

# Python Pour le Machine Learning

## Introduction Rapide au langage Python :

### ⇒ Les Variables

Pour déclarer une variable on fait :

```
x = 1
```

Pour l'afficher :

```
print(x)
```

Pour modifier la valeur de la variable on peut faire :

```
x = 1
```

```
print(x)
```

```
x = 2
```

```
print(x)
```

### ⇒ Les Opérations

On peut mettre à jour la variable en faisant des opérations dessus :

```
x = 2
```

```
x = x + 1 # On peut écrire cette ligne plus simplement « x += 1 »
```

```
print(x) # Ici x = 3 à la fin
```

On peut faire des calculs classiques :

```
x = 1
```

```
y = 2
```

```
print(x + y) # addition
```

```
print(x - y) # soustraction
```

```
print(x * y) # multiplication
```

```
print(x / y) # division
```

```
print(x % y) # modulo (retourne l'entier restant de la division)
```

```
print(x ** 2) # faire une puissance 2 ici par exemple avec **
```

## ⇒ Les Comparaisons

```
print(x < y) # x inférieur à y
print(x > y) # x supérieur à y
print(x ≤ y) # x inférieur ou égale à y
print(x ≥ y) # x supérieur ou égale à y
print(x == y) # x égale à y
print(x != y) # x différent de y
```

## ⇒ Les Types

```
a = True # bool c'est un booléen
a = 2 # int c'est un entier
a = 2.5 # float c'est un nombre à virgule
a = 'salut' # c'est un str un string donc une chaîne de caractère
```

## ⇒ Les Opérateurs Logiques

```
print(False & True) # AND
print(False | True) # OR
print(False ^ True) # XOR
```

Créer 2 variables différentes sur une même ligne :

```
x, y = 1, 2.5
```

## ⇒ Les Fonctions

```
f = lambda x: x**2
ça c'est équivalent à dire  $f(x) = x^2$ 
Donc on peut faire :
print(f(3)) # et on va obtenir 9
```

Les fonctions les plus courantes sont faites comme ça (avec des paramètres dans les fonctions ici) :

```
def energie_potentielle(masse, hauteur, g):
    E = masse * hauteur * g
    print(E, 'Joules')
```

```
energie_potentielle(5,8.3,9.81)
```

# Nous retourne la réponse de la formule écrite dans la fonction automatiquement avec le string 'Joules' derrière

On peut faire des constantes au sein des fonctions :

```
def energie_potentielle(masse, hauteur, g=9.81):
```

```
    E = masse * hauteur * g
```

```
    print(E, 'Joules')
```

energie\_potentielle(5,8.3) # on a plus besoin de mettre g ici vu qu'on l'a défini comme valeur par défaut mais on peut la changer ci besoin en le remettant en tant que paramètre lors de l'appel de la fonction

## ⇒ Les Conditions

```
x = 1
```

```
if x > 0 :
```

```
    print(x, 'positif')
```

```
elif x == 0 :
```

```
    print(x, 'égale à 0')
```

```
else :
```

```
    print(x, 'est négatif')
```

On peut aussi utiliser des opérateurs logiques :

```
x = 1
```

```
y = -5
```

```
if (x > 0) & (y > 0) :
```

```
    print('x et y sont positifs')
```

```
else :
```

```
    print('x ou y est négatif ou nul')
```

On peut utiliser les fonctions avec :

```
def signe(x) :  
    if (x > 0) :  
        print(x, 'est positif')  
    else :  
        print(x, 'est négatif ou nul')
```

signe(5) # Nous retourne « 5 est positif »

### ⇒ Les Boucles

```
for i in range(10) :  
    print('bonjour') # affichera 10 fois bonjour
```

```
for i in range(10) :  
    print(i) # affichera les nombres de 0 à 9
```

*on met « i » par défaut souvent mais on peut l'appeler  
« élément », « z », « itérations » etc*

```
for i in range(5, 10) :  
    print(i) # affichera les nombres de 5 à 9
```

```
for i in range(5, 10, 2) :  
    print(i) # affichera les nombres 5 7 et 9 car le pas est de 2
```

Boucle While:

```
x = 0  
while x < 10 :  
    print(x)  
    x += 1  
# Ici on affichera autant de fois x tant que x < 10
```

## ⇒ Les Listes & Tuples

```
list1 = [1, 4, 5, 6, 7]
list2 = ['a', 'b', 'c', 'd',]
list_fusion = [list1, list2]
```

```
tuple1 = (1, 2, 6, 8)
```

le tuple à la différence d'une liste on ne peut pas le modifier.

Les listes, tuples et string forment un ensemble qu'on nomme SEQUENCE, une séquence c'est une structure de données qui suit un Ordre.

Chaque élément a un index de 0 jusqu'à la fin et grâce à cet index on peut accéder directement à des éléments de notre liste, tuple ou dictionnaire.

```
villes = ['Berlin', 'Paris', 'Lyon', 'Barcelone']
print(villes[0]) # affichera Berlin
print(villes[-1]) # affichera Barcelone (permet d'afficher le dernier
élément directement)
```

C'est une technique qui se nomme **INDEXING**

```
print(villes[0:3]) # affichera Berlin Paris Lyon
print(villes[:3]) → Souvent on ne met pas le 0 on met direct « : »
print(villes[2:]) # affichera Lyon Barcelone
print(villes[2:3]) # affichera Lyon
print(villes[::2]) # affichera Berlin et Lyon car le pas est de 2, ça
saute de 2 en 2 ici
print(villes[::-1]) # affichera la liste dans le sens inverse, d'abord
Barcelone puis Lyon, Paris et Berlin
```

C'est une technique qui se nomme le **SLICING**

**INDEXING & SLICING** → techniques utilisent pour changer les valeurs des variables et les trouver rapidement.

Ajout d'élément dans une liste :

```
villes = ['Berlin', 'Paris', 'Lyon', 'Barcelone']
```

```
villes.append('Dublin') # rajoutera Dublin à la fin de la liste
```

```
villes.insert(2, 'Madrid') # rajoutera Madrid à l'index 2 et va donc  
décaler les autres index
```

```
villes2 = ['Rome', 'Amsterdam']
```

```
villes.extend(villes2) # rajoutera à la fin de la liste villes les  
éléments de la liste2
```

```
len(villes) # affichera 8, permet de voir le nombres d'éléments  
contenus dans une liste
```

```
villes.sort() # Trie la liste par ordre croissant
```

```
villes.sort(reverse=True) # Trie la liste par ordre décroissant
```

```
villes.count('Rome') # affichera 1, car Rome est présent qu'une  
fois, permet de compter le nombre d'élément précisément
```

On peut utiliser ça avec les conditions :

```
if 'Paris' in villes :
```

```
    print('Bonjour Paris')
```

```
else :
```

```
    print('ah bah non en fait')
```

On peut utiliser avec les boucles :

```
for i in villes :
```

```
    print(i) # affichera tous les noms des villes
```

```
for index, valeur in enumerate(villes) :
```

```
    print(index, valeur) # affichera tous les index et leurs valeurs
```

```
villes = ['Berlin', 'Paris', 'Lyon', 'Barcelone']
villes2 = ['Rome', 'Amsterdam']
for a, b in zip(villes, villes2) :
    print(a, b) # affichera les éléments des 2 listes
```

Ici la boucle for s'arrêtera à la liste la plus courte, donc si les 2 listes ne sont pas de mêmes longueur, ça sera la plus petite qui sera affichée en entière.

### ⇒ Les Dictionnaires (clefs:valeurs)

```
dict = {'chien': 'dog', 'chat': 'cat', 'souris': 'mouse'}
```

Dans un dictionnaire chaque clef est unique, on ne peut pas en avoir 2 qui sont les mêmes.

Mais les valeurs peuvent être répétées.

On peut Nested les dictionnaires :

```
dict1 = {'chien': 'dog', 'chat': 'cat', 'souris': 'mouse'}
```

```
dict2 = {'poires' : 4, 'pommes' : 5}
```

```
dict3 = {'dictionnaire1':dict1, 'dictionnaire2':dict2}
```

On a un dict3 qui contient 2 dictionnaires.

Parfois en DeepLearning on stock les paramètres de nos réseaux de neurones dans les dictionnaires :

```
import numpy as np
```

```
params = {
    'w1': np.random.randn(10,100),
    'b1': np.random.randn(10,1),
    'w2': np.random.randn(10,10),
    'b2' : np.random.randn(10,1)
}
```

Opérations sur dictionnaires :

```
dict1 = {'chien': 'dog', 'chat': 'cat', 'souris': 'mouse'}  
dict2 = {'poires' : 4, 'pommes' : 5}  
dict1.values() # On obtient les valeurs présentent dans notre  
dictionnaire  
dict1.keys() # On obtient les clés  
dict1.len() # on obtient la taille  
dict1['abricots'] = 42 # Rajoute un nouvel élément dans le  
dictionnaire
```

Dans un dictionnaire il n'y a pas d'ordre donc on ne peut pas les trier par index ou faire un append() pour rajouter un élément.

Boucler sur un dictionnaire :

```
for k, v in dict1.items() :  
    print(k,v) # affiche les clés et les valeurs
```

## ⇒ La Compréhension de Liste

```
list = []  
for i in range(10) :  
    list.append(i**2)  
c'est la même chose que faire :  
list = [i**2 for i in range(10)]
```

ça permet de gagner plus de temps dans le processus.

On peut le faire avec une Nested list :

```
list3 = [[i for i in range(3)] for j in range(3)]  
# ça va nous donner [[0,1,2], [0,1,2], [0,1,2]]
```

```
list3 = [[i+j for i in range(3)] for j in range(3)]  
# ça va nous donner [[0,1,2], [1,2,3], [2,3,4]]
```



## ⇒ La Compréhension de Dictionnaire

```
dictionnaire = {'0' : 'Pierre', '1' : 'Jean', '2' : 'Julie'}
```

```
prenoms = ['Pierre', 'Jean', 'Julie']
```

```
dict = {k:v for k, v in enumerate(prenoms)}
```

```
ages = [24, 62, 10]
```

```
dict2 = {prenom:age for prenom, age in zip(prenoms, ages)}
```

Condition & dictionnaire :

```
dict2 = {prenom:age for prenom, age in zip(prenoms, ages) if age  
> 18}
```

## ⇒ La Compréhension pour les Tuples

```
tuple1 = tuple((i**2 for i in range(10)))
```

tuple1 # affichera 0,1,4,9,16,25 jusqu'à 81, le carré de chaque entiers jusqu'à 9

## ⇒ Les Built-In Fonctions

Ce sont des fonctions pré-intégrés dans Python on retrouve tout un tas de fonctions : **abs()** / **round()** / **max()** / **min()** / **len()** / **sum()** / **all()** / **any()** etc

*(voir la doc de Python si besoin pour en trouver des spécifiques).*

<https://docs.python.org/3/library/functions.html>

Il existe la fonction « input » qui permet de récupérer ce que l'utilisateur saisi :

```
x = int(input('saisissez un nombre'))
```

```
print(x) # affichera le nombre entré par l'utilisateur
```

Fonction `.format()` :

```
x = 25
```

```
ville = Paris
```

```
message = 'la température est de {} degC a {}'.format(x, ville)
```

fonctionne aussi en faisant :

```
message = f'la température est de {x} degC a {ville}'
```

Fonction `.open()` :

Cette fonction permet de lire et d'écrire des fichiers

```
f = open('fichier.txt', 'w')
```

```
f.write('bonjour')
```

```
f.close()
```

### ⇒ Les Importations de bibliothèques & packages

On peut utiliser « import » pour importer nos propres fichiers .py (il suffit de mettre leurs noms)

```
ex : import mon_projet
```

Mais on peut aussi importer des bibliothèques de Python comme Numpy

```
ex : import numpy as np
```

On peut aussi importer juste une fonction d'un fichier

```
ex : import projet.ma_fonction
```

Utilisation du `np.array` :

```
tableau = np.array([1,2,3]) # crée un tableau de dimension 1 avec 3 lignes
```

**A savoir : le tableau Numpy (ndarray) est plus rapide pour les calculs scientifiques et donc la data science qu'une liste par exemple.**

## **Introduction à Numpy avec Python :**

Pour voir les dimensions et la forme global de son tableau :

`.shape()`

L'attribut shape retourné est un tuple, donc c'est une séquence, on peut accéder aux différentes valeurs du tuple en indiquant les index, shape[0] nous retourne le nombre de lignes par exemple et shape[1] les colonnes.

Avec Numpy il existe d'autres fonctions, voyons ici une liste de quelques unes d'entre elles :

*(voir la doc de Numpy pour en voir d'autres)*

<https://numpy.org/doc/stable/reference/routines.array-manipulation.html>

### **⇒ Fonctions Numpy**

`np.array(objet)` # crée un tableau NumPy à partir de l'objet que vous passez en argument

`np.full(shape, value)` # crée un tableau NumPy de forme spécifiée où toutes les valeurs sont égales à la valeur spécifiée.

`np.zeros(shape)` # crée un tableau NumPy de forme spécifiée rempli de zéros

`np.ones(shape)` # crée un tableau NumPy de forme spécifiée rempli de uns.

`np.eye()` # crée une matrice identité de taille N x N avec des uns sur la diagonale principale et des zéros ailleurs

`np.random.randn(shape)` # génère un tableau NumPy de forme spécifiée rempli de nombres aléatoires suivant une distribution

normale standard (moyenne 0 et écart-type 1) par exemple

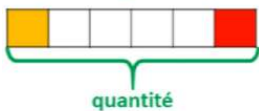
```
[0.12345678 -1.23456789]
```

`np.size()` # retourne le produit du nombre de lignes et de colonnes de notre tableau

### ⇒ Gestion des Tableaux

`np.linspace(début, fin, quantité)`

`np.linspace(début, fin, quantité)`



`np.arange(début, fin, pas)`

`np.arange(début, fin, pas)`



*(C'est 2 fonctions sont utiles pour matplotlib)*

### ⇒ Les Types de Format

Le type va donner plus ou moins de performance selon le format de ce type.

Pour définir un type on fait :

`dtype=float64` ou `dtype=float32` par exemple.

`np.float16`

$\pi = 3.14$

**Moins précis**  
**Plus rapide**

`np.float64`

$\pi = 3.1415$

926535897

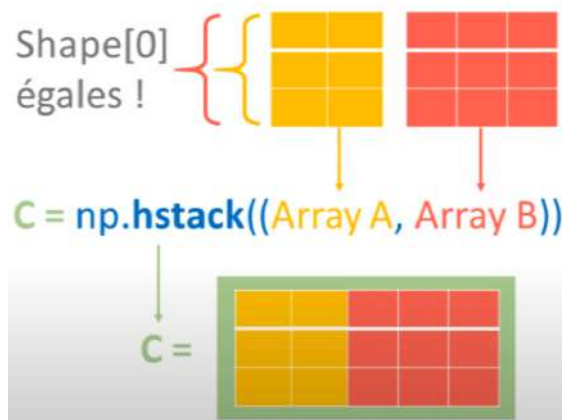
93

**Plus précis**  
**Moins rapide**

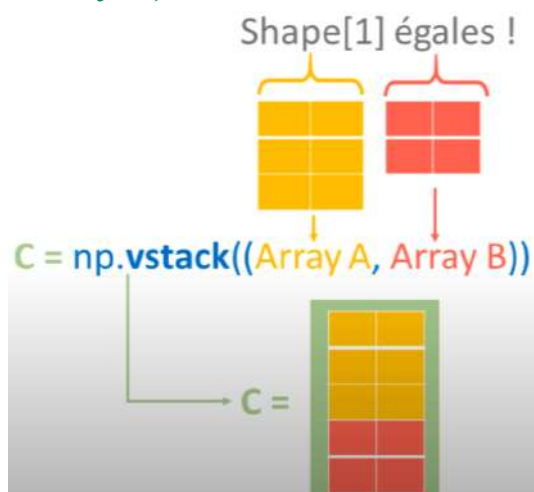
## ⇒ Les Gestions de Shape

Les **shapes** correspondent aux dimensions d'un tableau, il faut être rigoureux et précis pour éviter les erreurs de dimensions lorsqu'on manipule 2 tableaux différents !

`np.hstack((ArrayA, ArrayB))` # concatène deux tableaux NumPy horizontalement (en ajoutant les colonnes de ArrayB à la droite de ArrayA).

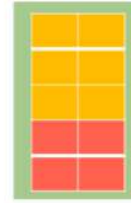


`np.vstack((ArrayA, ArrayB))` # concatène deux tableaux NumPy verticalement (en ajoutant les lignes de ArrayB en dessous de ArrayA)



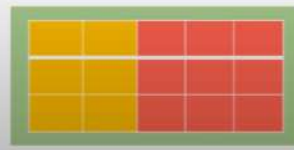
`np.concatenate((ArrayA, ArrayB), axis=0)` # concatène deux tableaux NumPy le long de l'axe 0 (verticalement)

```
C = np.vstack((Array A, Array B))
np.concatenate((Array A, Array B), axis = 0)
```



`np.concatenate((ArrayA, ArrayB), axis=1)` # concatène deux tableaux NumPy le long de l'axe 1 (horizontalement)

```
C = np.hstack((Array A, Array B))
np.concatenate((Array A, Array B), axis = 1)
```



`.squeeze()` # méthode en Python (utilisée souvent avec les tableaux NumPy) qui permet de supprimer les dimensions de taille 1 d'un tableau, réduisant ainsi sa dimension.

