# **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

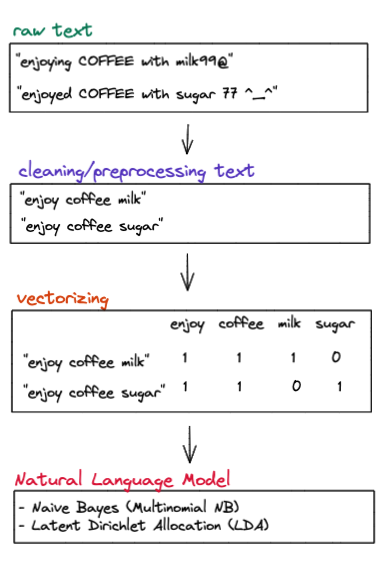
### **What is NLP?**

**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)

## **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

texts = [

' Bonjour, comment ca va ? ',

' Heyyyyy, how are you doing ? ',

' Hallo, wie gehts ? '

]

texts

[' Bonjour, comment ca va ? ',

' Heyyyyy, how are you doing ? ',

' Hallo, wie gehts ? ']

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

['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

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

text

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

text.strip('bdac')

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

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

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

text

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

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

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

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

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

text.split("/")

['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.

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

text

'i LOVE football sO mUch. FOOTBALL is my passion. Who else loves fOOtBaLL ?'

text.lower()

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

#### **💻 🔢 Numbers**

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

* text clustering
* collecting keyphrases

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

text

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

cleaned\_text = ''.join(char **for** char **in** text **if** **not** char.isdigit())

cleaned\_text

'i do not recommend this restaurant, we waited for so long, like minutes, 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!*

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

text

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

**import** **string** *# "string" module is already installed with Python*

string.punctuation

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

**for** punctuation **in** string.punctuation:

text = text.replace(punctuation, '')

text

'I love bubble tea OMG so tasty channel XOXO '

#### **💻 💪 Combo: strip + lowercase + numbers + punctuation/symbols**

sentences = [

" I LOVE Pizza 999 @^\_^",

" Le Wagon is amazing, take care - 666"

]

**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

cleaned = [basic\_cleaning(sentence) **for** sentence **in** sentences]

cleaned

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

#### **💻 🔍 Removing Tags with RegEx**

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

**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:

**import** **re**

txt = '''

This is a random text, authored by darkvador@gmail.com

and batman@outlook.com, WOW!

'''

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

['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:

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

text

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

**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', 'must', '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

**from** **nltk.corpus** **import** stopwords

stop\_words = set(stopwords.words('english')) *# you can also choose other languages*

🕺🏻 Here is an example of a tokenized sentence:

tokens = ["i", "am", "going", "to", "go", "to", "the",

"club", "and", "party", "all", "night", "long"]

❓ What stopwords could be removed ❓

stopwords\_removed = [w **for** w **in** tokens **if** w **in** stop\_words]

stopwords\_removed

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

❓ What are the meaningful words in this sentence ❓

tokens\_cleaned = [w **for** w **in** tokens **if** **not** w **in** stop\_words]

tokens\_cleaned

['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:

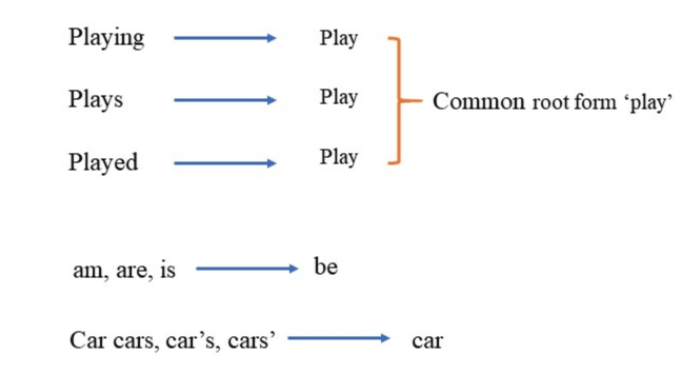
* topic modeling

❌ 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:

sentence

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

🗓 Let's apply the following steps:

1. Basic cleaning
2. Tokenizing
3. Removing stopwords (if not doing sentiment analysis!)
4. Lemmatizing

🧹 Step 1: Basic Cleaning

sentence

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

cleaned\_sentence = basic\_cleaning(sentence)

cleaned\_sentence

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

🎄 Step 2 : Tokenize

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

tokenized\_sentence = word\_tokenize(cleaned\_sentence)

print(tokenized\_sentence)

