = pd.DataFrame(
    actors_movie_vectorized.toarray(),
    columns = count_vectorizer.get_feature_names_out(),
    index = actors_movie
)

# Show the vectorized movies
actors_movie_vectorized
```

Out[ ]:

	actors	but	like	movie	not	the
I like the movie but NOT the actors	1	1	1	1	1	2
I like the actors but NOT the movie	1	1	1	1	1	2

 When using a `bag-of-words` representation, an efficient way to **capture context** is to consider:

- the count of single tokens (unigrams)
- the count of pairs (bigrams), triplets (trigrams), and more generally sequences of n words, also known as **n-grams**

Examples:

- "*mathematics*" is a unigram (n = 1)
- "*machine learning*" is a bigram (n = 2)
- "*natural language processing*" is a trigram (n = 3)
- "*deep convolutional neural networks*" is a 4-gram (n = 4)

 **`ngram_range`**

In both *CountVectorizer* and *TfidfVectorizer*, you can specify the length of your sequences with the parameter `ngram_range = (min_n, max_n)`.

Examples:

- `ngram_range = (1, 1)` 👉 (by default) will only capture the unigrams (single words)
- `ngram_range = (1, 2)` 👉 will capture the unigrams, and the bigrams
- `ngram_range = (1, 3)` 👉 will capture the unigrams, the bigrams, and the trigrams
- `ngram_range = (2, 3)` 👉 will capture the bigrams, and the trigrams but not the unigrams

😢 With a **unigram vectorization**, we couldn't distinguish two sentences with the same words.

In [ ]:

```
actors_movie_vectorized
```

Out[ ]:

	actors	but	like	movie	not	the
I like the movie but NOT the actors	1	1	1	1	1	2
I like the actors but NOT the movie	1	1	1	1	1	2

👩🏻‍🎨 What about a **bigram vectorization**?

In [ ]:

```
# Vectorize the sentences
count_vectorizer_n_gram = CountVectorizer(ngram_range = (2,2)) # BI-GRAMS
actors_movie_vectorized_n_gram = count_vectorizer_n_gram.fit_transform(actors_mo
vie)

# Show the representations in a nice DataFrame
actors_movie_vectorized_n_gram = pd.DataFrame(
    actors_movie_vectorized_n_gram.toarray(),
    columns = count_vectorizer_n_gram.get_feature_names_out(),
    index = actors_movie
)