Exemple :

```
A = np.array([1, 2, 3])
A = A.reshape((A.shape[0], 1))
A.shape
```

```
(3, 1)
```

```
A = A.squeeze()
```

```
A.shape
```

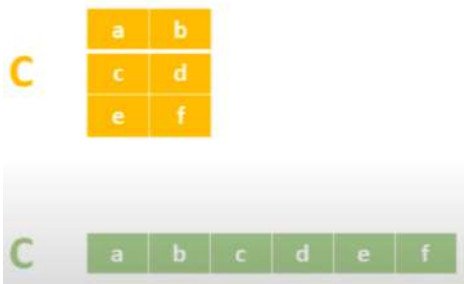
```
(3,)
```

Souvent en Machine Learning les algo peuvent avoir des problèmes quand on va faire un tableau en **1 dimension** on aura en **shape(3, )**, et ce vide donne lieu à des erreurs / bugs, donc on le **reshape** pour mettre un **1**, on reprend le shape de notre dimension 1 (lignes) et on met 1 pour préciser la dimension.

À l'inverse, si on veut par exemple observer une photo, créer des graphiques etc le 1 cette fois-ci peut causer des problèmes donc on va le **squeeze** pour le supprimer et retourner au **shape(3, )**.

`.ravel()` # est une méthode en Python (utilisée souvent avec les tableaux NumPy) qui aplatisse un tableau multidimensionnel en un tableau unidimensionnel sans créer de copie des données originales.

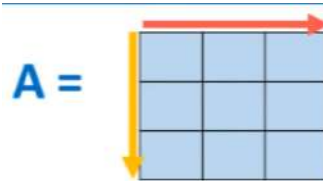
```
C = C.ravel()
```



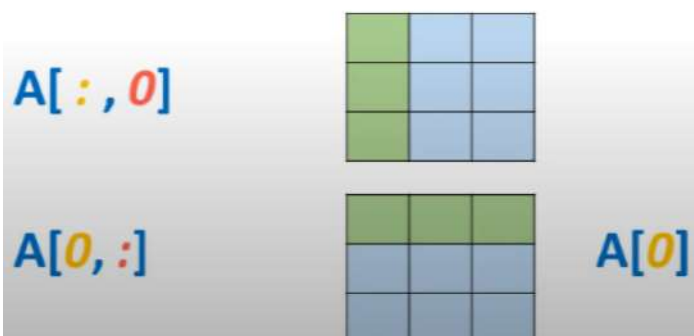
⇒ Sélection de données dans des tableaux (*indexing & slicing* avancé)



`A[ligne, colonne]`



`A[début : fin, début : fin]`



`A[0]`

$A[\text{début} : \text{fin}, \text{début} : \text{fin}]$

$A[1, :]$

$A[0:2, 0:2]$



$A[\text{début} : \text{fin} : \text{pas}, \text{début} : \text{fin} : \text{pas}]$

$A[::2, ::2]$


## ⇒ Le Boolean Indexing

```
A = np.random.randint(0, 10, [5, 5])  
A
```

```
array([[5, 7, 0, 0, 2],  
       [4, 1, 8, 2, 9],  
       [4, 9, 2, 6, 5],  
       [3, 4, 5, 2, 0],  
       [6, 3, 2, 0, 2]])
```

```
A < 5  
array([[False, False, True, True, True],  
       [ True, True, False, True, False],  
       [ True, False, True, False, False],  
       [ True, True, False, True, True],  
       [False, True, True, True, True]])
```

$A =$

5	4	5
4	4	4
5	4	5

$A < 5 \rightarrow$

F	T	F
T	T	T
F	T	F

$A[A < 5] = 10 \rightarrow$

5	10	5
10	10	10
5	10	5

## ⇒ Les Maths de bases en Machine Learning (pour coder)

*Rappel : axis=0 → lignes, axis=1 → colonnes*

### Fonctions mathématiques :

*(Voir la doc pour d'autres fonctions mathématiques..)*

<https://numpy.org/doc/stable/reference/routines.math.html>

`ndarray()` # récupère la somme du nombre d'attributs dans un tableau.

`sum()` # fait la somme d'éléments défini.



`argsort()` # méthode en NumPy qui renvoie les indices qui trieraient un tableau dans l'ordre croissant, en fonction des valeurs de ce tableau.

`np.exp()` # calcule l'exponentielle des éléments qu'on lui donne.

`np.log()` # calcule le logarithme naturel (logarithme en base e) d'un tableau ou d'une valeur numérique.

`np.cos()` # calcule le cosinus des éléments qu'on lui donne.

`np.sin()` # calcule le sinus des éléments qu'on lui donne.

`np.sinh()` # calcule la fonction sinus hyperbolique d'un tableau ou d'une valeur numérique.

`np.cumsum()` # calcule la somme cumulative des éléments d'un tableau, où chaque élément résultant est la somme des éléments précédents dans le tableau d'origine.

`np.prod()` # calcule le produit de tous les éléments d'un tableau NumPy donné.

`np.cumprod()` # calcule le produit cumulatif des éléments d'un tableau, où chaque élément résultant est le produit des éléments précédents dans le tableau d'origine.

`np.min()` # retourne la valeur la plus petite dans un tableau.

`np.max()` # retourne la valeur la plus grande dans un tableau.

`np.argmin()` # renvoie l'indice de l'élément le plus petit dans un tableau.

`np.argmax()` # renvoie l'indice de l'élément le plus grand dans un tableau.

`np.sort()` # trie le tableau dans l'ordre croissant

`np.sort(ascending=False)` # trie le tableau dans l'ordre décroissant

`np.argsort()` # renvoie les indices qui trieraient un tableau dans l'ordre croissant en fonction des valeurs de ce tableau.

`np.min(axis=0) =`

3	0	3
---	---	---

`np.min(axis=1) =`

0	3
---	---

```
np.random.seed(0)
A = np.random.randint(0, 10, [2, 3])
A
array([[5, 0, 3],
       [3, 7, 9]])
```

**Méthodes ndarray**

```
A.min(axis=0)
array([3, 0, 3])
```

`argmin(axis=0) =`

1	0	0
---	---	---

Retourne la position (index) du minimum

```
np.random.seed(0)
A = np.random.randint(0, 10, [2, 3])
A
array([[5, 0, 3],
       [3, 7, 9]])
```

**Méthodes ndarray**

```
A.argmin(axis=0)
array([1, 0, 0], dtype=int64)
```

## ⇒ Les Stats de bases en Machine Learning (pour coder)

(voir la documentation pour d'autres fonctions statistiques)

<https://numpy.org/doc/stable/reference/routines.statistics.html>

`np.min()` # retourne la valeur minimale d'un tableau.

`np.max()` # retourne la valeur maximale d'un tableau.

`np.mean()` # retourne la moyenne de toutes les valeurs d'un tableau.

`np.std()` # calcule l'écart-type (la déviation standard) d'un tableau NumPy, mesurant ainsi la dispersion des données par rapport à leur moyenne.

`np.var()` # calcule la variance d'un tableau NumPy, mesurant ainsi la dispersion ou la dispersion quadratique des données par rapport à leur moyenne.

`np.corrcoef()` # calcule la matrice de corrélation entre les différentes variables d'un tableau NumPy, permettant d'analyser les relations linéaires entre ces variables.

`np.unique()` # renvoie les valeurs uniques d'un tableau NumPy, éliminant ainsi les doublons, et permettant de déterminer le nombre d'occurrences de chaque valeur unique.

*A savoir :*

On peut ajouter devant les fonctions statistiques '**nan**', pour ignorer les valeurs qui sont des « nan » (not a number) dans notre tableau.

Exemple : `nanmean()` ou encore `nanstd()`

`.isnan()` # retourne toutes les valeurs nul et non nul en Booléens.

`.fillna()` # remplit les valeurs nul ou les supprime.

## ⇒ L'Algèbre Linéaire

(voir la documentation pour d'autres fonctions d'algèbre linéaire)

<https://numpy.org/doc/stable/reference/routines.linalg.html>

`X.T` # transpose la matrice X

`.dot()` # produit matricielle de 2 matrices exemple : `X.dot(Y)`

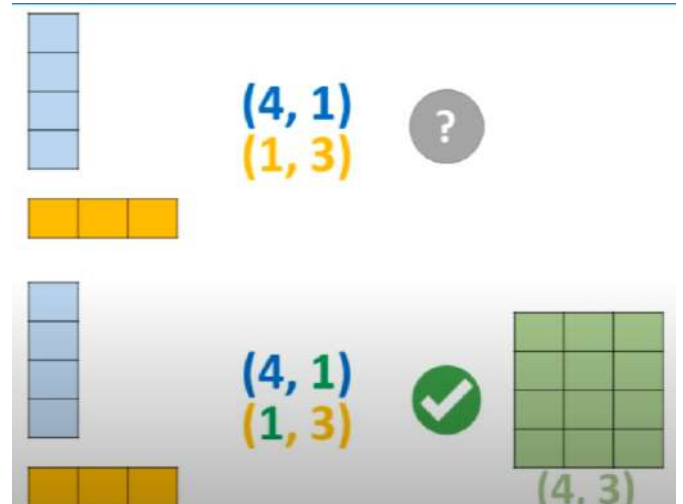
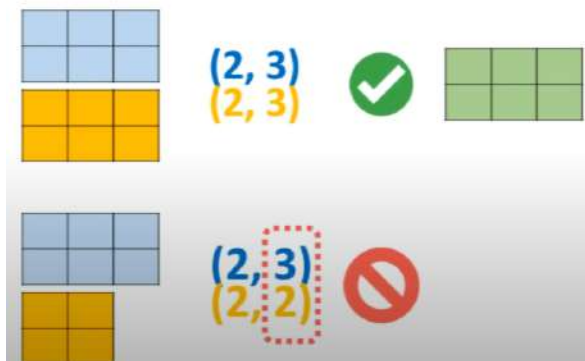
## ⇒ Le Broadcasting

*A Savoir :*

Le Broadcasting est parfois utilisé, il consiste à étendre les dimensions d'un tableau.

**La règle est simple:**

**A** et **B** dimensions égales ou égales à 1

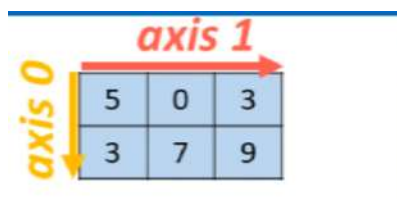


**ATTENTION** : dans cette méthode il faut bien que les dimensions soient claires, si on a (3, ) ça ne fonctionnera pas car ça va prendre le 3 pour l'index 1 et donc pour la dimension du tableau et pas les lignes !

Si on a (4, 1) et (3, ) c'est égale à (4, 1) et ( ,3) donc attention à ça !

*Utilisez le **.reshape()** pour clairement définir les dimensions !*

## ⇒ RÉSUMÉ DE NUMPY ⇐

	<b>Attributs utiles</b> A.shape A.size
<b>Constructeurs</b> np.array( <i>objet</i> , <i>dtype</i> ) np.zeros( <i>shape</i> , <i>dtype</i> ) np.ones( <i>shape</i> , <i>dtype</i> ) np.random.randn( <i>lignes</i> , <i>colonnes</i> )	<b>Méthodes utiles</b> A.min( <i>axis</i> ) A.max( <i>axis</i> ) A.sum( <i>axis</i> ) A.mean( <i>axis</i> ) A.std( <i>axis</i> )
<b>Manipulation</b> A.reshape ( <i>shape</i> ) A.ravel() A.concatenate( <i>axis</i> )	A.argmin( <i>axis</i> ) A.argmax( <i>axis</i> ) A.argsort( <i>axis</i> )

# Introduction à Matplotlib avec Python :

## ⇒ L’Affichage

(voir la doc pour plus de fonctions)

[https://matplotlib.org/stable/plot\\_types/index.html](https://matplotlib.org/stable/plot_types/index.html)

`plt.plot(x,y)` # affiche les données

`plt.show()` # affiche le graphique

`plt.scatter()` # affiche un nuage de points

**Attention** : vérifier que les x et y ont les mêmes dimensions avant d’afficher !

## ⇒ La Customisation de Graphique

`plot(x, y, label=..., lw=..., ls=..., c=...)`



Les possibilités sont infinies.

C’est bien le problème...

**Faites simple !**

## ⇒ Les Figures

`plt.figure()` # fait une figure vide c’est comme une feuille de travail

`plt.plot(x,y, label=‘label’)` # nous donne une figure avec une courbe, c’est le contenu, on rajoute un label pour notre `.legend()`

mais maintenant on peut changer la taille de la figure :

`plt.figure(figsize(12,6))`

et on peut rajouter d'autres plot, d'autres contenus dans notre figure :

```
plt.plot(x, x**3, label='autre label')
```

on peut aussi rajouter des extras après :

```
plt.xlabel('txt') # affiche un label sur l'axe x (existe aussi pour y)
```

```
plt.title('titre') # affiche un titre à la figure
```

```
plt.legend() # affiche une légend à la figure (s'il y a des labels)
```

```
plt.savefig('nom de la figure.png') # sauvegarde en photo la figure
```

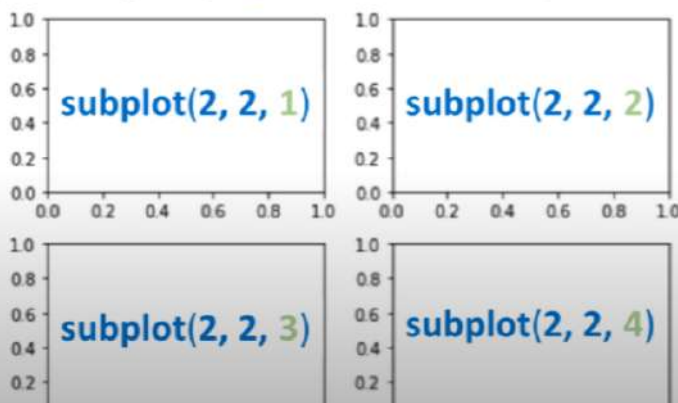
```
plt.show()
```

## ⇒ Le subplot

`plt.subplot(lignes, colonnes, position)` # génère une grille sur notre graphique pour afficher plusieurs figures

Une grille de graphiques :

```
plt.subplot(lignes, colonnes, position)
```



```
plt.figure()    <- Début de la figure
```

```
plt.subplot(2,1,1)
```

```
plt.plot(..., ...) } Graphique 1  
...  
plt.title()
```

```
plt.subplot(2,1,2)
```

```
plt.plot(..., ...) } Graphique 2  
...  
plt.title()
```

```
plt.show()    <- Affiche la figure
```



## ⇒ La POO pour nos figures (Programmation Orientée Objet)

```
fig, ax = plt.subplots()
ax.plot(x,y)
plt.show()
```

```
fig, ax = plt.subplots(n)
ax[0].plot(x, y)
ax[1].plot(x, y)
plt.show()
```

**fig** est un **objet**.

**ax** est un tableau **ndarray** qui contient des **objets**.

On utilise donc des **méthodes**.