['he', 'was', 'running', 'and', 'eating', 'at', 'the', 'same', 'time', 'he', 'has', 'a', 'bad', 'habit', 'of', 'swimming', 'after', 'playing', 'hours', 'in', 'the', 'sun']

🛑 Step 3: Remove Stopwords

tokenized\_sentence\_no\_stopwords = [w **for** w **in** tokenized\_sentence **if** **not** w **in** stop\_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) (Only supports English)

*# 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

]

original\_vs\_lemmatized.style.hide(axis='index')

| **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?

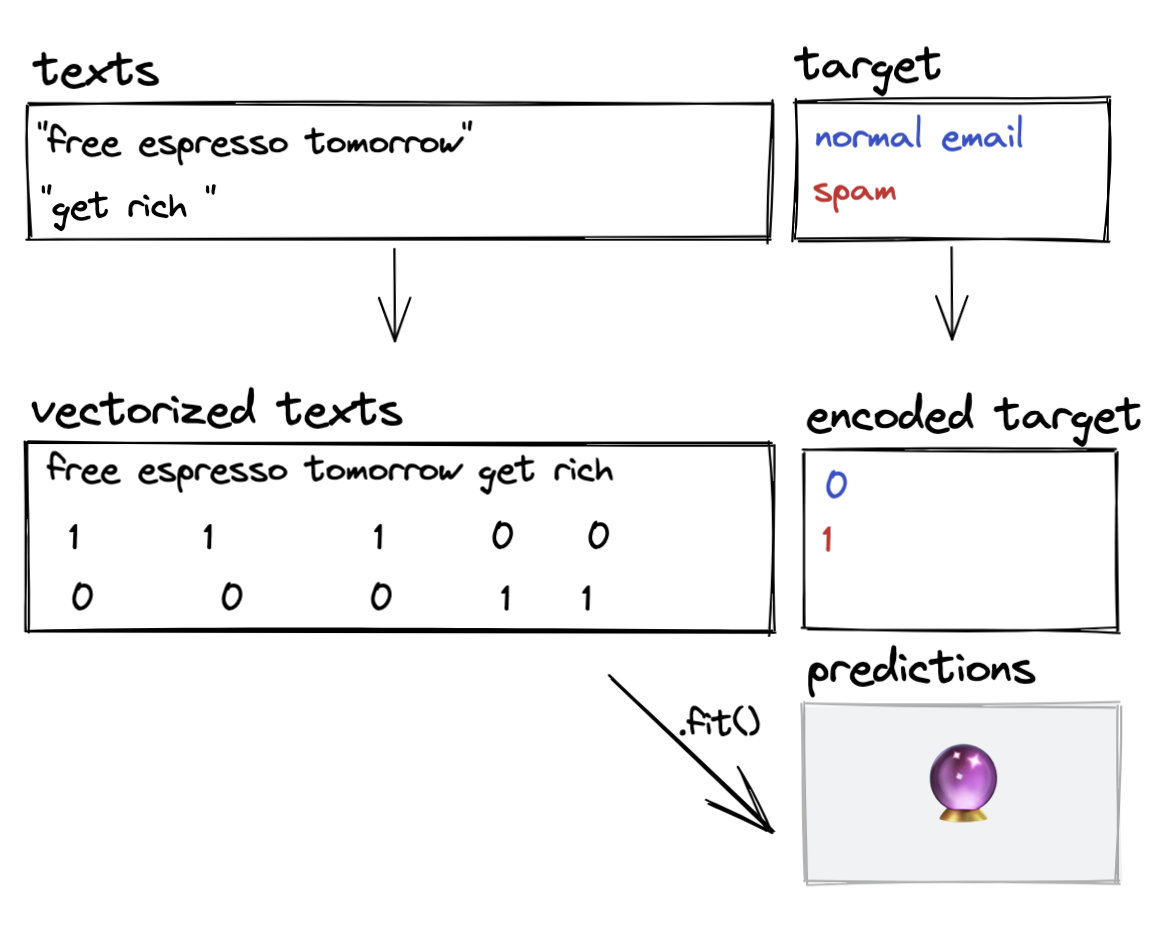
## **2. Vectorizing**

🤖 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-wordsrepresentations 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:

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

**from** **sklearn.feature\_extraction.text** **import** CountVectorizer

count\_vectorizer = CountVectorizer()

X = count\_vectorizer.fit\_transform(texts)

X.toarray()

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():

count\_vectorizer.get\_feature\_names\_out()

array(['cat', 'dog', 'for', 'good', 'health', 'is', 'running', 'the',

'with', 'young', 'your'], dtype=object)

**import** **pandas** **as** **pd**

vectorized\_texts = pd.DataFrame(

X.toarray(),

columns = count\_vectorizer.get\_feature\_names\_out(),

index = texts

)

vectorized\_texts

|  | **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**
* →
* hence the name "Bag of Words"
* ❌ A BoW does *NOT* take into account a **document's length**
* →
* Tf-idf to the rescue
* ❌ A BoW does *NOT* capture **document context**
* →
* N-gram to the rescue

### **2.2. Tf-idf Representation**

#### **Term Frequency (tf) & CountVectorizer**

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

Example: if the word *elections* appears relatively 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

t

e

r

m

f

r

e

q

u

e

n

c

y

, and is denoted by:

TF

x

,

d

=

Number of times term

x

appears in document

d

Total number of terms in the document

❓ In our last example, could we compute

t

f

y

o

u

n

g

,

d

o

c

u

m

e

n

t

4

❓

vectorized\_texts

|  | **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 |

→

t

f

y

o

u

n

g

,

d

o

c

u

m

e

n

t

4

=

5

counts of "young" /

8

total words

=

0.625

#### **Document Frequency (df)**

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

*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

d

f

x

❓ In our last example, could we compute

d

f

c

a

t

,

d

f

y

o

u

n

g

and

d

f

t

h

e

❓

vectorized\_texts

|  | **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

d

f

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!

What if we considered the **relative document frequency** of a word

x

can be computed as

d

f

x

N

?

→

d

f

x

= number of documents

d

containing the word

x

→

N

= total number of documents in a corpus

For the word "football" on Eurosport, we would expect this formula to be **close to 1** since the number of docs containing the word "football" will probably only be *slightly* less than the total number of docs (out of 100 maybe only 5 don't have the word "football", so we get 95/100).

👩🏻‍🏫 A word

x

in a corpus of texts will be considered **important** when its (relative) document frequency is **low**

⇔

its **inverse document frequency**

N

d

f

x

is high.

Again, if the word "football" appears in **all** the articles it is not very useful for helping us identify between two articles, but if **only a few** documents contain words like "concussion" or "wellbeing", (e.g. they appear in 2/100 articles) it will be much more useful in determining the topic of that article (they are probably specifically about player wellfare).

#### **Tf-idf Formula**

💡 Thus 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

=

t

f

x

,

d

"

t

f

"

×

[

l

o

g

(

N

+

1

d

f

x

+

1

)

+

1

]

"

i

d

f

"

where:

* t
* f
* x
* ,
* d
* = number of occurences of a word
* x
* in the document
* d
* / number of words in the document
* d
* f
* x
* = number of documents
* d
* containing the word
* x
* N
* = total number of documents in a corpus

### **2.3. 💻 TfidfVectorizer**

🧨 raw documents

→

matrix of tf-idf features

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

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']

**from** **sklearn.feature\_extraction.text** **import** TfidfVectorizer

*# 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

|  | **cat** | **dog** | **for** | **good** | **health** | **is** | **running** | **the** | **with** | **young** | **your** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 0.227904 | 0.357056 | 0.000000 | 0.000000 | 0.000000 | 0.227904 | 0.281507 | 0.714112 | 0.357056 | 0.227904 | 0.000000 |
| **1** | 0.000000 | 0.000000 | 0.463709 | 0.463709 | 0.463709 | 0.295980 | 0.365594 | 0.000000 | 0.000000 | 0.000000 | 0.365594 |
| **2** | 0.470063 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.470063 | 0.000000 | 0.000000 | 0.000000 | 0.470063 | 0.580622 |
| **3** | 0.514496 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.857493 | 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

*# Number of occurences of each word*

document\_frequency

|  | **cat** | **dog** | **for** | **good** | **health** | **is** | **running** | **the** | **with** | **young** | **your** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **document\_frequency** | 3 | 1 | 1 | 1 | 1 | 3 | 2 | 1 | 1 | 3 | 2 |

*# 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

|  | **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?**

*# 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

|  | **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:

actors\_movie = [

"I like the movie but NOT the actors",

"I like the actors but NOT the movie"

]

*# 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

|  | **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.

actors\_movie\_vectorized

|  | **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**?