# Show the vectorized movies with bigrams
actors_movie_vectorized_n_gram
```

Out[ ]:

	actors but	but not	like the	movie but	not the	the actors	the movie
I like the movie but NOT the actors	0	1	1	1	1	1	1
I like the actors but NOT the movie	1	1	1	0	1	1	1

😄 The two sentences are now distinguishable

🗃 Vectorizing - Takeaways

- 🤖 ❤️ 🔢

- There are two methods for vectorizing:
  - `CountVectorizer` (counting)
  - `TfidfVectorizer` (weighing: take the *document length* into consideration)

- The most important parameters of these vectorizers are:
  - `min_df` *(infrequent words)*
  - `max_df` *(frequent words)*
  - `max_features` *(curse of dimensionality)*
  - `ngram_range = (min_n, max_n)` *(capturing the context of the words)*

🚀 Let's discover two NLP algorithms:

- **(Multinomial) Naive Bayes** → classification algorithm
- **Latent Dirichlet Allocation (LDA)** → unsupervised topic labeling

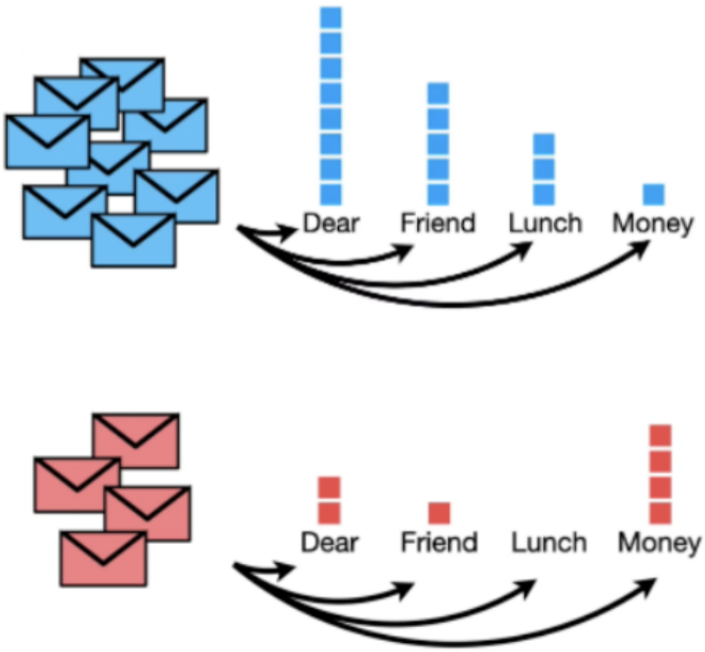
3. (Multinomial) Naive Bayes Algorithm

The **Multinomial Naive Bayes** algorithm is a classification algorithm based on **Bayes' Theorem** in probability theory

3.1. ✉️ The E-mail Classification Problem

🎯 We want to classify e-mails based on their content:

- ✅ Normal (N)
- ☠️ Spam (S)



🤔 What is the probability that an e-mail containing some specific words be spam?

👩🏻‍🔬 Mathematical Approach

Mathematically speaking, the probability that an e-mail containing specific words is spam can be denoted by:

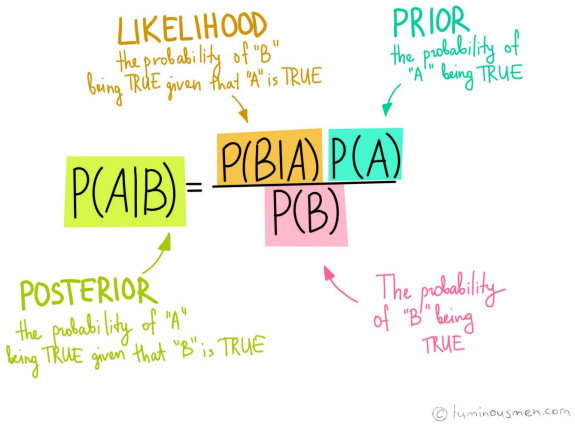
$$\mathbb{P}(S|x_1, x_2, \ldots, x_k)$$

where:

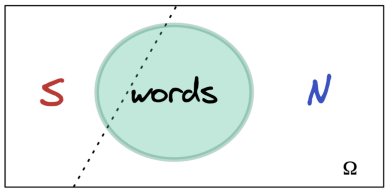
- S = "this e-mail is spam"
- x_k = "the word x_k appears in this e-mail"

$\mathbb{P}(S|x_1, x_2, \ldots, x_k)$

$$= \frac{\mathbb{P}(x_1,x_2,...,x_k|S)\times\mathbb{P}(S)}{\mathbb{P}(x_1,x_2,...,x_k)} \text{ (Bayes' Theorem)}$$