## ⇒ Les Graphiques les plus utilisées

- `plt.imshow(np.corrcoef(x.T), cmap='Blues')`  
`plt.colorbar()`  
`plt.show()` # affiche un graphique de corrélation pour la transposée de la matrice X
- `plt.imshow(img)` # affiche une image
- `plt.contourf(x,y,z, cmap='RdGy')`  
`plt.colorbar()` # affiche les contours mais avec un mapping de couleur qu'on choisit au préalable
- `plt.contour(x,y,z, 20, colors='black')` # affiche les contours des reliefs
- `plt.hist(face.ravel(), bins=255)`  
`plt.show()` # affiche la fréquence d'apparition de point



- `plt.hist2d(x, y)`  
`plt.colorbar()` # affiche histogramme en 2d
- `plt.hist(x, bins=5)` # affiche un histogramme
- `from mpl_toolkits.mplot3d import Axes3D`  
`ax = plt.axes(projection='3d')`  
`ax.plot_surface(X,Y,Z, cmap='plasma')`  
# donne une projection 3D en une surface de courbe
- `from mpl_toolkits.mplot3d import Axes3D`  
`ax = plt.axes(projection='3d')`  
`ax.scatter(x,y,z, c=y)`  
# donne une projection 3D des nuages de points
- `plt.scatter()` # utilisé beaucoup pour les classifications

# Introduction à SciPy avec Python :

*(l'algèbre linéaire et les stats ont déjà été vu avec Numpy mais elles sont aussi présente dans SciPy si besoin)*

**La doc de SciPy :**

<https://docs.scipy.org/doc/scipy/reference/index.html>

## ⇒ **Module Interpolate**

Ce module est très utile quand on veut rajouter des données s'il en manque, par exemple :

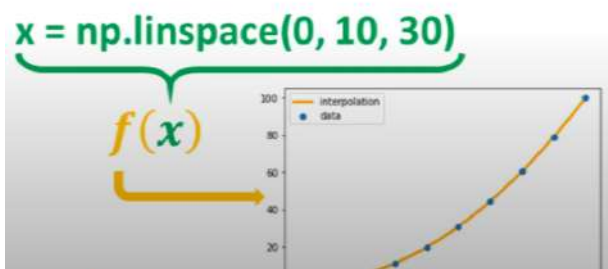
**Problème:** fréquences d'acquisition **différentes**



`from scipy.interpolate import interp1d` # importation de l'élément

**Interp1d** nous retourne  
une **fonction** :  $f(\dots) = \dots$

On peut utiliser cette fonction  
comme on veut :



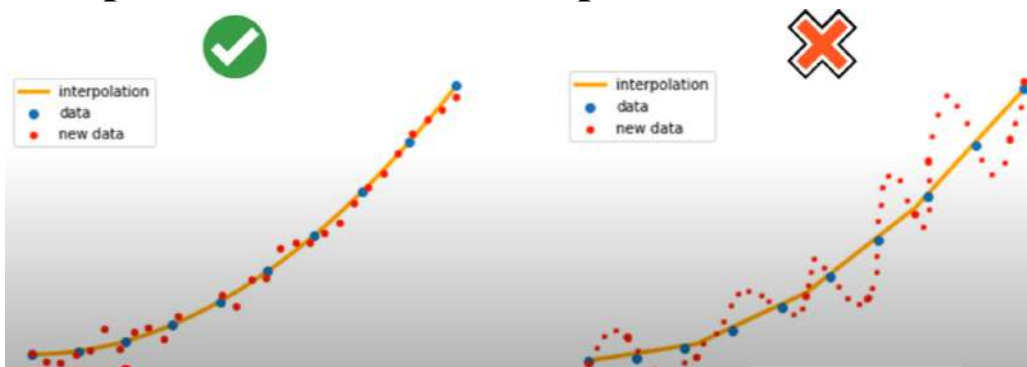
`f = interp1d(x,y, kind='linear')` # on place nos 2 données puis on choisit quels types d'interpolations on souhaite avoir (le plus basique c'est le linéaire)

(voir la doc pour voir les autres types d'interpolations :  
<https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.interp1d.html#scipy.interpolate.interp1d>  
)

```
new_x = np.linspace(0,10,30)  
result = f(new_x) # permet d'utiliser directement la fonction f qui  
va automatiquement utiliser interp1d et faire l'interpolation.
```

```
plt.scatter(x,y)  
plt.scatter(new_x, result, c='r') # affiche les points et  
l'interpolation
```

**Attention** : il faut être sûr et bien vérifier que notre interpolation ne nous cache pas la vérité !



*A Savoir* : il existe la même fonction mais en 2D, `interp2d(x,y,z)`  
→ elle fonctionne comme `interp1d`.

⇒ **Module Optimize**

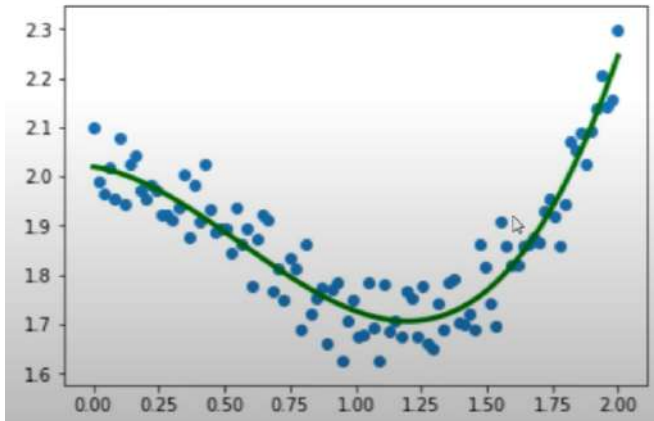
**La doc :**

<https://docs.scipy.org/doc/scipy/reference/optimize.html>

`optimize.curve_fit()`

Cette fonction utilise la méthode **des moindres carrés** pour trouver les **bons paramètres** d'un *modèle f*.

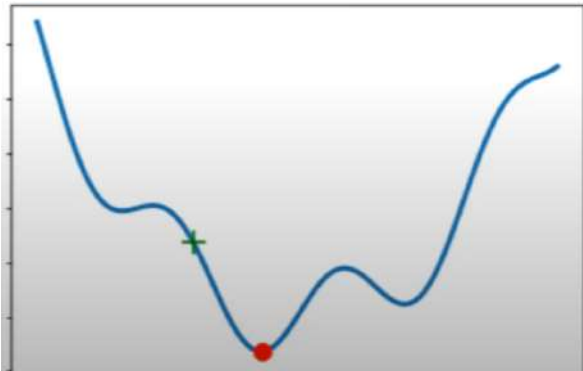
Par exemple la courbe va être optimisée sur les points et on aura :



**Précision** : *On peut utiliser Sklearn pour le faire automatiquement et plus rapidement (on verra ça plus tard).*

`optimize.minimize()`

Cette fonction exécute un algorithme de **minimisation**, selon un point de départ **x0** → trouve minimum **LOCAL**.



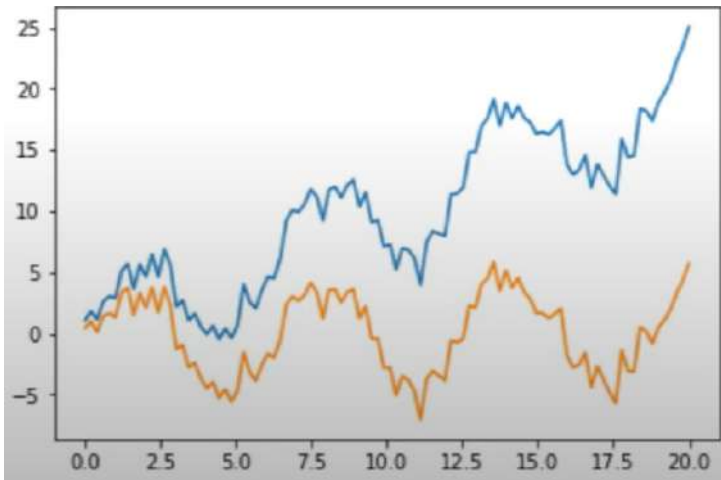
Parfois, pour trouver le minimum global, on peut juste changer le point de départ pour le trouver (ne fonctionne pas tout le temps mais ça peut être une solution).

De plus, **cette fonction est puissante car on peut l'utiliser sur autant de dimension que l'on souhaite !**

## ⇒ Traitement du Signal / fft

from scipy import signal

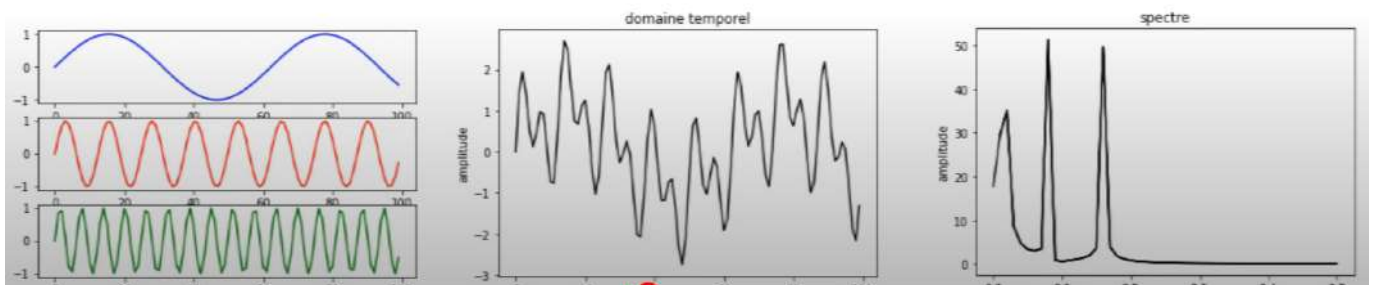
signal.detrend(y) # élimine la tendance linéaire d'un signal



On voit clairement que le signal en jaune c'est le `signal.detrend` car il n'y a plus de tendance linéaire

## ⇒ La Transformation de Fourier

Elle analyse les **fréquences** présentes dans un **signal périodique**.  
En résultat : Un **spectre**.



Exemple ici, le graphique **domaine temporel** est la **combinaison** des **3 graphiques** (3 signaux) **tout à gauche** et c'est ce que l'on retrouve tout autour de nous dans la vie.

La Transformation de Fourier permet de prendre un signal comme celui du **domaine temporel**, et d'en extraire les différentes fréquences qui l'a compose !

Le résultat nous donne un spectre (graphique nommé spectre) dans lequel en abscisse on observe les fréquences et en ordonnée les amplitude pour chaque fréquence.

Ici dans l'exemple, les 3 pics que l'on voit dans le spectre compose les 3 signaux différents qu'on a tout à gauche, on les retrouve dans notre spectre sous forme de pique.

**La doc :** <https://docs.scipy.org/doc/scipy/reference/fft.html>

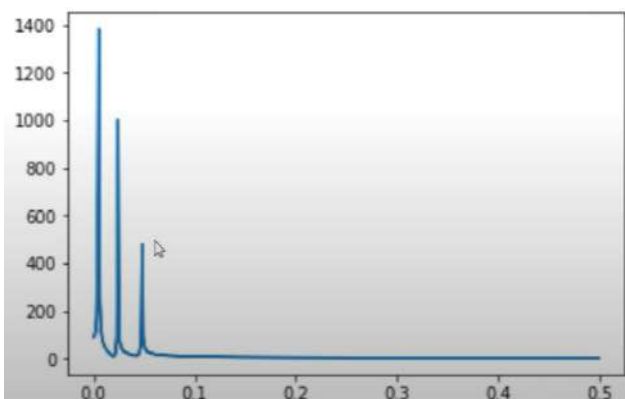
Le code pour implémenter l'explication ci-dessus de Fourier :

```
power = fftpack.fft(signal)  
freq = fftpack.fftfreq(signal.size)
```

En pratique le code ressemble à ça :

```
fourier = fftpack.fft(y)  
power = np.abs(fourier)  
frequencies = fftpack.fftfreq(y.size)  
plt.plot(np.abs(frequencies), power)
```

On obtient dans l'exemple :



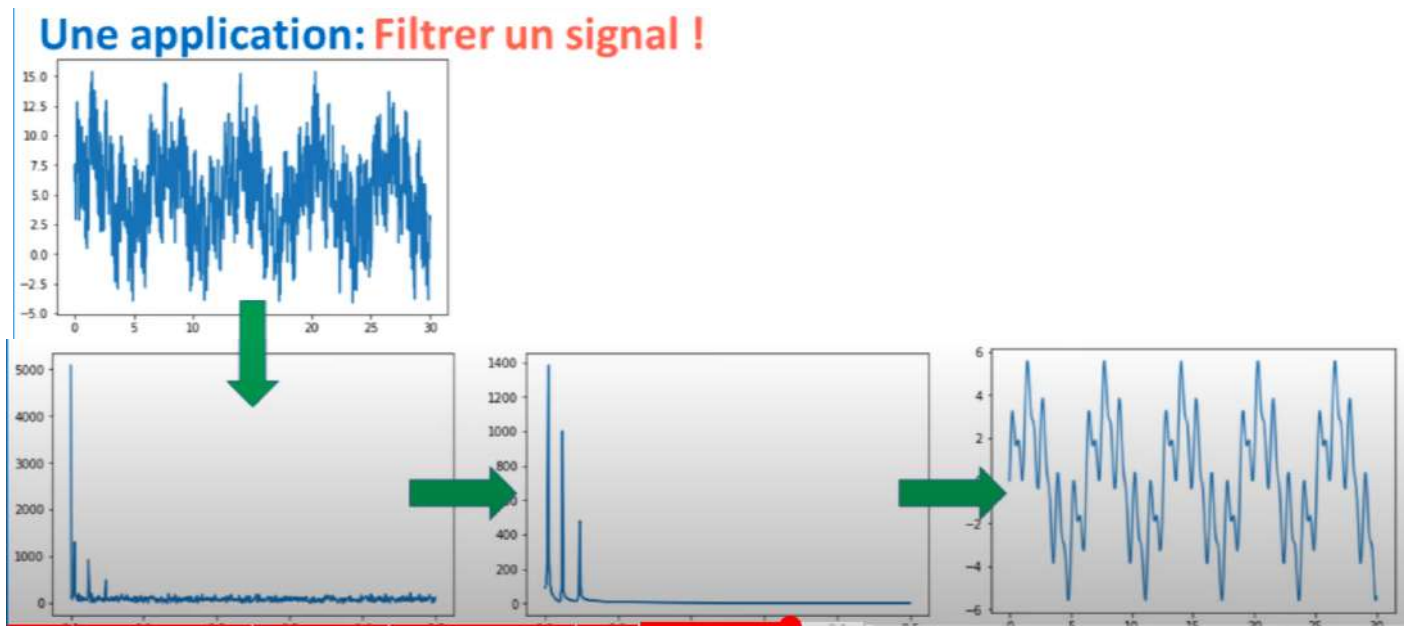
L'application et l'utilisation de Fourier et l'implémentation du théorème en code, permet de filtrer un signal rapidement en 3 étapes :

1- Faire transformation de Fourier pour produire le spectre de notre signal.

2- Utiliser du Boolean Indexing pour éliminer toutes les valeurs dans ce spectre qui sont inférieures à un certain seuil.

3- Utiliser ce spectre et appliquer la transformation de Fourier inverse, pour repasser dans le monde réel avec en sortie un signal parfaitement propre !

Illustrations :



Pour faire la transformation inverse on peut utiliser la fonction :  
`fftpack.ifft()` # inverse la fonction de fourier

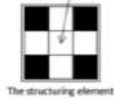
⇒ nd image (image processing)

La doc :

<https://docs.scipy.org/doc/scipy/reference/ndimage.html>

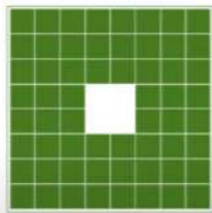
## La morphology :

Une structure



The structuring element

Se déplace sur  
une matrice



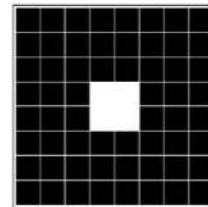
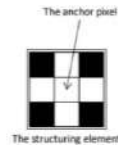
The source image

2 types d'opérations:

- **Dilation:** imprime des pixels
- **Érosion:** efface des pixels

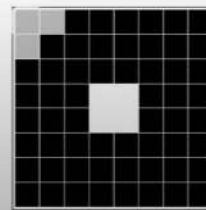
Figure 10.10

In this operation, new pixels are marked white. The structuring element slides over the source in the anchor pixel of the structuring element is at a white pixel in the source, the structuring element "impression". These pictures should clarify things.



The source image

I'll take you through the entire sliding process and we'll see the four pixels in the center dilate.



The source image

La morphology c'est une technique mathématique qui permet de transformer des matrices et donc des images (vu que les images sont composées de pixels et que chaque pixel est dans une matrice pour former l'image).

## Le Principe :

On définit une structure (le plus souvent ça ressemble à une croix) et cette structure va se déplacer de pixel en pixel, sur toute notre image. Lorsqu'elle va rencontrer un pixel blanc, cette structure va effectuer une opération, elle va soit imprimer tout autour d'elle des pixels blancs (selon son motif) c'est ce qu'on appelle la **dilation**, soit elle va effacer tout autour d'elle des pixels c'est ce qu'on appelle **l'érosion**.

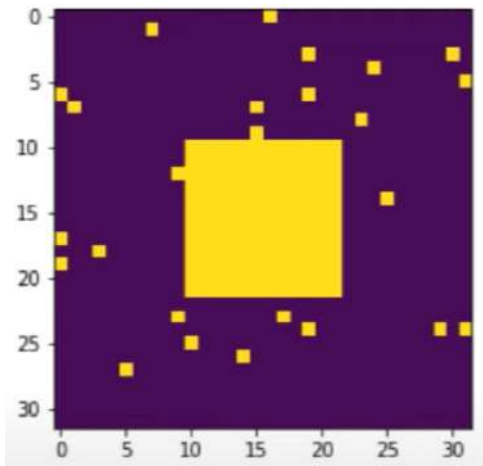
Ces techniques de morphology, peuvent être utilisés pour supprimer du bruit sur une image, donc des mauvais pixels, ou tâches etc qui sont sur une image qu'on aimerait lisser et donc enlever pour améliorer la qualité de l'image globale.