*# 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\_movie)

*# 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

|  | **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

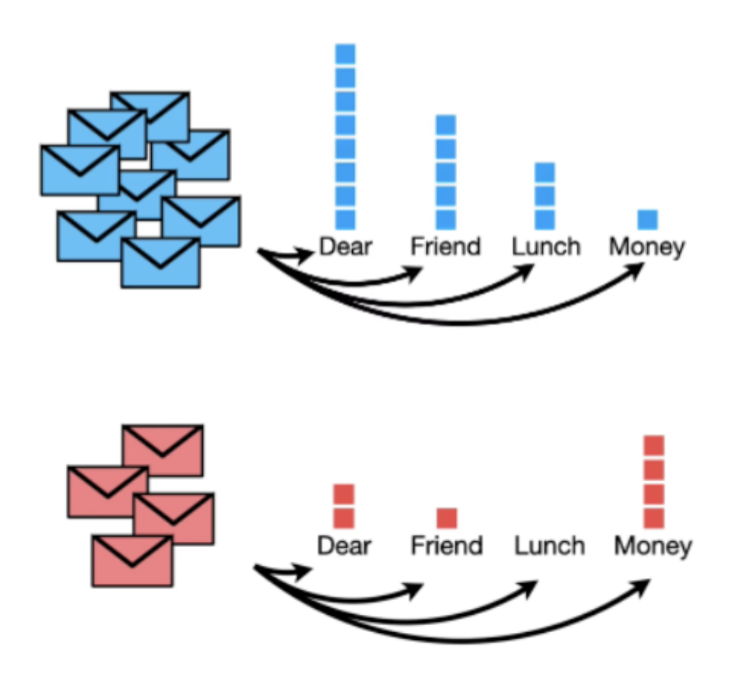
## **3. (Multinomial) Naive Bayes Algorithm**

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:

P

(

S

|

x

1

,

x

2

,

.

.

.

,

x

k

)

where:

* S
* = "this e-mail is spam"
* x
* k
* = "the word
* x
* k
* appears in this e-mail"

P

(

S

|

x

1

,

x

2

,

.

.

.

,

x

k

)

=

P

(

x

1

,

x

2

,

.

.

.

,

x

k

|

S

)

×

P

(

S

)

P

(

x

1

,

x

2

,

.

.

.