$$= \frac{\mathbb{P}(x_1,x_2,...,x_k|S)\times\mathbb{P}(S)}{\mathbb{P}(x_1,x_2,...,x_k \cap S)+\mathbb{P}(x_1,x_2,...,x_k \cap N)} \text{ (Law of Total Probabilities)}$$



$$= \frac{\mathbb{P}(x_1,x_2,...,x_k|S)\times\mathbb{P}(S)}{\mathbb{P}(x_1,x_2,...,x_k|S)\times\mathbb{P}(S)+\mathbb{P}(x_1,x_2,...,x_k|N)\times\mathbb{P}(N)} \text{ (Conditional Probability)}$$

👉 Let's focus on a specific term:

$$\mathbb{P}(x_1, x_2, \ldots, x_k|S)$$

> The **Naive Bayes algorithm** makes the strong assumption that the words in an e-mail are **conditionally independent**

By applying the **independence property**:

$$\mathbb{P}(x_1, x_2, \ldots, x_k|S) = \mathbb{P}(x_1|S) \times \mathbb{P}(x_2|S) \times \ldots \times \mathbb{P}(x_k|S)$$
$$= \prod_{i=1}^{k} \mathbb{P}(x_i|S)$$

In the **Naive Bayes algorithm**, the **probability that an e-mail is spam if it contains certain words** is the following:

Spam Formula

$$\mathbb{P}(S|x_1, x_2, \dots, x_k) = \frac{\mathbb{P}(S) \times \prod_{i=1}^{k} \mathbb{P}(x_i|S)}{\mathbb{P}(S) \times \prod_{i=1}^{k} \mathbb{P}(x_i|S) + \mathbb{P}(N) \times \prod_{i=1}^{k} \mathbb{P}(x_i|N)}$$

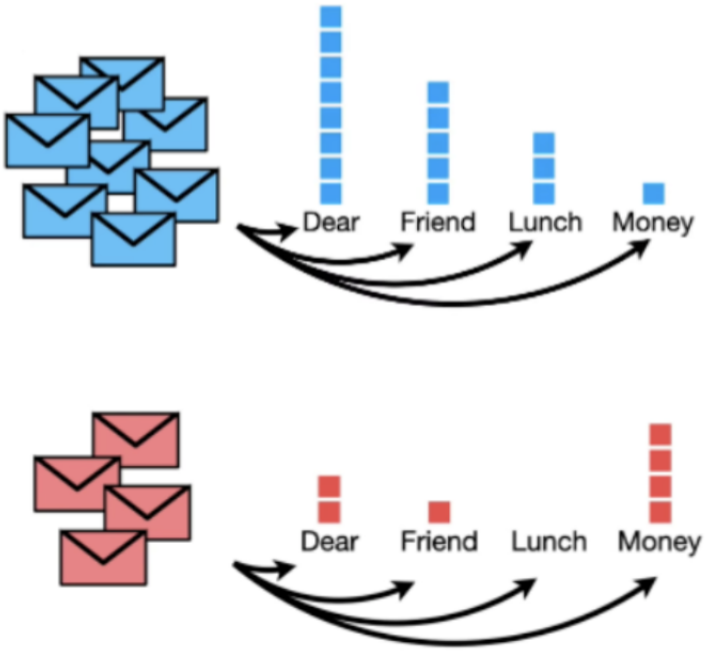
Computational Approach

Imagine that you have an e-mail inbox with:

- 8 normal e-mails
- 4 spam e-mails

? What is the probability that an e-mail with *Dear Friend* is spam ?

$\rightarrow \mathbb{P}(S|\text{ " Dear "," Friend "})$



Prior probabilities

$\mathbb{P}(N) = \frac{8}{8+4} = 0.67$

$\mathbb{P}(S) = \frac{4}{8+4} = 0.33$

Likelihood

$\mathbb{P}(\text{" Dear "}|N) = \frac{8}{17} = 0.47$

$\mathbb{P}(\text{" Friend "}|N) = \frac{5}{17} = 0.29$

$\mathbb{P}(\text{" Dear "}|S) = \frac{2}{7} = 0.29$

$\mathbb{P}(\text{" Friend "}|S) = \frac{1}{7} = 0.14$

Probability of being spam if the e-mail contains Dear Friend