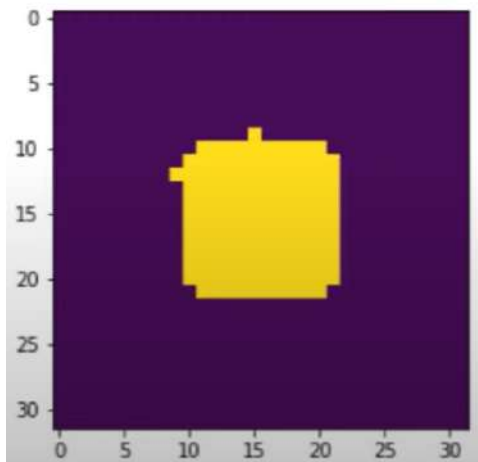
`open_x = ndimage.binary_opening(X) # combinaison entre une dilation et une érosion. On passe X ici qui est une matrice.`

Sur un exemple on aura :

Avant :



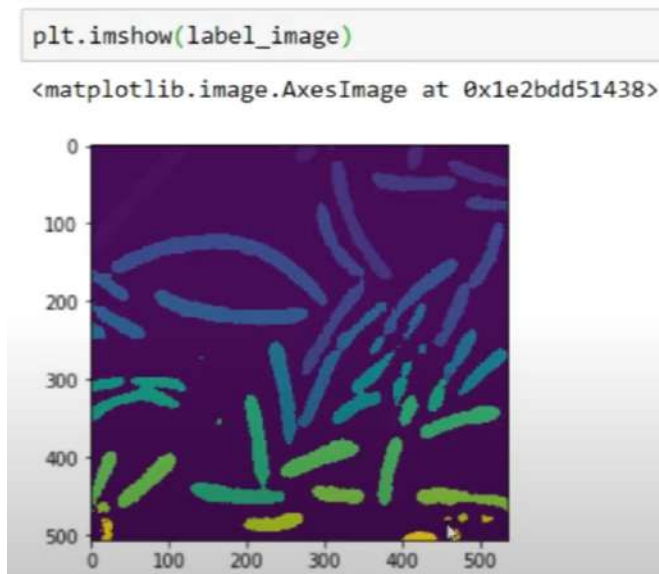
Après :



On a un peu perdu des informations car on voit que les coins ont un peu été touchés et donc on endommage un peu la structure qui nous intéresse mais ici rien de grave.

`ndimage.label(image) # permet de labelliser et donc d'étiqueté chaque élément qu'on cible sur une photo, par exemple s'il y a des bactéries ça va labelliser les bactéries.`

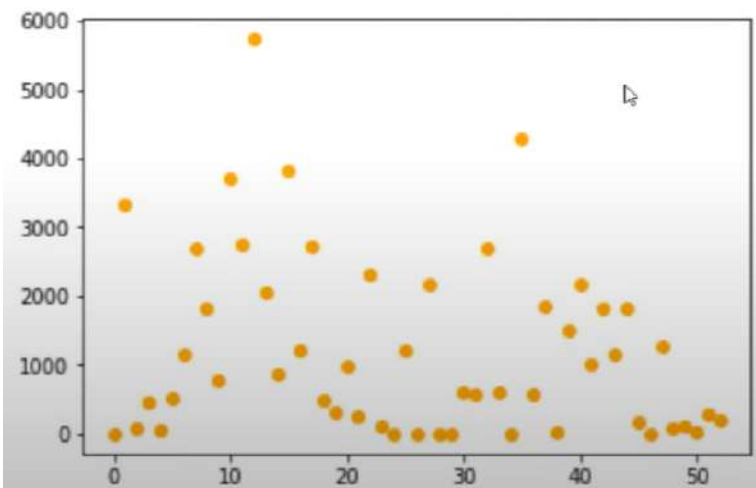
## Illustration :



`ndimage.sum(image, label, range())` # Permet de compter tous les pixels qu'il y a dans chaque groupe.

Donc si on reprend notre exemple ici, ça va permettre de mesurer la taille (relative) de chaque bactéries présente dans la photo.

## Illustration :



**Et avec ça, on peut l'utiliser pour faire office de Dataset et on peut commencer à l'utiliser pour faire des statistiques, des maths, prédictions etc !**

# **Introduction à Pandas avec Python :**

**⇒ Importer les données et les read**

**La doc :**

[https://pandas.pydata.org/docs/user\\_guide/index.html#user-guide](https://pandas.pydata.org/docs/user_guide/index.html#user-guide)

```
import pandas as pd
pd.read_excel('nom_fichier.xls')
pd.read_csv('nom_fichier.csv')
pd.read_json('nom_fichier.json')
pd.read_html('nom_fichier.html')
pd.read_sql('nom_fichier.sql')
pd.read....
(voir la doc pour en trouver d'autres)
```

**⇒ Les fonctions qu'on utilise énormément**

**(ici *df* = *pd.read\_csv('fichier.csv')*)**

```
df.head() # Affiche le début du Dataframe
df.describe() # Statistiques rapides du Dataframe
df.drop(['name_column1', 'name_column2'], axis=1) # Élimine
certaines colonnes
df.drop(['name_line1', 'name_line2'], axis=0) # Élimine certaines
lignes
```

```
df.dropna(axis=0) # Élimine les lignes aux données manquantes
df['column'].value_counts() # Compte les répétitions
df.groupby(['name_column']) # Analyse par groupe
```

```
df.shape # Nous donne les dimensions du Dataframe
df.columns # Affiche les noms des colonnes
```

`df.fillna(df['colonne4'].mean())` # Remplace les valeurs manquantes de la colonne4 par la moyenne des autres valeurs de cette colonne.

On peut remplacer par l'écart type ou d'autres statistiques.

`df.dropna(subset=['colonne4'], axis=0, inplace=True)` # Supprime les lignes où il y a des valeurs manquantes.

Le subset permet de designer qu'elle colonne on souhaite regarder où il y a des valeurs manquantes & le `inplace=True` permet de faire les modifications directement sur le Dataframe df sans réassigner à cette même variable le résultat (ou une autre).

### Exemple de l'utilisation de `inplace=True` :

```
data = data.dropna(axis=0)

# équivalent:

data.dropna(axis=0, inplace=True)
```

`df.dropna(axis=0)` # Supprime toutes les lignes où il y a des valeurs manquantes dans le Dataframe

*A Savoir* : Pandas fonctionne en collaboration avec Matplotlib.

`df.plot()` # Affiche le graphique du Dataframe

`df.plot.bar()` # Affiche un graphique en barre du Dataframe

`df.hist(bins=..)` # Affiche un histogramme du Dataframe

`df.plot.scatter(x=.., y=..)` # Affiche un nuage de point du Dataframe

`pd.plotting.scatter_matrix(df)` # Affiche une matrice de graphique avec plusieurs informations

**[Pour plus de performance on utilisera Seaborn]**

## ⇒ Les 2 Structures de données de Pandas

### 1- Les Séries

### 2- Les Dataframes

Series			Series			DataFrame	
apples			oranges			apples	oranges
0	3	+	0	0	=	0	3
1	2		1	3		1	2
2	0		2	7		2	0
3	1		3	2		3	1
							2

Quand on assemble plusieurs Séries ensemble ça nous donne un DataFrame.

### Mais qu'est ce qu'une Série ?

Série : tableau Numpy 1D + axe d'index

apples	
0	3
1	2
2	0
3	1

Axe index  
(indépendant de Numpy)

ndarray  
(contient son propre index)

NB : On peut choisir un autre index que 0, 1, 2, 3, ...

Pour changer le nom d'un index on fait :

`df.set_index('le_nouveau_nom')`

On peut donc assembler plusieurs Séries quand elles partagent le même axe :

apples			oranges			apples		oranges
0	3	+	0	0	=	0	3	0
1	2		1	3		1	2	3
2	0		2	7		2	0	7
3	1		3	2		3	1	2

Un DataFrame c'est un peu comme un dictionnaire mais de Séries.

*Rappel* : dict['clef'] = valeur

**Df ['column'] = une Série**

	pclass	survived	sex	age
0	1	1	female	29.0000
1	1	1	male	0.9167
2	1	0	female	2.0000
3	1	0	male	30.0000
4	1	0	female	25.0000

⇒ **L'Indexing & Slicing dans Pandas**

**data ['age'] = Série** (ndarray)

**data ['age'][0:10]** (indexing)

**data ['age'] < 18** (mask)

**data[data ['age'] < 18]** (boolean indexing)

**data [ ['age', 'pclass']] = DataFrame**

**data.iloc [0:2, 0:2]** -> localisation par **index**

**data.loc[0:2 , 'age']**

⇒ **Le DateTime**

Pour avoir un index de type Datetime :

- index\_col = 'Date'
- parse\_dates = True

Ce qui donne dans le code :

**pd.read\_csv('nom.csv', index\_col='Date', parse\_dates=True)**

Le « `parse_dates` » permet de convertir en date les valeurs.  
Et on obtiendra à la fin sur nos données :

```
bitcoin.index
DatetimeIndex(['2011-10-04', '2011-10-05', '2011-10-06', '2011-10-07',
               '2011-10-08', '2011-10-09', '2011-10-10', '2011-10-11',
               '2011-10-12', '2011-10-13',
               ...
               '2019-09-25', '2019-09-26', '2019-09-27', '2019-09-28',
               '2019-09-29', '2019-09-30', '2019-10-01', '2019-10-02',
               '2019-10-03', '2019-10-04'],
              dtype='datetime64[ns]', name='Date', length=2923, freq=None)
```

Grâce à cet Index, on peut faire de l'Indexing sur les Dates :

*Exemple sur un dataframe de bourse de bitcoin*

```
df_bitcoin['2019']['Close']
df_bitcoin['2019-09']['Close']
df_bitcoin['2019':'2023']['Close']
df_bitcoin['2019':'2023', 'Close']
```

Pandas possède une fonction pour les dates également :

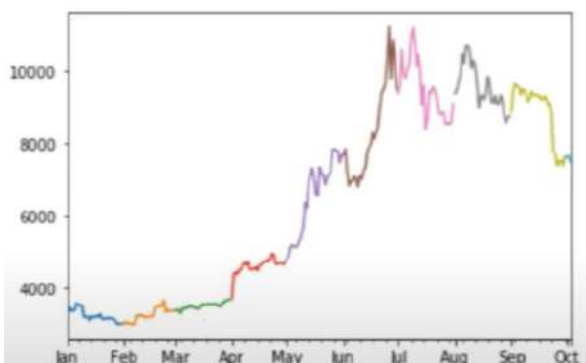
```
pd.to_datetime('2020/03/25') # Donnera un timestamp
```

**Le type `datetime64` peut aussi mesurer le temps.**

⇒ **Resample()**

Grâce à cette fonction de Pandas on peut regrouper nos données selon une **fréquence**.

Illustration :



On utilise pas mal la fonction **aggregate** par dessus la **resample**.

Aggregate (**.agg()**) elle permet de rassembler dans un seul tableau plusieurs statistiques qu'on fait sur la fonction resample.



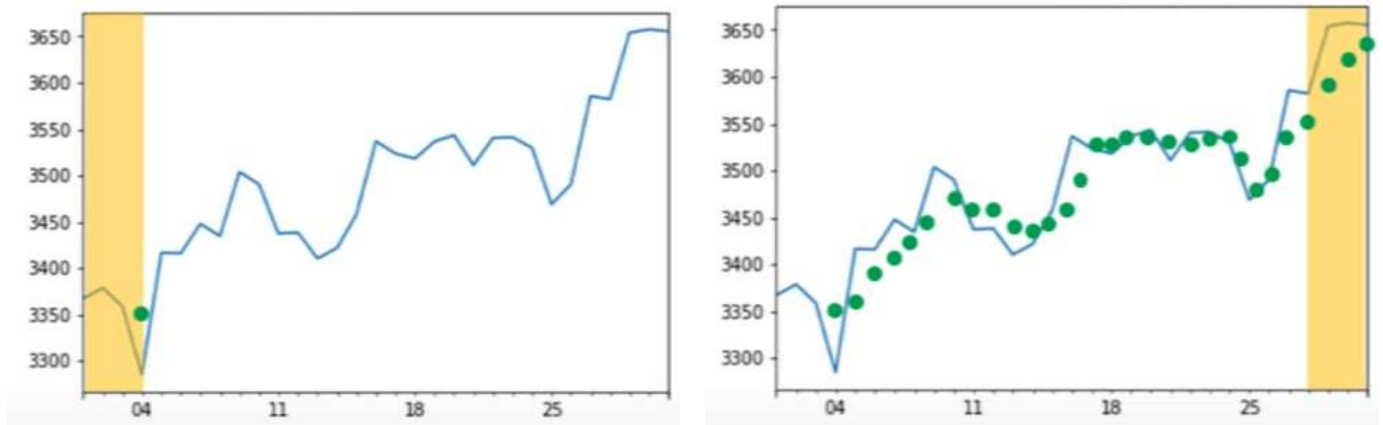
Illustration :

```
bitcoin.loc['2019', 'Close'].resample('W').agg(['mean', 'std', 'min', 'max'])
```

Date	mean	std	min	max
2019-01-06	3413.619995	86.412199	3345.330078	3565.800049
2019-01-13	3308.722830	187.370818	3090.370117	3513.979980
2019-01-20	3193.085728	42.170285	3129.989990	3263.669922
2019-01-27	3139.568534	20.884335	3110.709961	3171.270020
2019-02-03	3010.647182	21.591732	2986.850098	3054.060059

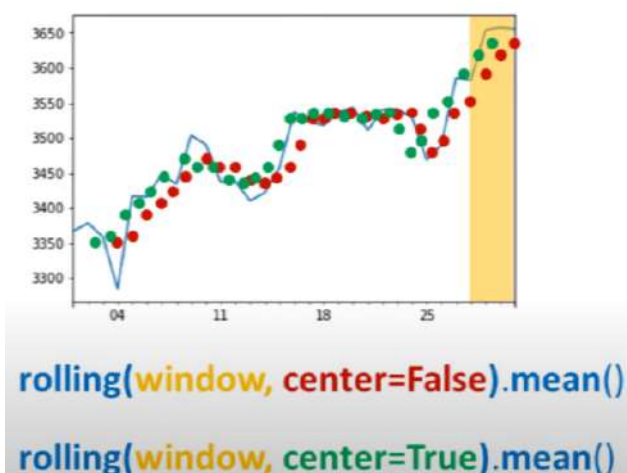
## ⇒ Moving Average

Au lieu de faire la moyenne de toutes les valeurs on fait la moyenne sur une fenêtre de valeurs.



On définit cette fonction « roulante » avec la fonction `pd.rolling(window)` # `window = 1..4..10..` etc bref on définit une valeur pour la taille de notre windows qu'on a.

Et pour centrer ou non la courbe de la moyenne on a :





## ⇒ Exponential Weighted Function

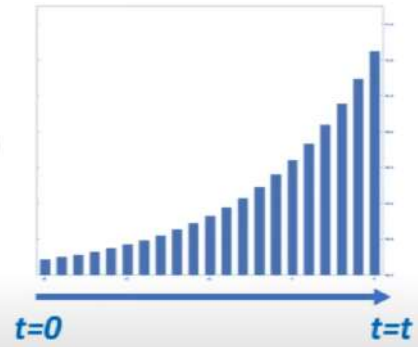
Une moyenne mobile exponentielle est donnée par la formule suivante :

Une **moyenne mobile exponentielle** est donnée par la **formule** suivante:

$$\bar{x}_t = \alpha (x_t + (1 - \alpha)x_{t-1} + (1 - \alpha)^2 x_{t-2} + (1 - \alpha)^3 x_{t-3} + \dots)$$

Avec  $x_t$  : la valeur de  $x$  à l'instant  $t$

et  $\alpha$  : le facteur de lissage ( $0 < \alpha < 1$ )



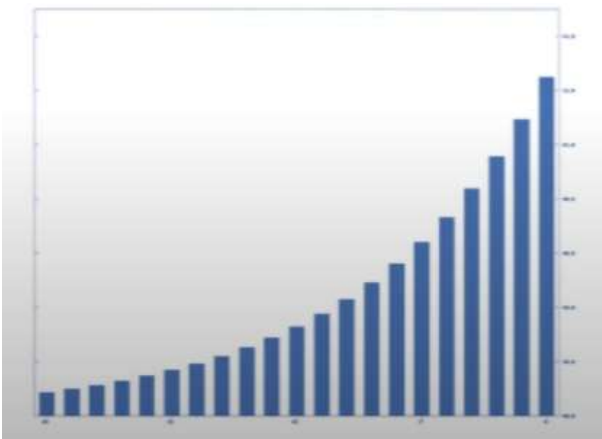
(Les valeurs perdent peu à peu du poids avec le temps)

**Autre façon de l'écrire:**

$$\bar{x}_t = \sum_{n=0}^{\infty} \alpha(1 - \alpha)^n x_{t-n}$$

(Fonction EWM de Pandas)

`ewm(alpha=0.5).mean()`



On a juste à définir une valeur pour notre alpha.

## ⇒ Assembler des Datasets

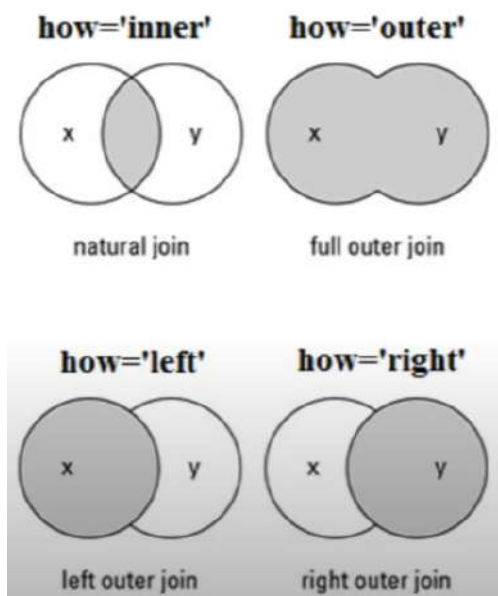
Il existe Merge et Join dans Pandas pour l'assemblage.