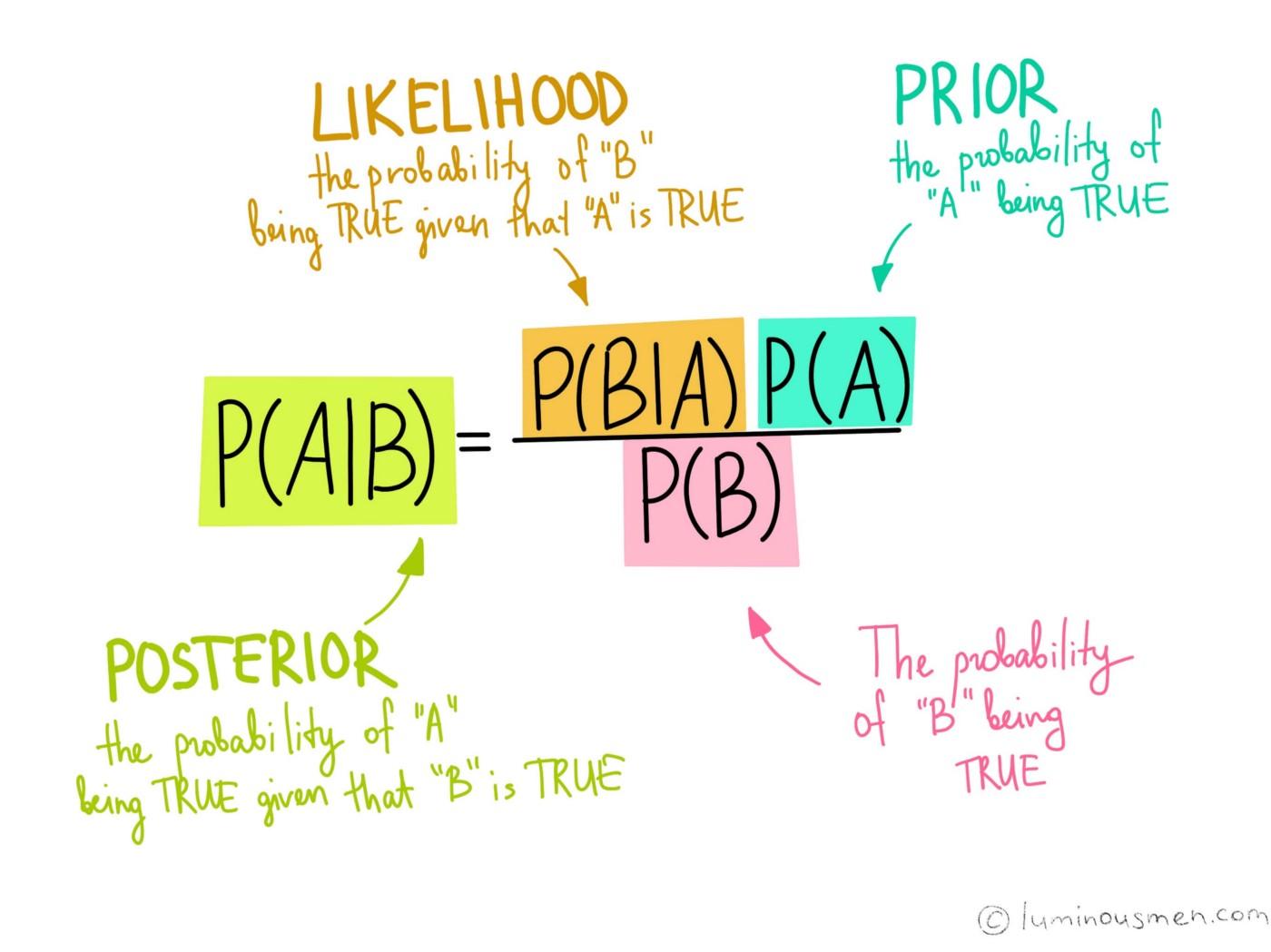
,

x

k

)

(*Bayes' Theorem*)



=

P

(

x

1

,

x

2

,

.

.

.

,

x

k

|

S

)

×

P

(

S

)

P

(

x

1

,

x

2

,

.

.

.

,

x

k

∩

S

)

+

P

(

x

1

,

x

2

,

.

.

.

,

x

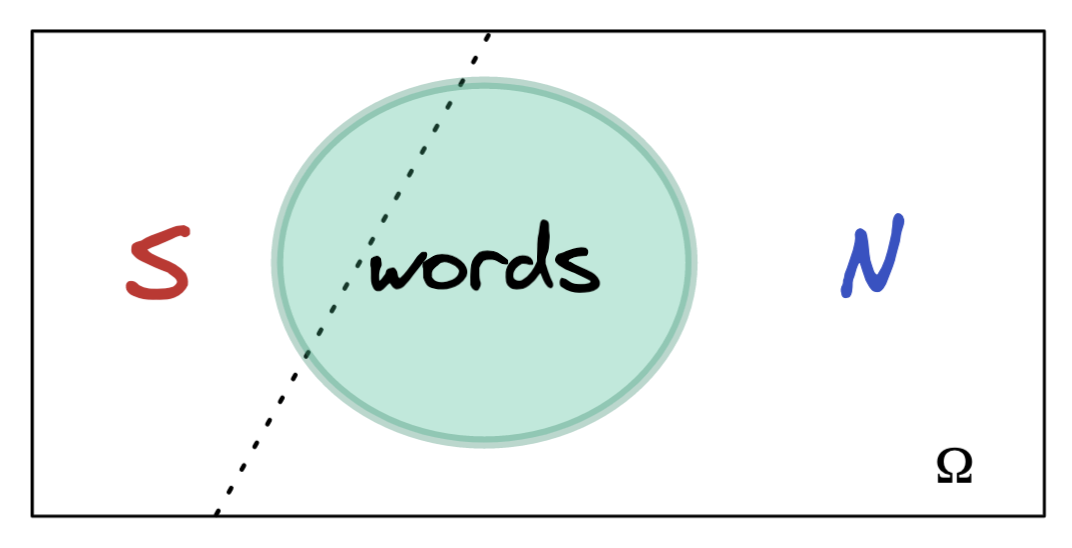
k

∩

N

)

*(Law of Total Probabilities)*

**

=

P

(

x

1

,

x

2

,

.