$\mathbb{P}(S|\text{ " Dear "," Friend "})$

$= \frac{\mathbb{P}(S) \times \mathbb{P}(\text{"Dear"}|S) \times \mathbb{P}(\text{"Friend"}|S)}{\mathbb{P}(S) \times \mathbb{P}(\text{"Dear"}|S) \times \mathbb{P}(\text{"Friend"}|S) + \mathbb{P}(N) \times \mathbb{P}(\text{"Dear"}|N) \times \mathbb{P}(\text{"Friend"}|N)}$

$= \frac{\frac{4}{8+4} \times \frac{2}{7} \times \frac{1}{7}}{\frac{4}{8+4} \times \frac{2}{7} \times \frac{1}{7} + \frac{8}{8+4} \times \frac{8}{17} \times \frac{5}{17}}$

$$= \frac{0.0136}{0.0136+0.0923}$$

$$= 0.129$$

$$= 12.9\%$$

🏂 Smoothing

Imagine that you want to compute $\mathbb{P}(S|\text{ " }Dear\text{ "},\text{ " }Lunch\text{ "})$.

😰 You will be in trouble

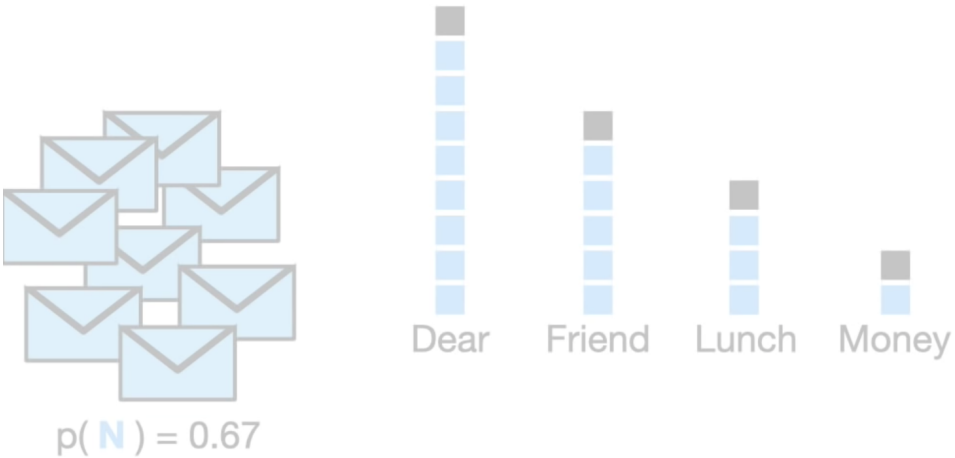
- $\mathbb{P}(\text{" }Dear\text{ " }|S) = \frac{2}{7} = 0.29$
- $\mathbb{P}(\text{" }Lunch\text{ " }|S) = \frac{0}{7} = 0.00$

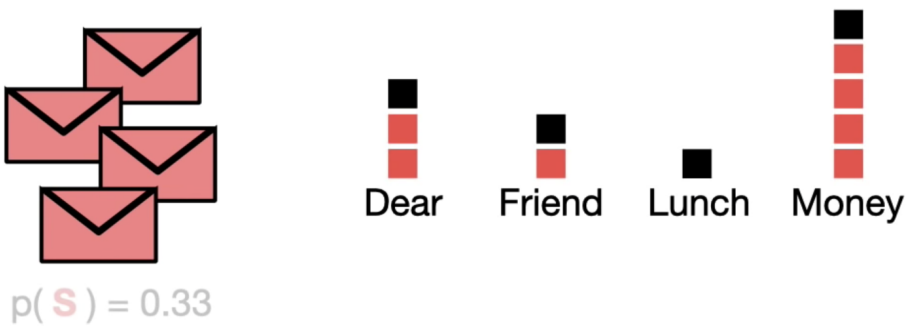
and when multiplying these probabilities, you will have a null probability!

🤔 How do we overcome this problem with words which don't appear in spam e-mails?

💡 We can add +1 (or $\alpha > 0$) to *term frequencies*.

🤵 This is called **smoothing** and α is the **smoothing parameter**





3.2. Pros and Cons of the NB Algorithm

✅ Pros:

- Easy to implement
- Not an iterative learning process — fast!
- Works particularly well on text data because it can handle a large vocabulary
- Not a parametric model (no β to learn, no loss function to minimize)

❌ Cons:

- Assumes that the words appearing in a document don't depend on any previous words

3.3. 💻 Implementation of the Naive Bayes Algorithm

📚 **`MultinomialNB`** (https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.MultinomialNB.html)

✉️ Let's have a look at a dataset with thousands of e-mails classified either as spam or as a normal e-mail.

In [ ]:

```
import pandas as pd

data = pd.read_csv("data/emails.csv")
data.head()
```

Out[ ]:

	text	spam
0	Subject: naturally irresistible your corporate...	1
1	Subject: the stock trading gunslinger fanny i...	1
2	Subject: unbelievable new homes made easy im ...	1
3	Subject: 4 color printing special request add...	1
4	Subject: do not have money , get software cds ...	1

In [ ]:

```
data.shape
```

Out[ ]:

```
(5728, 2)
```

In [ ]:

```
round(data["spam"].value_counts(normalize = True), 2)
```

Out[ ]:

```
0    0.76
1    0.24
Name: spam, dtype: float64
```

In [ ]:

```
import numpy as np

from sklearn.model_selection import cross_validate
from sklearn.pipeline import make_pipeline
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import recall_score

# Feature/Target
X = data["text"]
y = data["spam"]

# Pipeline vectorizer + Naive Bayes
pipeline_naive_bayes = make_pipeline(
    TfidfVectorizer(),
    MultinomialNB()
)

# Cross-validation
cv_results = cross_validate(pipeline_naive_bayes, X, y, cv = 5, scoring = ["recall"])
average_recall = cv_results["test_recall"].mean()
np.round(average_recall,2)
```

Out[ ]:

```
0.45
```

💪 On average, the Naive Bayes algorithm is able to capture almost half of the spam e-mails, which is quite a good performance for a "naive" model!

3.4. 💻 Tuning the Vectorizer and the Naive Bayes Algorithm Simultaneously

🚨 Different vectorizing hyperparameters will affect the performance of the model. As such, it is important to simultaneously tune the hyperparameters of both the vectorizer and the Naive Bayes model.

💡 Remember that all the transformers and estimators of Scikit-Learn can be pipelined!

In [ ]:

```
from sklearn.model_selection import GridSearchCV

# Define the grid of parameters
parameters = {
    'tfidfvectorizer__ngram_range': ((1,1), (2,2)),
    'multinomialnb__alpha': (0.1,1)
}

# Perform Grid Search
grid_search = GridSearchCV(
    pipeline_naive_bayes,
    parameters,
    scoring = "recall",
    cv = 5,
    n_jobs=-1,
    verbose=1
)

grid_search.fit(data.text,data.spam)

# Best score
print(f"Best Score = {grid_search.best_score_}")

# Best params
print(f"Best params = {grid_search.best_params_}")
```

```
Fitting 5 folds for each of 4 candidates, totalling 20 fits
Best Score = 0.9524932488436137
Best params = {'multinomialnb__alpha': 0.1, 'tfidfvectorizer__ngram_
range': (1, 1)}
```

4. Topic Modeling and Latent Dirichlet Allocation 🔥

☢️ Disclaimer

📚 This section is based on a research paper called **Latent Dirichlet Allocation** (https://www.jmlr.org/papers/volume3/blei03a/blei03a.pdf) *[dɪʁiˈkleː]* published in 2003 in the Journal of Machine Learning (https://www.jmlr.org/) by

- David M. Bei (Columbia)
- Andrew Y. Ng (Stanford)
- Michael I. Jordan (Berkeley)

📚 If you read Wikipedia/Latent_Dirichlet_Model (https://en.wikipedia.org/wiki/Latent_Dirichlet_allocation#Model), you will see that there are plenty of parameters and complex probability distributions to deal with

🐣 Consider this section an introduction to topic modeling, we will give you:

- an intuition about how LDA works
- how to use it on some texts

4.1. What is LDA?

🤖 **`Latent Dirichlet Allocation`** is an unsupervised algorithm for finding topics in documents

- "Latent" = hidden (topics)
- "Dirichlet" = type of probability distribution
  - Document → collection of topics
  - Topic → collection of tokens/words

📚 **`sklearn.decomposition.LatentDirichletAllocation`** (https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.LatentDirichletAllocation.html)

👇 Consider the following documents:

In [ ]:

```
documents
```

Out[ ]:

	documents
0	I like mangos and oranges
1	Frogs and turtles live in ponds
2	Kittens and puppies are fluffy
3	I had a spinach and kiwi smoothie
4	My kitten loves strawberries

🍔 **Inputs**:

- **Document-term matrix**: documents to be converted using a vectorizer
- **Number of topics**: number of topics to be discovered within the documents
  - Each "topic" consists of a set of *unordered* words → **bag-of-words** format
- **Number of iterations** → LDA is an unsupervised iterative process

🎯 **Output**:

- **Topics** across different documents/pieces of text
  - These topics can be interpreted as "non-linear Principal Components" of the documents in the corpus

4.2. 💻 Implementation of the LDA

👇 Remember our original documents?

In [ ]:

```
documents
```

Out[ ]:

	documents
0	I like mangos and oranges
1	Frogs and turtles live in ponds
2	Kittens and puppies are fluffy
3	I had a spinach and kiwi smoothie
4	My kitten loves strawberries

4.2.1. 💻 Cleaning the dataset

In [ ]:

```
def cleaning(sentence):

    # Basic cleaning
    sentence = sentence.strip() ## remove whitespaces
    sentence = sentence.lower() ## lowercase
    sentence = ''.join(char for char in sentence if not char.isdigit()) ## remov
e numbers

    # Advanced cleaning
    for punctuation in string.punctuation:
        sentence = sentence.replace(punctuation, '') ## remove punctuation

    tokenized_sentence = word_tokenize(sentence) ## tokenize
    stop_words = set(stopwords.words('english')) ## define stopwords

    tokenized_sentence_cleaned = [ ## remove stopwords
        w for w in tokenized_sentence if not w in stop_words
    ]

    lemmatized = [
        WordNetLemmatizer().lemmatize(word, pos = "v")
        for word in tokenized_sentence_cleaned
    ]

    cleaned_sentence = ' '.join(word for word in lemmatized)

    return cleaned_sentence
```

In [ ]:

```
cleaned_documents = documents["documents"].apply(cleaning)
cleaned_documents.head()
```

Out[ ]:

```
0          like mangos oranges
1       frog turtle live ponds
2        kitten puppies fluffy
3        spinach kiwi smoothie
4     kitten love strawberries
Name: documents, dtype: object
```

4.2.2. 💻 Vectorizing

In [ ]:

```
vectorizer = TfidfVectorizer()

vectorized_documents = vectorizer.fit_transform(cleaned_documents)
vectorized_documents = pd.DataFrame(
    vectorized_documents.toarray(),
    columns = vectorizer.get_feature_names_out()
)

vectorized_documents
```

Out[ ]:

	fluffy	frog	kitten	kiwi	like	live	love	mangos	oranges	ponds	pupp
0	0.000000	0.0	0.000000	0.00000	0.57735	0.0	0.000000	0.57735	0.57735	0.0	0.0000
1	0.000000	0.5	0.000000	0.00000	0.00000	0.5	0.000000	0.00000	0.00000	0.5	0.0000
2	0.614189	0.0	0.495524	0.00000	0.00000	0.0	0.000000	0.00000	0.00000	0.0	0.6141
3	0.000000	0.0	0.000000	0.57735	0.00000	0.0	0.000000	0.00000	0.00000	0.0	0.0000
4	0.000000	0.0	0.495524	0.00000	0.00000	0.0	0.614189	0.00000	0.00000	0.0	0.0000

4.3.3 💻 Finding the topics

In [ ]:

```
from sklearn.decomposition import LatentDirichletAllocation

# Instantiate the LDA
n_components = 2
lda_model = LatentDirichletAllocation(n_components=n_components, max_iter = 100)

# Fit the LDA on the vectorized documents
lda_model.fit(vectorized_documents)
```

Out[ ]:

```
LatentDirichletAllocation
LatentDirichletAllocation(max_iter=100, n_components=2)
```

Document Mixture (of Topics)

In [ ]:

```
document_topic_mixture = lda_model.transform(vectorized_documents)
```

In [ ]:

```
document_topic_mixture
```

Out[ ]:

	topic_0	topic_1	Original Text
sentence 0	0.80	0.20	I like mangos and oranges
sentence 1	0.18	0.82	Frogs and turtles live in ponds
sentence 2	0.80	0.20	Kittens and puppies are fluffy
sentence 3	0.20	0.80	I had a spinach and kiwi smoothie
sentence 4	0.80	0.20	My kitten loves strawberries

🤔 How could our topic modeling be improved?

- by increasing the number of sentences
- by increasing the number of iterations

Topic Mixture (of Words)

In [ ]:

```
topic_word_mixture = pd.DataFrame(
    lda_model.components_,
    columns = vectorizer.get_feature_names_out()
)
```

In [ ]:

```
topic_word_mixture
```

Out[ ]:

	fluffy	frog	kitten	kiwi	like	live	love	mangos	orange
topic_0	1.098881	0.50989	1.474904	0.511486	1.061562	0.50989	1.098879	1.061562	1.06156
topic_1	0.515308	0.99011	0.516143	1.065864	0.515789	0.99011	0.515310	0.515789	0.51578

🕵️ What are the five most relevant words for each topic?

In [ ]:

```
def print_topics(lda_model, vectorizer, top_words):
    # 1. TOPIC MIXTURE OF WORDS FOR EACH TOPIC
    topic_mixture = pd.DataFrame(
        lda_model.components_,
        columns = vectorizer.get_feature_names_out()
    )

    # 2. FINDING THE TOP WORDS FOR EACH TOPIC
    ## Number of topics
    n_components = topic_mixture.shape[0]

    ## Top words for each topic
    for topic in range(n_components):
        print("-"*10)
        print(f"For topic {topic}, here are the the top {top_words} words with w
eights:")

        topic_df = topic_mixture.iloc[topic]\
            .sort_values(ascending = False).head(top_words)

        print(round(topic_df,3))
```

In [ ]:

```
print_topics(lda_model, vectorizer, 5)
```

```
----------
For topic 0, here are the the top 5 words with weights:
kitten          1.475
fluffy          1.099
puppies         1.099
love            1.099
strawberries    1.099
Name: 0, dtype: float64
----------
For topic 1, here are the the top 5 words with weights:
kiwi        1.066
smoothie    1.066
spinach     1.066
frog        0.990
live        0.990
Name: 1, dtype: float64
```

4.3. Bonus: LDA Under the Hood

🎯 The goal of an LDA is to find topics across documents.

👩🏻‍🏫 The LDA converts the **vectorized documents** (= `document_term_matrix`) into two matrices:

- `document_topic_mixture`
- `topic_word_mixture`

0️⃣ **Choose the number of topics you want to detect in your corpus of documents**

Example: `n_components = 2` $\rightarrow$ Topic 0 and Topic 1

1️⃣ **Randomly assign each word in each document to one of topics**

Example: The word "mangos" in Document 0 is randomly assigned to Topic 1

2️⃣ **Go through every word and its topic assignment in each document**

(1) **Document Mixture** `p(topic t | document d)`

$\rightarrow$ how often a topic t occurs in a document d

(2) **Topic Mixture** `p(word w | topic t)`

$\rightarrow$ how often the word w occurs in the topic d

(3) **Update** `p(word w with topic t) = p( t | d) * p( w | t)`

🔁 Go through multiple iterations of step 2️⃣

🚀 Eventually, the topics will start making sense

Document Mixture (of Topics)

- Computing `p(topic t | document d)` for every topic and every document is called **document mixture**
- The ideal document mixture for our example would be the following (*falsely assuming without verification that topic* 0 = *food and topic* 1 = *animals*):

In [ ]:

```
document_topic_matrix_ideal
```

Out[ ]:

	documents	topic_food	topic_animals
0	I like mangos and oranges	1.0	0.0
1	Frogs and turtles live in ponds	0.0	1.0
2	Kittens and puppies are fluffy	0.0	1.0
3	I had a spinach and kiwi smoothie	1.0	0.0
4	My kitten loves strawberries	0.5	0.5

Topic Mixture (of Words)

- Computing `p(word w | topic t)` for every word and every topic is called **topic mixture**
- The ideal topic mixture for our example would be the following:

In [ ]:

```
topic_word_matrix_ideal
```

Out[ ]:

	topic	like	mangos	oranges	frog	turtle	live	ponds	kitten	puppies	fluffy	spinac
0	topic_food	1	1	1	0	0	0	0	0	0	0	
1	topic_animal	1	0	0	1	1	1	1	1	1	1	

Your turn! 🚀