DataFrame			DataFrame		
	apples	oranges		apples	oranges
0	3	0	0	3	0
1	2	3	1	2	3
2	NaN	7			
3	NaN	2			

Ou bien

On peut **merger** des dataframes ou des séries de plusieurs manières : inner, outer, ...

On peut assembler des datasets dans un sens précis :



Le code de la fonction (exemple) :

```
pd.merge(df1, df2, on='Date', how='inner')
```

On peut changer les suffixes avec :

```
pd.merge(df1, df2, on='Date', how='inner', suffixes=('_data1', '_data2'))
```

### ⇒ Fonctions Map et Apply

La fonction **map()** applique une **fonction** sur chaque élément d'une **colonne**.

`Data['age'].map(lambda x : x+1)` # Va ajouter 1 an en + à chaque personne qui a son age dans le tableau

La fonction **apply()** applique une **fonction** sur chaque élément du **Dataframe**.

### ⇒ Transformer des catégories en valeurs numériques (dummies) pour pouvoir faire fonctionner nos algo de Machine Learning dessus


```
Data['sex'].map( {'male':0 , 'female':1} )
```

```
Data['sex'].replace(['male', 'female'] , [0,1] )
```

```
Data['sex'].astype('category').cat.codes
```



```
Name: sex, Length: 1309, dtype: category  
Categories (2, object): [female, male]
```

0
1
0
1
0

**Ces 3 fonctions différentes permet de transformer les valeurs des catégories en valeurs numériques.**

# Introduction à Seaborn avec Python :

⇒ Importation et utilisation

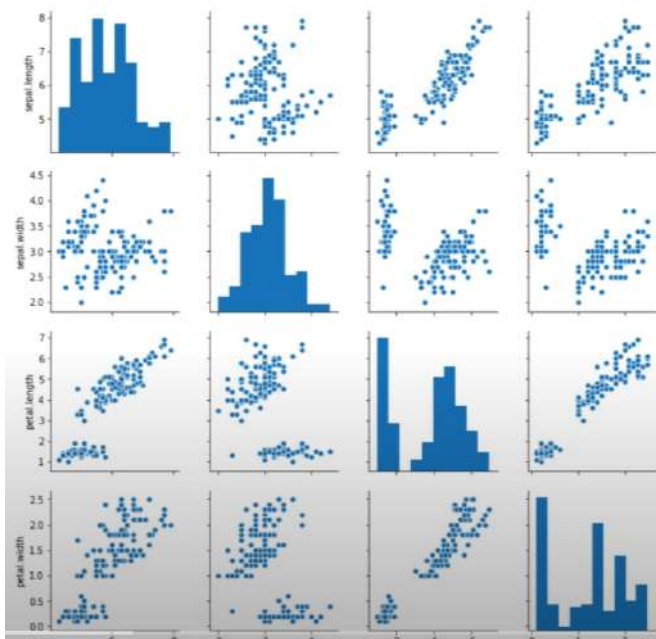
La doc :

<https://seaborn.pydata.org/examples/index.html>

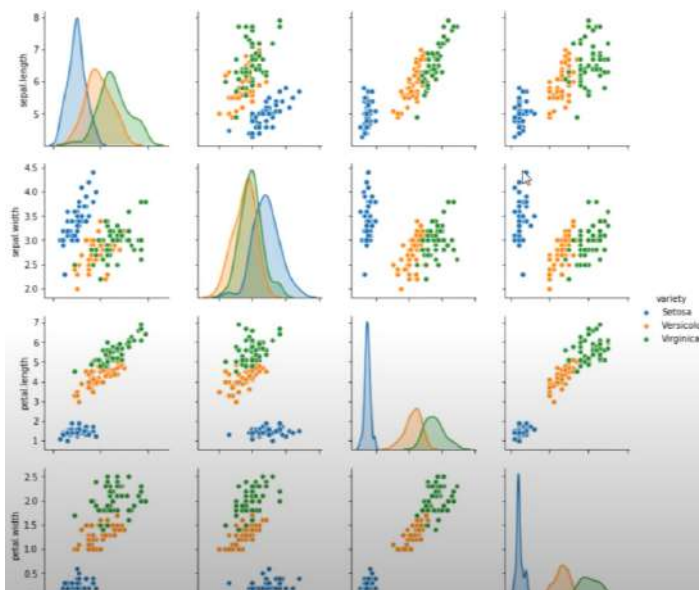
```
import seaborn as sns # importer la bibliothèque
```

Afficher plusieurs graphiques informatifs :

```
sns.pairplot(data1)
```



```
sns.pairplot(data1, hue='species') # affiche les stats selon les variétés (ici de fleurs)
```



## ⇒ Structure de presque toutes les fonctions Seaborn

Les fonctions Seaborn ont presque toutes la même structure:

**sns.fonction(x, y, data, hue, size, style)**

Les données  
à afficher

Options de  
segmentation

Exemple :

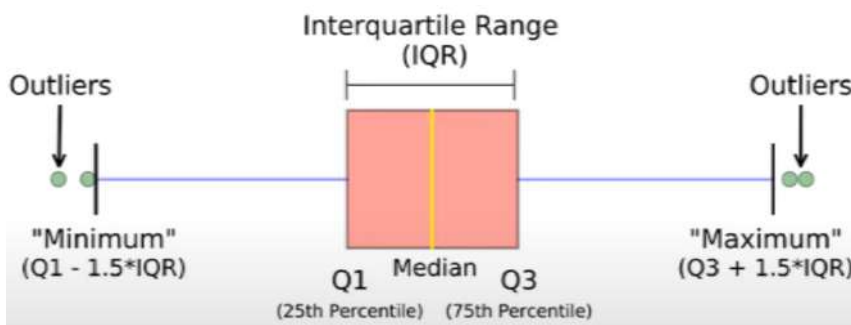
```
sns.catplot(x='survived', y='age', data=titanic, hue='sex')
```

## ⇒ Les Fonctions les plus utilisées avec Seaborn

```
→ sns.catplot(x='col1', y='col2', data=df, hue='var1')
```

```
→ sns.boxplot(x='col1', y='col2', data=df, hue='var1')
```

Explication du fonctionnement des boxplot :



```
→ sns.distplot(df['col1']) # visualise la distribution de la colonne1
```

```
→ sns.jointplot('col1', 'col2', data=df, kind='kde') # visualisation  
de la distribution selon 2 variables à la fois (il existe différents  
styles (hex par exemple) avec kind= voir la doc pour ça)
```

```
→ sns.heatmap(df.corr()) # permet de visualiser les matrices de  
corrélation
```

⇒ Quand utiliser Seaborn et quand utiliser Matplotlib

SEABORN	MATPLOTLIB
Data	Fonctions, matrices, etc.
Exploration Statistique	Mathématique, science, ingénierie, etc.
Vision globale	Graphique spécialisé

## Introduction à Sklearn avec Python :

La doc :

<https://scikit-learn.org/stable/>

⇒ 1-Fonctionnement de Sklearn & création d'un modèle

1. Sélectionner un **estimateur** et préciser ses **hyperparamètres** :

```
model = LinearRegression(.....)
```

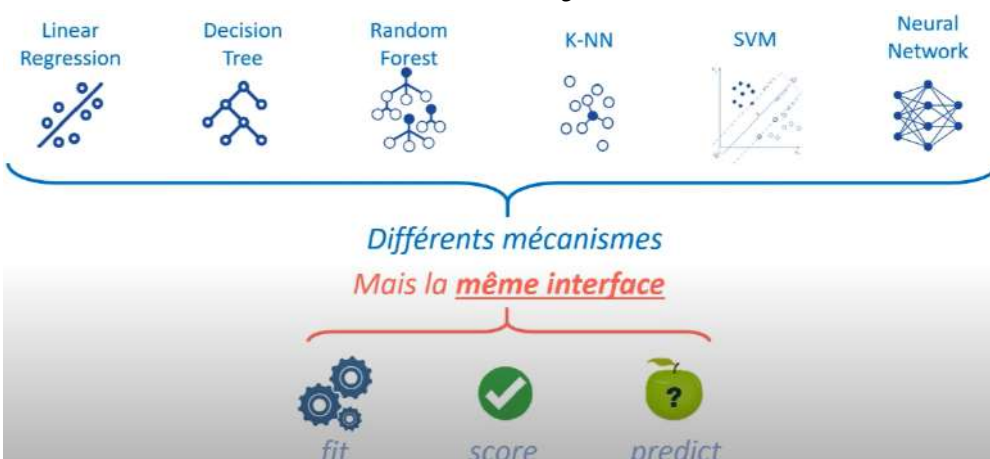
objet      Constructeur      Hyperparamètres

Exemples :

```
model = SGDRegressor(eta0=0.3) # Learning_rate = 0,3
```


```
model = RandomForestClassifier(n_estimators=100)
```

L'interface de Sklearn est toujours la même :




Exemple :

Linear Regression



```
model = LinearRegression()  
model.fit(X, y)  
model.score(X, y)  
model.predict(X)
```

Decision Tree



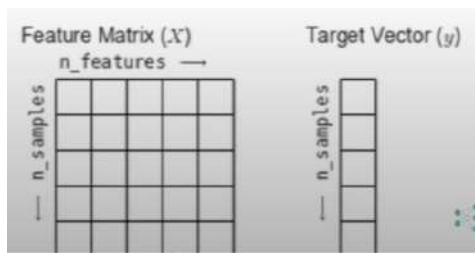
```
model = DecisionTreeClassifier()  
model.fit(X, y)  
model.score(X, y)  
model.predict(X)
```

## ⇒ 2-Entraînement du modèle dans Sklearn

On va entraîner le modèle sur les données X et y (divisées en 2 tableaux Numpy) :

**(X et y doivent toujours avoir 2 dimensions !)**

**[n\_samples, n\_features]**



**model.fit(X,y)**

## ⇒ 3-Évaluation du modèle dans Sklearn

Évaluer le modèle avec :

**model.score(X,y)**

## ⇒ 4-Utilisation du modèle dans Sklearn

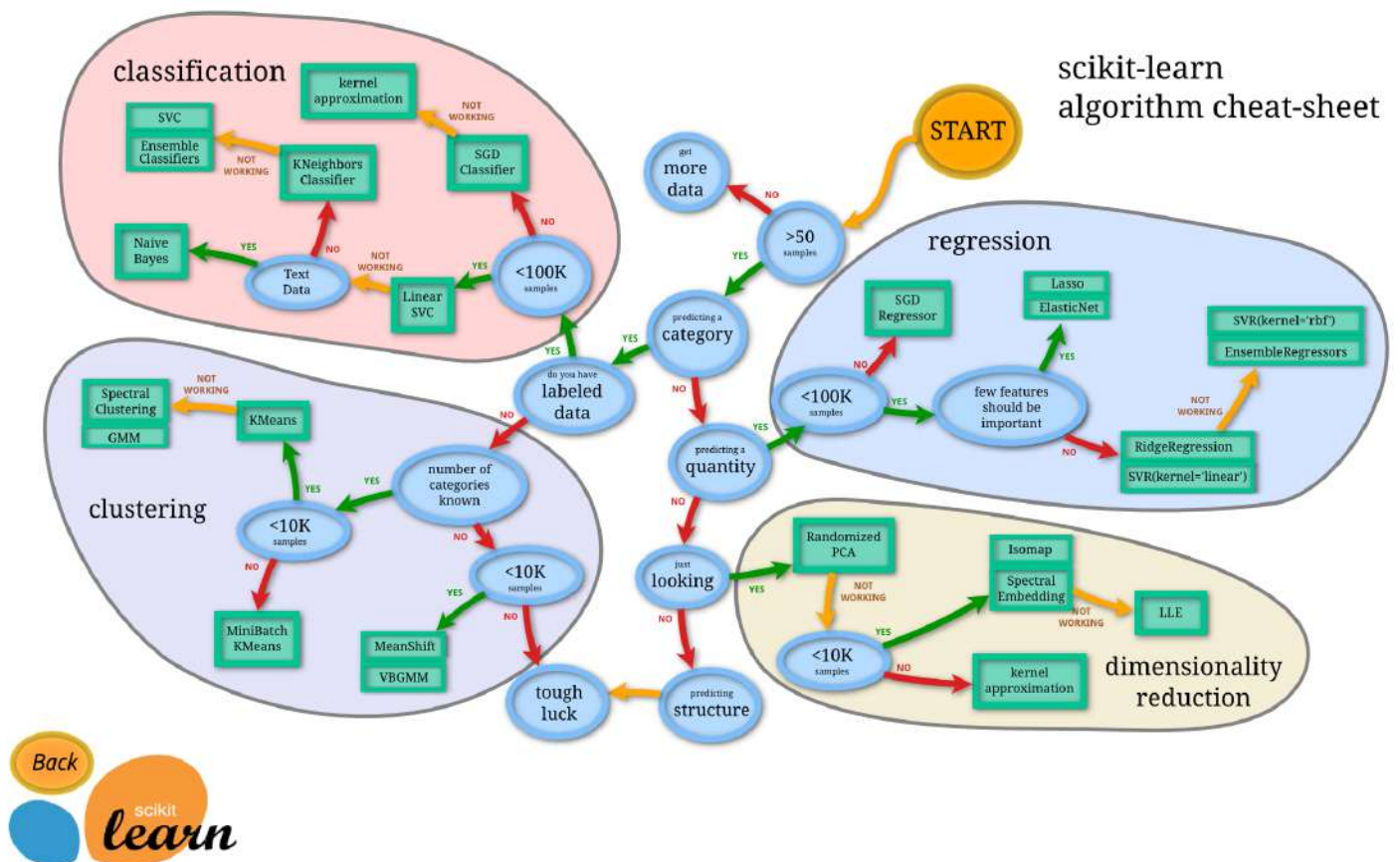
Utiliser le modèle avec :

**model.predict(X)**



**Attention** : Petite précision il faut pas juste importer sklearn tout seul, il faut importer chaque fonction que l'on souhaite utiliser avec par exemple : `from sklearn.linear_model import LinearRegression`

Carte pour vous aidez à choisir le meilleur algorithme selon vos besoins :

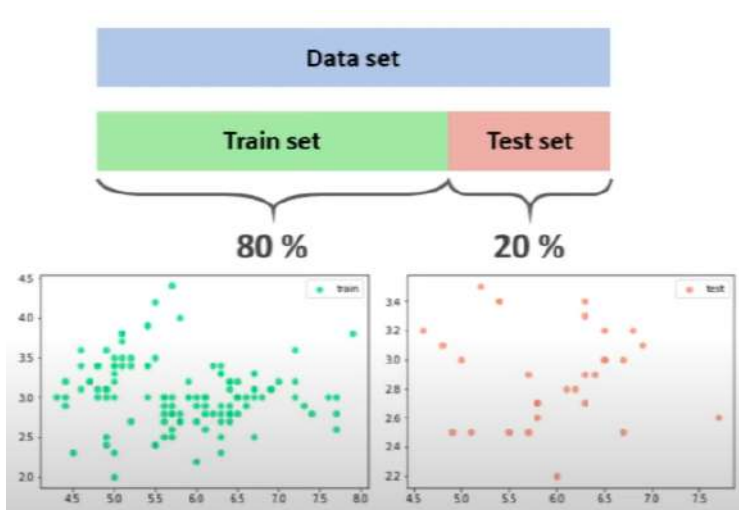


## ⇒ Train Test Split dans Sklearn

*Il ne faut jamais évaluer la performance d'un modèle sur les données d'entraînements, il faut évaluer la performance sur de nouvelles données sur lesquelles le modèle ne c'est pas entraîné !*

Et c'est pour cela qu'on divise toujours notre Dataset en Train et en Test d'où le Split.





**En règle générale, on divise en 80 % de données en Train et 20 % en Test.**

Ce qui nous donne :

```
model.fit(X_train, y_train)
model.score(X_test, y_test)
```

Pour Split on utilise donc la fonction `train_test_split()`  
`from sklearn.model_selection import train_test_split`  
`X_train, X_test, y_train, y_test = train_test_split(X, y,`  
`test_size=0.2)`

Le Dataset va d'abord être mélangé puis diviser en 20 % de test et 80 % de Train en données.

On peut contrôler l'aléatoire avec l'attribut `random_state=4` par exemple. C'est comme un **seed**, ça permet d'avoir toujours les mêmes données qui seront divisées pareil.