.

.

,

x

k

|

S

)

×

P

(

S

)

P

(

x

1

,

x

2

,

.

.

.

,

x

k

|

S

)

×

P

(

S

)

+

P

(

x

1

,

x

2

,

.

.

.

,

x

k

|

N

)

×

P

(

N

)

*(Conditional Probability)*

👉 Let's focus on a specific term:

P

(

x

1

,

x

2

,

.

.

.

,

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**:

P

(

x

1

,

x

2

,

.

.

.

,

x

k

|

S

)

=

P

(

x

1

|

S

)

×

P

(

x

2

|

S

)

×

.

.

.

×

P

(

x

k

|

S

)

=

k

∏

i

=

1

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**

P

(

S

|

x

1

,

x

2

,

.

.

.

,

x

k

)

=

P

(

S

)

×

∏

k

i

=

1

P

(

x

i

|

S

)

P

(

S

)

×

∏

k

i

=

1

P

(

x

i

|

S

)

+

P

(

N

)

×

∏

k

i

=

1

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 ❓

→

P

(

S

|

"

D

e

a

r

"

,

"

F

r

i

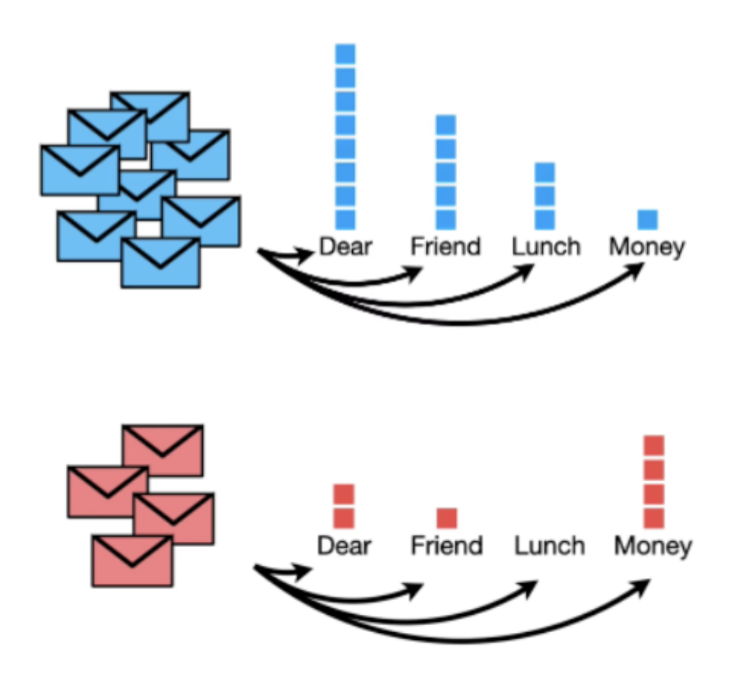
e

n

d

"

)



**Prior probabilities**

P

(

N

)

=

8

8

+

4

=

0.67

P

(

S

)

=

4

8

+

4

=

0.33

**Likelihood**

P

(

"

D

e

a

r

"

|

N

)

=

8

17

=

0.47

P

(

"

F

r

i

e

n

d

"

|

N

)

=

5

17

=

0.29

P

(

"

D

e

a

r

"

|

S

)

=

2

7

=

0.29

P

(

"

F

r

i

e

n

d

"

|

S

)

=

1

7

=

0.14

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

P

(

S

|

"

D

e

a

r

"

,

"

F

r

i

e

n

d

"

)

=

P

(

S

)

×

P

(

"

D

e

a

r

"

|

S

)

×

P

(

"

F

r

i

e

n

d

"

|

S

)

P

(

S

)

×

P

(

"

D

e

a

r

"

|

S

)

×

P

(

"

F

r

i

e

n

d

"

|

S

)

+

P

(

N

)

×

P

(

"

D

e

a

r

"

|

N

)

×

P

(

"

F

r

i

e

n

d

"

|

N

)

=

4

8

+

4

×

2

7

×

1

7

4

8

+

4

×

2

7

×

1

7

+

8

8

+

4

×

8

17

×

5

17

=

0.0136

0.0136

+

0.0923

=

0.129

=

12.9

%

#### **🏂 Smoothing**

Imagine that you want to compute

P

(

S

|

"

D

e

a

r

"

,

"

L

u

n

c

h

"

)

.