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=4)
```

Et ensuite on peut faire notre modèle et l'entraînement, par exemple :

```
from sklearn.neighbors import KNeighborsClassifier
model = KNeighborsClassifier(n_neighbors=1)
model.fit(X_train, y_train)
model.score(X_test, y_test)
```

## ⇒ Le Validation Set (Améliorer le modèle)

**Pour améliorer notre modèle et l'optimiser, il va falloir gérer nos hyper-paramètres et les faire varier jusqu'à obtenir les meilleurs qui donneront le meilleur modèle.**

Le Validation Set nous permet ici de chercher les réglages du modèle qui donnent les meilleures performances tout en gardant de côté les données du Test Set pour évaluer la machine sur des données qu'elle aura jamais vue.

Quand on veut comparer 2 modèles de ML, par exemple un `KNeighborsClassifier(n_neighbors=3)`

et un

`KNeighborsClassifier(n_neighbors=6)`

On va commencer par entraîner ces 2 modèles sur le Train Set puis on sélectionnera celui qui aura la meilleure performance sur le Validation Set, ensuite on pourra évaluer ce modèle sur notre Test Set afin d'avoir une idée sur sa performance dans la vraie vie.

## ⇒ La Cross-Validation

Cela consiste à entraîner puis valider notre modèle sur plusieurs découpes de données possible du Train Set.

Vu que les données sont choisis aléatoirement dans les découpes alors selon comment la machine prend aléatoirement les données pour apprendre elle peut apprendre plus ou moins vite, donc voici ce que fait la cross-validation :

	Train set	A	B
Split 1	Val Train Train Train Train	0.92	0.91
Split 2	Train Val Train Train Train	0.88	0.90
Split 3	Train Train Val Train Train	0.89	0.91
Split 4	Train Train Train Val Train	0.93	0.92
Split 5	Train Train Train Train Val	0.86	0.90

Au final, on va faire la **moyenne** des **5 scores** ici qu'on obtient, donc **0.89 | 0.92**, et donc quand on voudra comparer 2 modèles on sera sûr d'avoir choisi celui qui aura les meilleures performances.

Il existe plusieurs façon de découper le Train Set avec la technique de Cross-Validation (voir sur la doc de Sklearn), celle qu'on vient de voir s'appelle le **StratifiedKFold**.

Pour faire tout ce qui a été dit dans Python, on fait :

```
from sklearn.model_selection import cross_val_score
from sklearn.metrics import recall
cross_val_score(KNeighborsClassifier(), X_train, y_train, cv=8,
scoring='recall')
# cv = cross-validation on choisit ici de le split en 5 et le scoring
permet de choisir la métrique qu'on souhaite utiliser pour évaluer
notre modèle (exemple : 'accuracy', 'recall', 'precision' etc.
```

```
cross_val_score(KNeighborsClassifier(), X_train, y_train, cv=8,
scoring='recall').mean() # donne la moyenne
```

Maintenant on peut choisir le meilleur modèle, ici par exemple on test :

```
cross_val_score(KNeighborsClassifier(1), X_train, y_train, cv=8,
scoring='recall').mean()
cross_val_score(KNeighborsClassifier(2), X_train, y_train, cv=8,
scoring='recall').mean()
cross_val_score(KNeighborsClassifier(3), X_train, y_train, cv=8,
scoring='recall').mean()
cross_val_score(KNeighborsClassifier(4), X_train, y_train, cv=8,
scoring='recall').mean()
```

On va chercher à avoir la moyenne la plus haute pour avoir l'hyper-paramètre le plus performant et les meilleurs réglages.

Pour aller plus vite on peut faire ici :

```
val_score = []  
for k in range(0,50) :  
    score = cross_val_score(KNeighborsClassifier(4), X_train,  
                             y_train, cv=8, scoring='recall').mean()  
    val_score.append(score)  
plt.plot(val_score)
```

Mais il existe une fonction qui fait ça déjà pour nous, la validation curve.

### ⇒ La Validation Curve

```
validation_curve(model, X_train, y_train,  
                 'hyperparamètre', valeurs, cv=5)  
└───┬─── train_score, val_score
```

→ Teste toutes les **valeurs** pour un **hyper-paramètre** donné.  
Calcule le score sur **Train Set** et **Val Set** grâce à la  
**CrossValidation**.

Exemple :

**49 valeurs** sont testés avec **cv=5**, alors **train\_score** et **val\_score** sont de dimensions **(49,5)**.

En code en Python on fait donc :

```
from sklearn.model_selection import validation_curve  
model = KNeighborsClassifier()  
k = np.arange(1, 50)  
train_score, val_score = validation_curve(model, X_train, y_train,  
                                          'n_neighbors', k, cv=5)
```

**val\_score.shape** # on aura (49, 5), les 49 itérations qu'on a testés ici de 1 à 50, et 5 colonnes de notre cv (cross-validation)

`val_score.mean(axis=1)` # Pour avoir le score moyen pour chaque validation, on va récupérer la moyenne de chaque ligne donc `axis=1`.

`plt.plot(k, val_score.mean(axis=1))` # Affiche la moyenne de chaque ligne pour chaque itérations de l'hyper-paramètre, on met `k` en abscisse (c-a-d nos valeurs d'hyper-paramètre) et la moyenne en ordonnée).

On peut aussi voir le `train_score` en faisant :

`plt.plot(k, val_score.mean(axis=1), label='validation')`

`plt.plot(k, train_score.mean(axis=1), label='train')`

`plt.ylabel('score')`

`plt.xlabel('n_neighbors')`

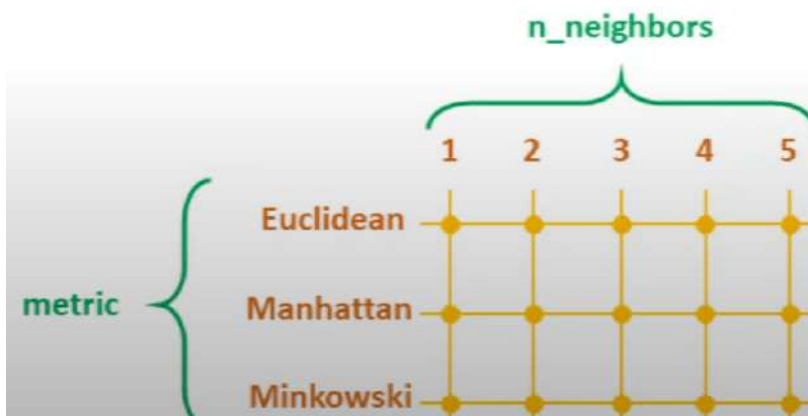
`plt.legend()`

Et ça c'est très utile pour détecter les cas **d'Overfitting** ! Si on a un très bon score en `train_score` mais un mauvais score en `val_score` c'est que le modèle a fait du Sur-apprentissage !

⇒ **GridSearchCV**

`Grid = GridSearchCV(model, param_grid, cv)`

**Permet de construire une grille de modèles avec toutes les combinaisons d'hyper-paramètres présents dans `param_grid`.**



En Python on fait :

```
from sklearn.model_selection import GridSearchCV
param_grid = {'n_neighbors':np.arange(1, 50),
              'metric':['euclidean', 'manhattan']}
grid = GridSearchCV(KNeighborsClassifier(), param_grid, cv=5)
```

La grille est un estimator !

→ `entraînement.fit()`

→ `best_score_`

→ `best_params_`

→ `best_estimator_`

`grid.fit(X_train, y_train)` # Entraînement

`grid.best_score_` # Voir le modèle qui a obtenu le meilleur score

`grid.best_params_` # Voir les meilleurs paramètres de ce modèle

`model = grid.best_estimator_` # Sauvegarde le modèle

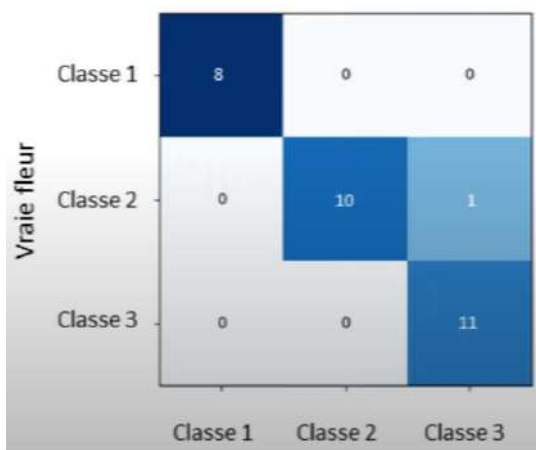
`model.score(X_test, y_test)` # Tester maintenant le modèle sur des nouvelles données

Il existe un outil de mesure très efficace ici c'est la

`confusion_matrix` :

→ Outil de mesure **très utile** pour évaluer la qualité d'un modèle de **classification**.

→ Montre les **erreurs** de **classement**.



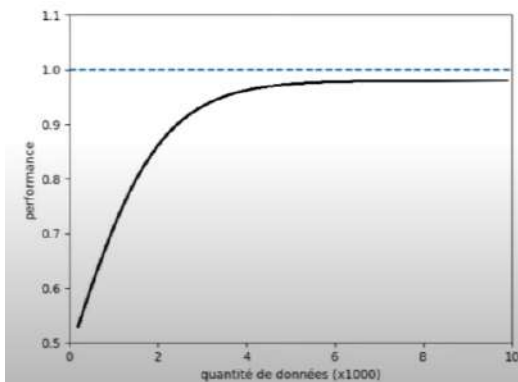
	Classe 1	Classe 2	Classe 3
Vraie fleur Classe 1	8	0	0
Classe 2	0	10	1
Classe 3	0	0	11

Avec Python on fait :

```
from sklearn.metrics import confusion_matrix  
confusion_matrix(y_test, model.predict(X_test))
```

## ⇒ Learning Curve

Les courbes d'apprentissage montrent l'évolution des performances du modèle, en fonction de la quantité de données qu'on lui fournit.



Normalement plus on fournit de données et meilleur la performance du modèle sera !

Pas tout le temps en fait. Voir jamais, au bout d'un moment la performance va plafonner et quand c'est le cas, inutile d'ajouter plus de données.

```
learning_curve(model, X, y, train_sizes, cv=5)
```

↳  $N$ ,  $train\_score$ ,  $val\_score$

$(X_{train}, y_{train})$

Train

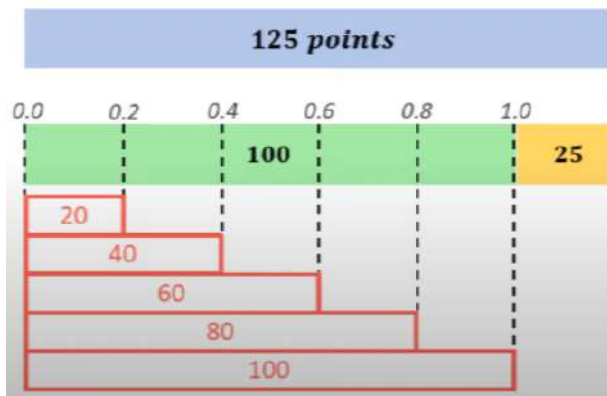
Val

En Python on fait :

```
from sklearn.model_selection import learning_curve  
learning_curve(model, X_train, y_train,  
train_sizes=np.linspace(0.2, 1.0, 5), cv=5)
```

# np.linspace(0.2, 1.0, 5) donne en pourcentage 5 lots de données de 20 à 100 % donc 0,2 / 0,4 / 0,6 / 0,8 / 1.

Ce qui nous donnera sur 125 points par exemple :



Et ces 20-40-60-80-100 on va les retrouver dans notre variable N retourné par notre learning\_curve :

```
N, train_score, val_score = learning_curve(model, X_train, y_train, train_sizes=np.linspace(0.2, 1.0, 5), cv=5)
```

Et si on le fait pour un Dataset donné on aura :

```
N, train_score, val_score = learning_curve(model, X_train, y_train, train_sizes=np.linspace(0.1, 1.0, 10), cv=5)
```

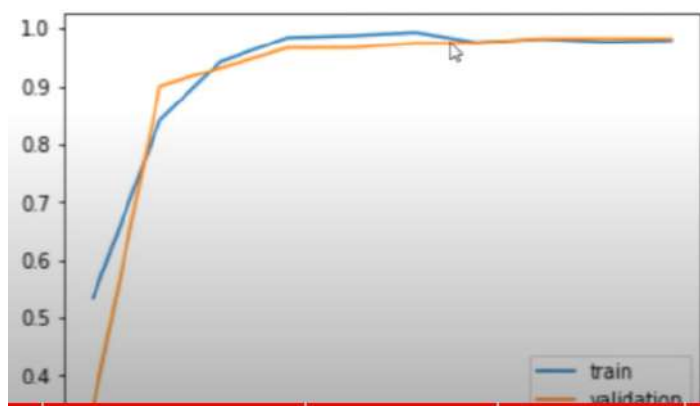
```
print(N)
```