😰 You will be in trouble

* P
* (
* "
* D
* e
* a
* r
* "
* |
* S
* )
* =
* 2
* 7
* =
* 0.29
* P
* (
* "
* L
* u
* n
* c
* h
* "
* |
* S
* )
* =
* 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

α

>

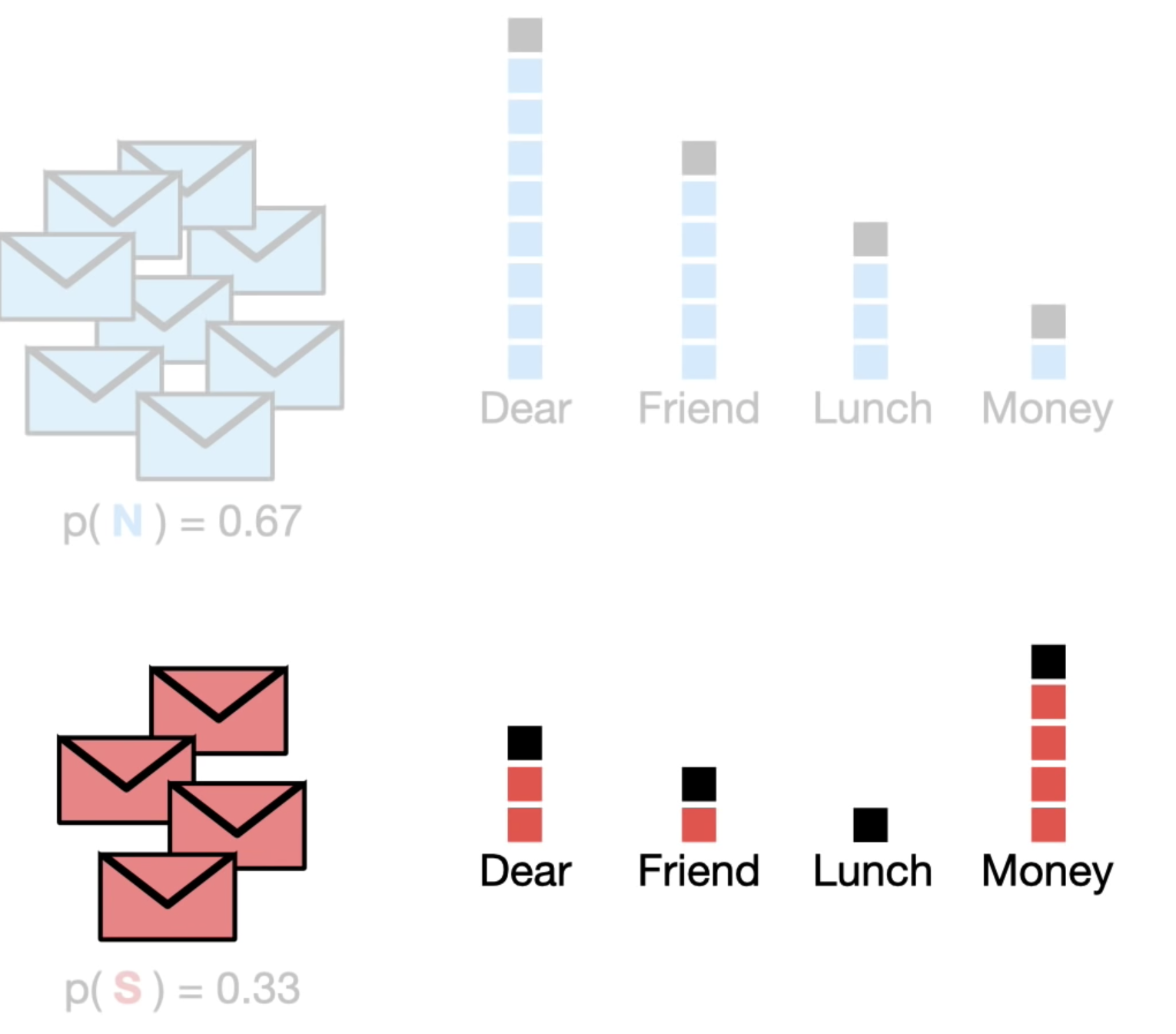
0

) 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.