```
plt.plot(N, train_score.mean(axis=1), label='train')
```

```
plt.plot(N, val_score.mean(axis=1), label='validation')
```

```
plt.xlabel('train_sizes')
```

```
plt.legend()
```



On constate bien qu'au bout d'un moment on a plus besoin de données supplémentaires les performances stagnent.



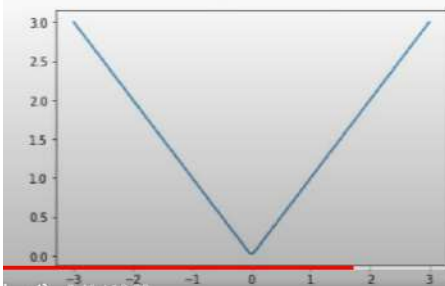
Et récolter des données dans la réalité ça coûte cher et c'est inutile de récolter des données pour un modèle qui ne s'améliorera pas.

!!!=> Utilisation de la MAE, RMSE ou MSE <=!!!

La MSE pénalise beaucoup plus les grandes erreurs que la MAE

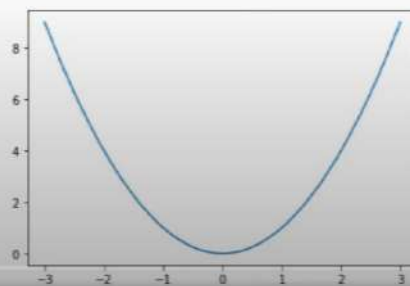
### Mean Absolute Error

l'importance d'une erreur est **linéaire** avec son amplitude



### Root Mean Squared Error

l'importance d'une erreur est **exponentielle** avec son amplitude



Donc si on prend l'exemple d'erreur sur une voiture qui doit freiner, si on utilise la MAE l'erreur ne sera que faiblement pénaliser donc on a un grand risque que le modèle en refasse donc on utilisera plus la MSE et surtout la RMSE pour ce rapprocher de la réalité.

MSE → vous accordez une grande importance aux grandes erreurs.

MAE → l'importance d'une erreur est linéaire avec son amplitude. Si le Dataset contient des valeurs **aberrantes** (outliers).

Il existe aussi la Median Absolute Error → très peu sensible aux **grandes erreurs**.

Pour les métriques de régression le plus souvent toutes les utilisés peut donner de bonnes statistiques et c'est efficace pour évaluer les performances de notre modèle.

**!!!=> Le Coefficient de Détermination  $R^2$  <=!!!**

$$R^2 = 1 - \frac{\sum (y_{vrai} - y_{pred})^2}{\sum (y_{vrai} - \overline{y_{vrai}})^2}$$

erreur quadratique  
variance

moyenne des  $y_{vrai}$

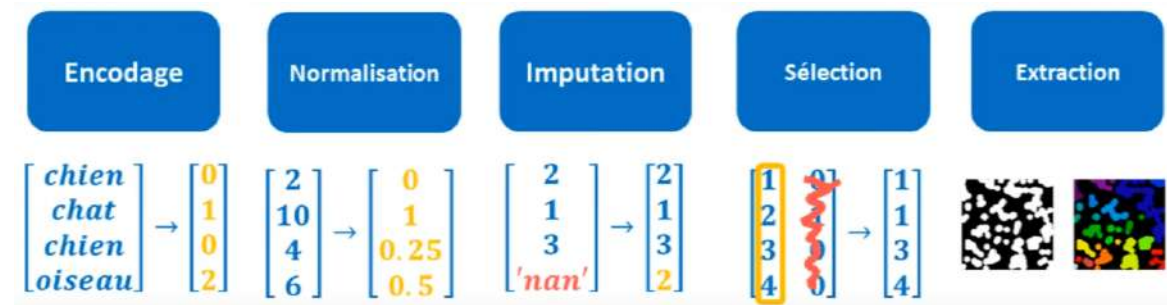
Évalue la **performance** du modèle par rapport au **niveau de variation** présent dans les données.

## Data Pre-Processing & Pipelines avec Python :

### ⇒ Le Data Pre-Processing

C'est une étape cruciale et essentielle dans tout projet de ML, elle consiste à trier et nettoyer les données proprement pour que la machine puisse avoir des données correctes à assimiler pour apprendre.

Il y a plusieurs choses que l'on peut faire en pre-processing :



- L'encodage : consiste à convertir les données qualitatives en valeurs numériques.

- Normalisation : permet de mettre sur une même échelle toutes les valeurs quantitatives.
- L'Imputation : permet de remplacer les valeurs manquantes ou de les supprimer.
- Sélection : utilise les tests statistiques comme le test de khi-deux pour sélectionner les variables les plus utiles au développement d'un modèle.
- Extraction : extraction de caractéristiques qui consiste à générer de nouvelles variables à partir d'informations cachées dans le Dataset.

### **Avec sklearn en Python :**

`sklearn.preprocessing` # module de preprocessing permettant de faire des transformations d'encodage, de normalisations et quelques autres fonctions..

`sklearn.impute` # imputer les valeurs manquantes

`sklearn.feature_selection` # sélectionne les variables les plus utiles

`sklearn.feature_extraction` # génère de nouvelles variables à partir d'informations cachées dans le Dataset

### **Utiliser les transformer pour le pre-processing :**

En Python on fait :

`import numpy as np`

`from sklearn.preprocessing import LabelEncoder`

`X = np.array(['Chat', 'Chien', 'Chat', 'Oiseau'])`

`transformer = LabelEncoder()`

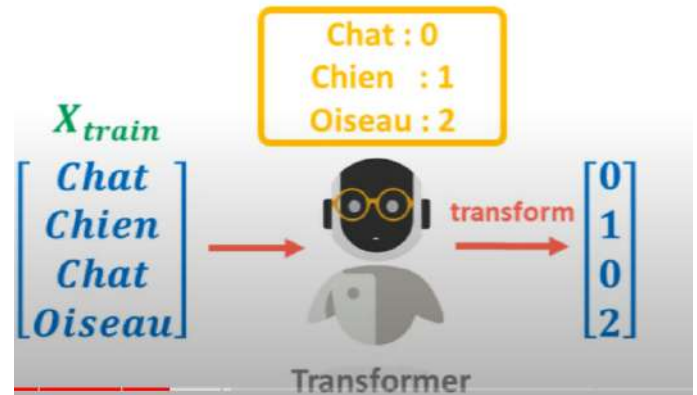
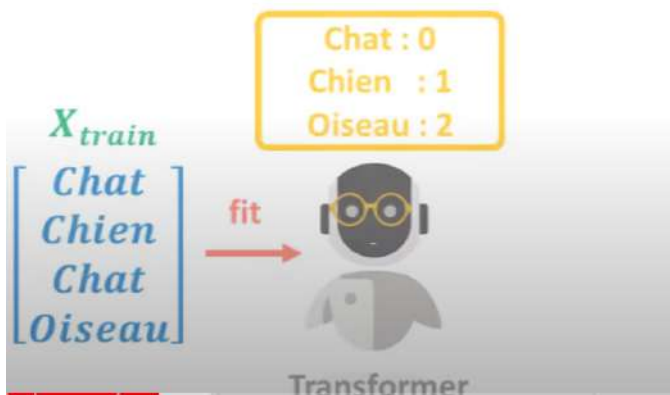
`transformer.fit(X)` # permet de développer une fonction de transformation en analysant les données du Train Set

`transformer.transform(X)` # applique une transformation sur toutes les données qu'on lui fournit (Train Set, Test Set et mêmes les données futures)

`transformer.fit_transform(X)` # combine la méthode fit et transform en une seule méthode

**fit( $X_{train}$ )** : développe une fonction de **transformation** à partir de  $X_{train}$

**transform( $X$ )** : applique la **transformation** sur les données  $X_{train}$ ,  $X_{test}$  et toutes autres données futures.



## L'encodage :

Pour faire des calculs, une machine a besoin de valeurs numériques.

Il existe 2 types d'encodage :

- Encodage Ordinal : associe chaque catégorie ou classe d'une variable à une valeur **décimale unique**.

Exemple :



Pour ça dans sklearn on dispose de **LabelEncoder()** et de **OrdinalEncoder()**.

**LABEL ENCODER**

Encode chaque classe de la **variable y** en une **valeur numérique** (0,  $n_{classe}-1$ )

**y**

Chat	0
Chien	1
Chat	0
Oiseau	2

Note : la méthode **inverse\_transform** permet de **décoder** les données

Mais si on a un tableau qui contient plusieurs variables le `LabelEncoder()` ne va pas fonctionner. Il faut ici utiliser le `OrdinalEncoder()`.

### ORDINAL ENCODER

Encode les catégories des **variables X** en **valeurs numériques** (0, n\_classe-1)

$X_1$	$X_2$	
Chat	Poils	→ $\begin{bmatrix} 0 & 1 \\ 1 & 1 \\ 0 & 1 \\ 2 & 0 \end{bmatrix}$
Chien	Poils	
Chat	Poils	
Oiseau	Plumes	

Note : C'est l'équivalent de `LabelEncoder`, mais pour les **features X**

- Encodage One-Hot : Ne peut pas comparer les catégories entre elle car chaque catégorie aura sa dimension (contrairement à l'encodage Ordinal), permet de mettre sous forme binaire les différentes valeurs des différentes catégories.

**Illustration :**

	Chat	Chien	Oiseau
Chat	1	0	0
Chat	1	0	0
Chien	0	1	0
Oiseau	0	0	1
Chien	0	1	0

Pour ça sklearn dispose de `LabelBinarizer()`, `MultiLabelBinarizer()` et `OneHotEncoder()`.

### LABEL BINARIZER

Encode chaque classe de la **variable y** en **One-Hot**

$y$	
Chat	→ $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$
Chien	
Chat	
Oiseau	

Note : la méthode `inverse_transform` permet de **décoder** les données

### ONEHOT ENCODER

Encode les catégories des **variables X** en **One-Hot**

$X_1$	$X_2$	
Chat	Poils	→ $\begin{bmatrix} 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \end{bmatrix}$
Chien	Poils	
Chat	Poils	
Chat	Poils	
Oiseau	Plumes	

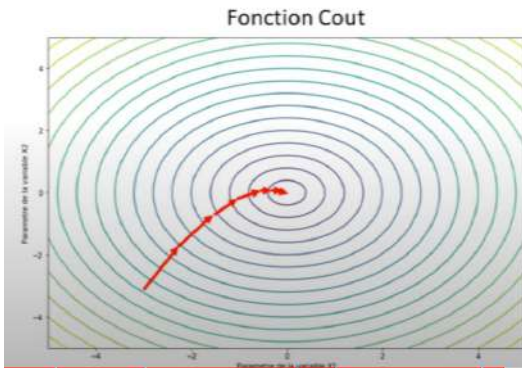
Note : C'est l'équivalent de `LabelBinarizer`, mais pour les **features X**



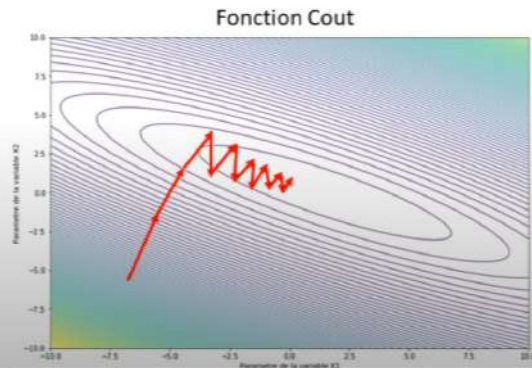
## La Normalisation

On encode les variables **qualitatives** et on normalise les variables **quantitatives**.

Avec Normalisation



Sans Normalisation



On voit ici que sans la normalisation il est très difficile de converger vers le minimum quand une variable prend le dessus. D'où l'importance de mettre toutes les valeurs sur une même échelle avant de passer les données dans la machine.

Avec sklearn on va voir les 3 plus utilisés :

- Normalisation MinMax
- La Standardisation avec le StandardScaler
- Le RobustScaler

### MINMAXSCALER

Transforme **chaque variable X** de telle sorte à être comprise entre **0 et 1**.

$$X_{scaled} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

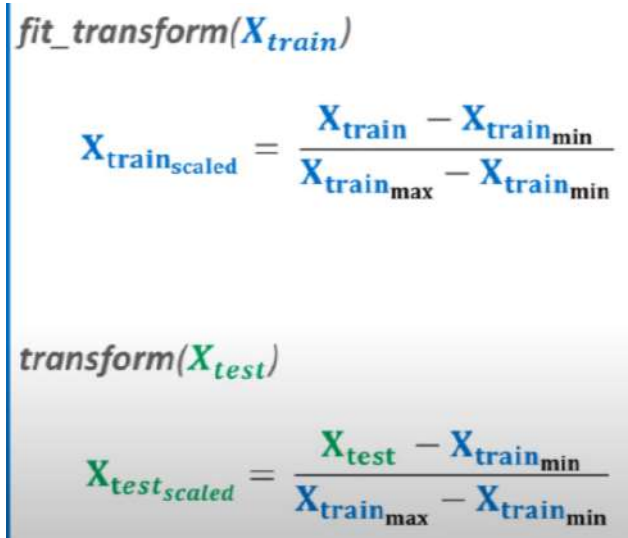
Exemple:

$$\begin{bmatrix} 70 \\ 80 \\ 120 \end{bmatrix} \rightarrow \begin{bmatrix} 0 \\ 0.2 \\ 1 \end{bmatrix}$$

**Note :** Pour remettre vos prédictions à leur véritable échelle : **inverse transform**

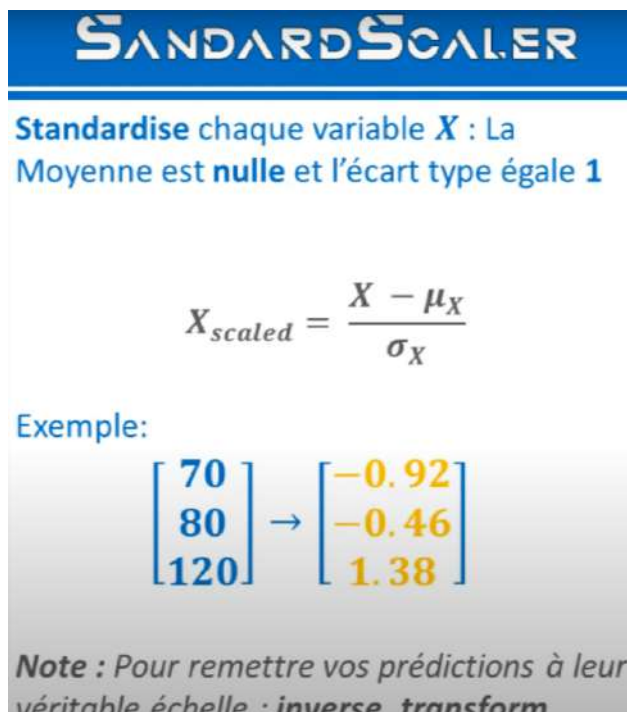
Quand on va vouloir tester sur de nouvelles données il faudra juste ne pas oublier de mettre à l'échelle qu'on a utilisé pour normaliser (de scaler) nos nouvelles données.

Dans la méthode transform de notre scaler on va trouver le calcul suivant :



The image shows a presentation slide with a blue vertical bar on the left. It contains two sections. The first section is titled `fit_transform( $X_{train}$ )` and shows the formula: 
$$X_{train\_scaled} = \frac{X_{train} - X_{train\_min}}{X_{train\_max} - X_{train\_min}}$$
 The second section is titled `transform( $X_{test}$ )` and shows the formula: 
$$X_{test\_scaled} = \frac{X_{test} - X_{train\_min}}{X_{train\_max} - X_{train\_min}}$$

Autre technique, le StandardScaler :




The image shows a presentation slide with a blue header that says "STANDARD SCALER". Below the header, it says "Standardise chaque variable  $X$  : La Moyenne est nulle et l'écart type égale 1". Then it shows the formula: 
$$X_{scaled} = \frac{X - \mu_X}{\sigma_X}$$
 Below the formula, it says "Exemple:" and shows a transformation: 
$$\begin{bmatrix} 70 \\ 80 \\ 120 \end{bmatrix} \rightarrow \begin{bmatrix} -0.92 \\ -0.46 \\ 1.38 \end{bmatrix}$$
 At the bottom, it says "Note : Pour remettre vos prédictions à leur véritable échelle : *inverse transform*".

**Mais ces 2 techniques vue précédemment sont très sensibles aux Outliers (valeurs aberrantes)**

C'est pour ça qu'on peut faire appel à un transformeur très peu sensible aux outliers, le **RobustScaler** :

**ROBUSTSCALER**

Transforme chaque variable  $X$  en étant peu sensible aux **outliers**

$$X_{scaled} = \frac{X - \text{mediane}}{IQR}$$


Exemple:

$$\begin{bmatrix} 70 \\ 80 \\ 120 \end{bmatrix} \rightarrow \begin{bmatrix} -0.4 \\ 0 \\ 1.6 \end{bmatrix}$$

## PolynomailFeatures

Très utile pour créer des variables **polynomiales** à partir de nos variables **existantes**, on appelle ça faire du **feature engineering**. Avec ça on peut développer des modèles de ML plus riches et plus sophistiqués.

**POLYFEATURES**

Crée de nouvelles variables **polynômiales** à partir des variables existantes.

Exemple : 1 variable  $x \rightarrow$  Polynôme 2

$$x \rightarrow 1 \quad x^1 \quad x^2$$

$$\begin{bmatrix} 1 \\ 2 \\ 0.5 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 0.5 & 0.25 \end{bmatrix}$$

La machine apprend alors un modèle :

$$f(x) = ax^2 + bx + c$$

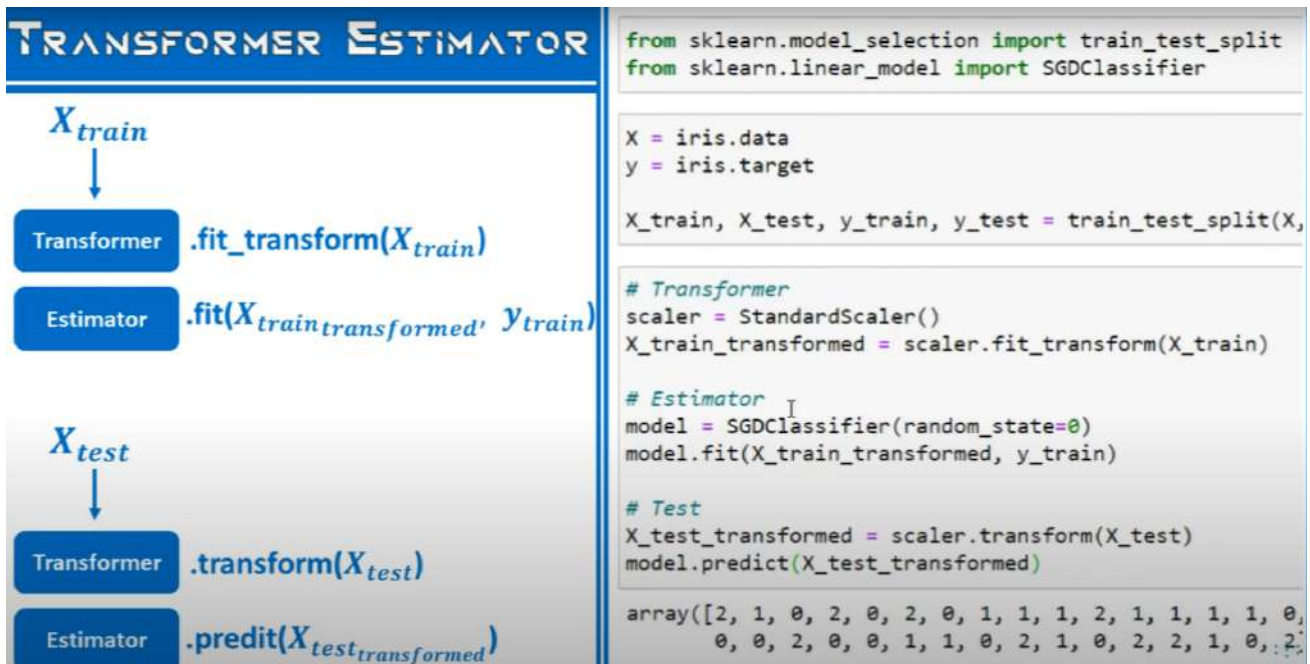
Exemple : 2 variables  $x_1, x_2 \rightarrow$  Polynôme 2

$$x_1, x_2 \rightarrow 1, x_1, x_2, x_1^2, x_1x_2, x_2^2$$

Ces variables n'étant **pas sur la même échelle**, il faut les **normaliser** avant leur passage dans l'estimateur !



## ⇒ Sklearn Pipeline



Méthode plus simple pour faire les actions vu ci-dessus :

On va regrouper notre **transformer** et notre **estimator** dans une **pipeline** (chaîne de transformation).

On obtient ainsi un Composite Estimator, donc un estimator composé de plusieurs éléments, cet estimator possède comme les autres estimator une méthode `.fit`, `.predict` et `.score`.

Quand on va utiliser la méthode `.fit` tous les composants de cet estimator vont utiliser leur méthode `.fit`, ainsi on développe notre transformer et notre estimator d'un seul coup, en 1 ligne de code. Idem quand on utilise la méthode `.predict`, alors tous les transformer vont transformer les données et l'estimator qui est en bout de chaîne va effectuer une prédiction.

### Avec Sklearn :

Pour créer une **Pipeline**, 2 options :

- Classe Pipeline
- Fonction **make\_pipeline**

```
model = make_pipeline(StandardScaler(),
                      SGDClassifier())
```

```
model.fit( $X_{train}$ ,  $y_{train}$ )
model.score( $X_{test}$ ,  $y_{test}$ )
model.predict( $X_{test}$ )
```

En code en Python on a :

```
from sklearn.pipeline import make_pipeline
model = make_pipeline(StandardScaler(), SGDClassifier())
model.fit(X_train, y_train)
model.predict(X_test)
```

Les avantages des pipelines :

- 1- Simple à utiliser.
- 2- Évite d'avoir des fuites de données ou des données mal transformées.
- 3- Permet de faire des cross-validation sur l'ensemble de notre chaîne de transformation.

Et en plus, on peut utiliser GridSearchCV avec les pipelines pour automatiser tout une chaîne de transformation :

On peut utiliser **GridSearchCV** pour trouver les meilleurs paramètres de la pipeline.