**import** **pandas** **as** **pd**

data = pd.read\_csv("data/emails.csv")

data.head()

|  | **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 |

data.shape

(5728, 2)

round(data["spam"].value\_counts(normalize = **True**), 2)

0 0.76

1 0.24

Name: spam, dtype: float64

**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)

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!

**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) *[diʀiˈkleː]* 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:

documents

|  | **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?

documents

|  | **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**

**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()) *## remove 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

cleaned\_documents = documents["documents"].apply(cleaning)

cleaned\_documents.head()

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**

vectorizer = TfidfVectorizer()

vectorized\_documents = vectorizer.fit\_transform(cleaned\_documents)

vectorized\_documents = pd.DataFrame(

vectorized\_documents.toarray(),

columns = vectorizer.get\_feature\_names\_out()

)

vectorized\_documents

|  | **fluffy** | **frog** | **kitten** | **kiwi** | **like** | **live** | **love** | **mangos** | **oranges** | **ponds** | **puppies** | **smoothie** | **spinach** | **strawberries** | **turtle** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 0.000000 | 0.0 | 0.000000 | 0.00000 | 0.57735 | 0.0 | 0.000000 | 0.57735 | 0.57735 | 0.0 | 0.000000 | 0.00000 | 0.00000 | 0.000000 | 0.0 |
| **1** | 0.000000 | 0.5 | 0.000000 | 0.00000 | 0.00000 | 0.5 | 0.000000 | 0.00000 | 0.00000 | 0.5 | 0.000000 | 0.00000 | 0.00000 | 0.000000 | 0.5 |
| **2** | 0.614189 | 0.0 | 0.495524 | 0.00000 | 0.00000 | 0.0 | 0.000000 | 0.00000 | 0.00000 | 0.0 | 0.614189 | 0.00000 | 0.00000 | 0.000000 | 0.0 |
| **3** | 0.000000 | 0.0 | 0.000000 | 0.57735 | 0.00000 | 0.0 | 0.000000 | 0.00000 | 0.00000 | 0.0 | 0.000000 | 0.57735 | 0.57735 | 0.000000 | 0.0 |
| **4** | 0.000000 | 0.0 | 0.495524 | 0.00000 | 0.00000 | 0.0 | 0.614189 | 0.00000 | 0.00000 | 0.0 | 0.000000 | 0.00000 | 0.00000 | 0.614189 | 0.0 |

#### **4.3.3 💻 Finding the topics**

**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)

LatentDirichletAllocation

LatentDirichletAllocation(max\_iter=100, n\_components=2)

##### ***Document Mixture (of Topics)***

document\_topic\_mixture = lda\_model.transform(vectorized\_documents)

document\_topic\_mixture

|  | **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)***

topic\_word\_mixture = pd.DataFrame(

lda\_model.components\_,

columns = vectorizer.get\_feature\_names\_out()

)

topic\_word\_mixture

|  | **fluffy** | **frog** | **kitten** | **kiwi** | **like** | **live** | **love** | **mangos** | **oranges** | **ponds** | **puppies** | **smoothie** | **spinach** | **strawberries** | **turtle** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **topic\_0** | 1.098881 | 0.50989 | 1.474904 | 0.511486 | 1.061562 | 0.50989 | 1.098879 | 1.061562 | 1.061562 | 0.50989 | 1.098881 | 0.511486 | 0.511486 | 1.098879 | 0.50989 |
| **topic\_1** | 0.515308 | 0.99011 | 0.516143 | 1.065864 | 0.515789 | 0.99011 | 0.515310 | 0.515789 | 0.515789 | 0.99011 | 0.515308 | 1.065864 | 1.065864 | 0.515310 | 0.99011 |

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

**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 weights:")

topic\_df = topic\_mixture.iloc[topic]\

.sort\_values(ascending = **False**).head(top\_words)

print(round(topic\_df,3))

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

→

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)

→

how often a topic

t

occurs in a document

d

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

→

how often the word

w

occurs in the topic

t

(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*):

document\_topic\_matrix\_ideal

|  | **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:

topic\_word\_matrix\_ideal

|  | **topic** | **like** | **mangos** | **oranges** | **frog** | **turtle** | **live** | **ponds** | **kitten** | **puppies** | **fluffy** | **spinach** | **kiwi** | **smoothie** | **love** | **strawberry** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | topic\_food | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| **1** | topic\_animal | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |

# **Your turn! 🚀**