```
grid = GridSearchCV(pipeline, params, cv)
```

```
params = {  
    <Composant>__<paramètre> : [...]  
}
```

```
grid.fit(X_train, y_train)
```

```
grid.best_estimator
```

## ⇒ Sklearn Pipeline Avancée

Quand on construit une pipeline, quand on va utiliser des fonctions pour normaliser nos données, si on a des variables qualitatives dans notre Dataset alors on va voir une erreur car on ne peut pas mettre à l'échelle des valeurs qualitatives, il faut choisir nos colonnes quantitatives et éviter de prendre les qualitatives.

Pour ça il existe la fonction `make_column_transformer` :



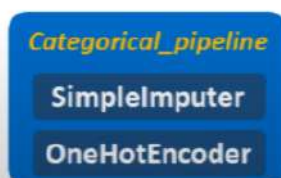
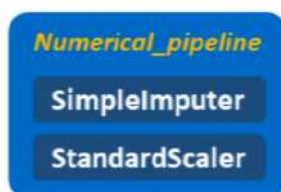
En code Python on a :

```
from sklearn.compose import make_column_transformer
transformer = make_column_transformer((StandardScaler(),
['age', 'fare'])) # permet de sélectionner que les colonnes age et
fare ici qui sont quantitatives
```

## Application classique de Column Transformer :

Séparez les catégories et variables numériques en 2 listes, puis utilisez ColumnTransformer pour traiter chaque liste de variables séparément.

On aura ainsi une pipeline avancée avec 1 numerical\_pipeline qui va traiter les données numériques et une categorical\_pipeline qui va traiter les données catégorielles :



Code en Python :

```
y = titanic['survived']
X = titanic.drop('survived', axis=1)
numerical_features = ['p_class', 'age', 'fare']
categorical_features = ['sex', 'deck', 'alone']

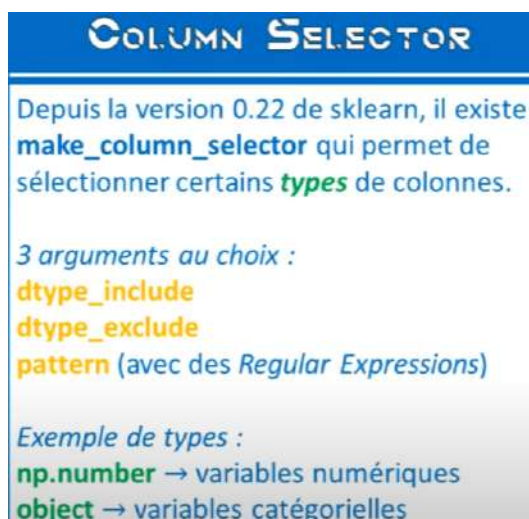
numerical_pipeline = make_pipeline(SimpleImputer(),
StandardScaler())
categorical_pipeline =
make_pipeline(SimpleImputer(strategy='most_frequent'),
OneHotEncoder())
# strategy='most_frequent' permet de remplacer les valeurs
manquantes par la valeur la plus fréquente dans la colonne.

preprocessor = make_column_transformer((numerical_pipeline,
numerical_features), (categorical_pipeline, categorical_features))

model = make_pipeline(preprocessor, SGDClassifier())
model.fit(X, y)
```

Et voilà ! On a notre Pipeline !

Bonus :



**COLUMN SELECTOR**

Depuis la version 0.22 de sklearn, il existe **make\_column\_selector** qui permet de sélectionner certains **types** de colonnes.

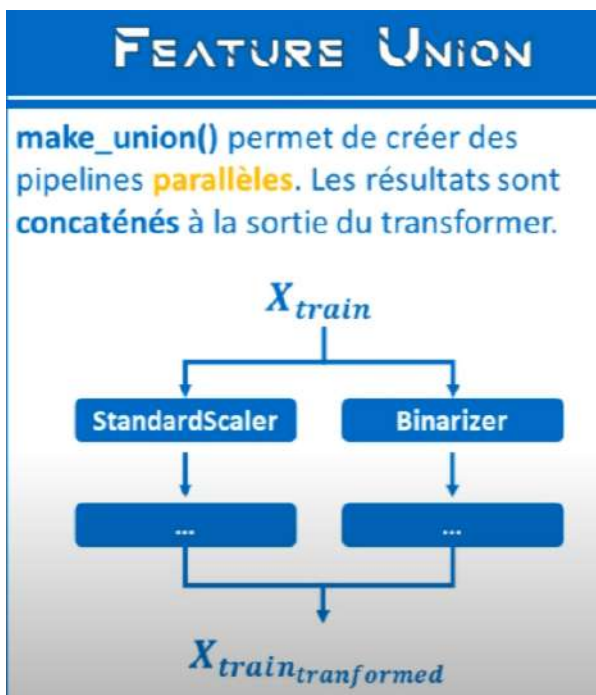
3 arguments au choix :

- dtype\_include**
- dtype\_exclude**
- pattern** (avec des *Regular Expressions*)

Exemple de types :

- np.number** → variables numériques
- object** → variables catégorielles

## Paralléliser les transformers en mettant plusieurs transformers de Pipeline en parallèle :



En code Python :

```
numercial_features = X[['age', 'fare']]  
pipeline = make_union(StandardScaler(), Binarizer())  
pipeline.fit_transform(numercial_features)
```

```
pipeline.fit_transform(numercial_features).shape
```

On obtient :

Age	Fare	Age	Fare
0.02	3.57	0	1
0.68	0.12	1	0
1.32	0.16	1	0
-1.53	-0.2	0	0

On a les 2 colonnes qui ont été standardisés et les 2 autres qui ont été discrétisés avec le Binarizer().

## ⇒ Sklearn Imputer : Nettoyage de Données

Cela permet de supprimer et ou remplacer les valeurs manquantes dans notre Dataset.

Le **SimpleImputer()** :

**SIMPLE IMPUTER**

Remplace toutes les **valeurs manquantes** par une valeur **statistique**.

- **missing\_values**
- **strategy**
  - *mean*
  - *median*
  - *most\_frequent*
  - *constant*
- **fill\_value** (pour *constant* )

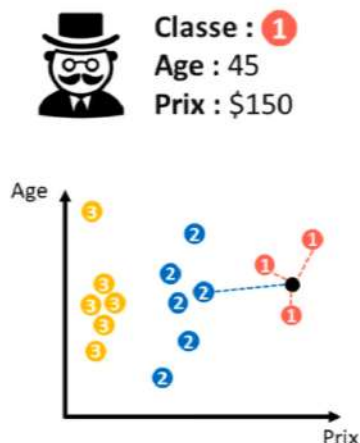
Le code en Python :

```
from sklearn.impute import SimpleImputer
SimpleImputer(missing_values=np.nan, strategy='mean')
# ici on cible les valeurs nan (not a number) et on dit qu'on les
remplace par la moyenne des valeurs de la colonne.
```

Le **KNNImputer()** :

**KNNIMPUTER**

Remplace toutes les valeurs **manquantes** par les valeurs des **plus proches voisins**.





Le code en Python :

```
from sklearn.impute import KNNImputer
imputer = KNNImputer(n_neighbors=1)
imputer.fit_transform(X)
# ca va remplacer la valeur manquante par la valeur des K
échantillons voisins (les plus semblables)
```

Le **MissingIndicator()** :

## MISSINGINDICATOR

Variable **booléenne** qui indique **l'absence** de valeurs dans le dataset.

**Utilité ?**  
→ Parfois, le **manque d'information** représente **une information...**

Classe passager		Membre équipage
2	→	False
3		False
3		False
nan		True

**Application :**  
Utiliser avec la fonction **make\_union + OneHotEncoder**

Classe passager		Classe passager	Membre équipage
2	→	2	0
3		3	0
3		3	0
nan		-99	1

Le code en Python :

```
from sklearn.impute import MissingIndicator
from sklearn.pipeline import make_union
```

```
MissingIndicator().fit_transform(X)
```

```
# retourne en valeurs booléennes les valeurs manquantes
```

```
pipeline = make_union(SimpleImputer(strategy='constant',  
fill_value=-99), MissingIndicator())
```

```
pipeline.fit_transform(X)
```

```
# on va obtenir d'un côté les colonnes qui ont subi le  
SimpleImputer et de l'autre le fait que l'individu soit plutôt un  
passager ou plutôt un membre d'équipage
```

### Astuce en Data Science :

Essayez de voir si le manque d'informations peut représenter une nouvelle information pour votre modèle.

### ⇒ Sklearn Feature Selection

On va d'abord voir le Variance Threshold

## VARIANCE THRESHOLD

**VarianceThreshold** élimine les variables dont la **variance** est inférieure à un certain seuil.

**Explication :**  
Pour effectuer des prédictions, un estimateur a besoin d'informations qui **varient** en accord avec la **Target**

Ville	Surface	Prix
Paris	50	600,000
Paris	40	450,000
Paris	45	550,000
Paris	65	700,000

« A quoi sert une variable qui ne varie pas ? »

### Methods

- `fit(self, X[, y])`
- `fit_transform(self, X[, y])`
- `get_params(self[, deep])`
- `get_support(self[, indices])`
- `inverse_transform(self, X)`
- `set_params(self, **params)`
- `transform(self, X)`

Retourne un **Booléen** indiquant les **variables sélectionnées**



On va voir le Variance SelectKBest

## SELECTKBEST

SelectKBest sélectionne les **K variables X** dont le **score du test de dépendance** avec **y** est le **plus élevé**.

test de dépendance :

- feature\_selection.chi2(X, y)
- feature\_selection.f\_classif(X, y)
- feature\_selection.f\_regression(X, y[, center])
- feature\_selection.mutual\_info\_classif(X, y)
- feature\_selection.mutual\_info\_regression(X, y)

classification :  $\chi^2$ , ANOVA  
Régression : Pearson Correlation

SelectKBest(test\_dependance, K)

Attributs :

- scores\_ : score du test de dépendance
- pvalues\_ : valeur-p du test.

Méthodes habituelles :

- get\_support()

On va voir le SelectFromModel

## SELECTFROMMODEL

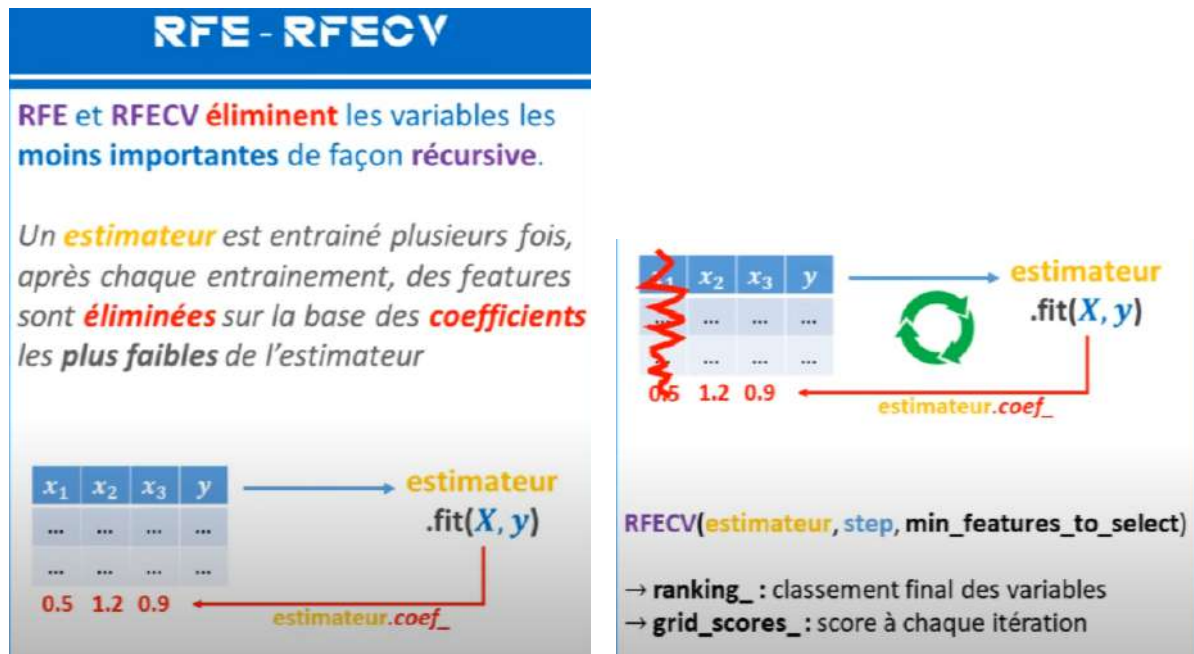
SelectFromModel entraine un **estimateur** puis sélectionne les **variables** les plus **importantes** pour cet estimateur.

*Note : Compatible avec les estimateurs qui développent une fonction paramétrée (attribut **.coef\_** ou **.feature\_importance\_**)  
⇒ K-Nearest Neighbour incompatible*

Threshold = 0.8

**Lexique** : Métatransformer = Transformer qui contient un estimator

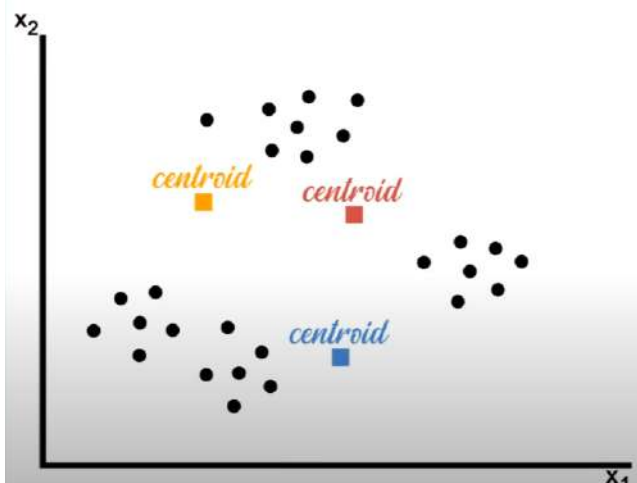
On va voir le RFE + RFECV



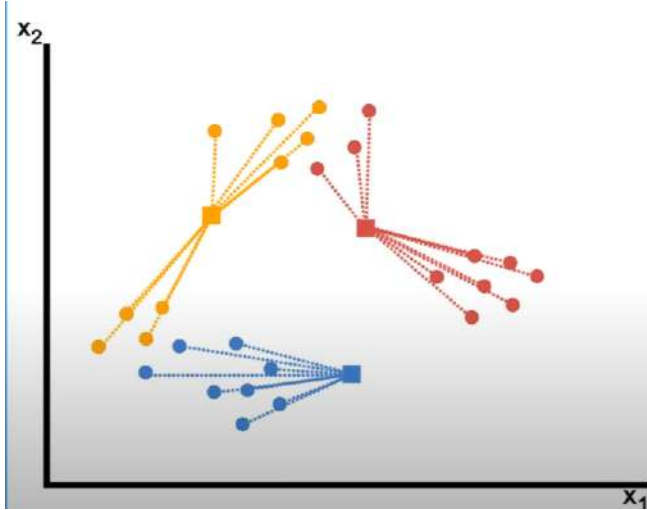
## Apprentissage non-supervisé avec Python :

### ⇒ Le K-Means Clustering

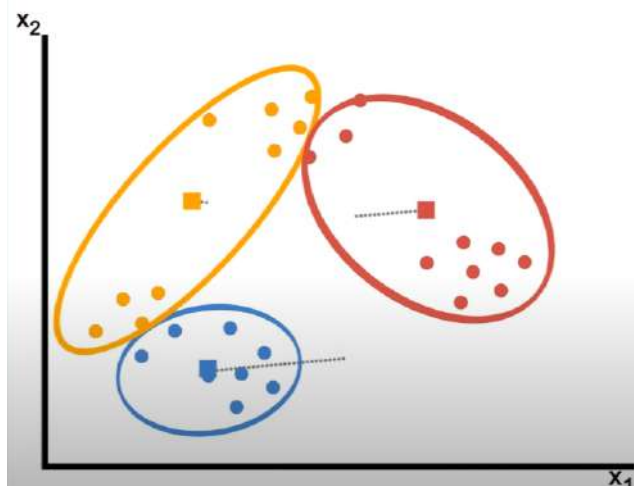
On place K centroids au hasard dans le Dataset :  
(Ce sont les barycentres des futurs clusters)



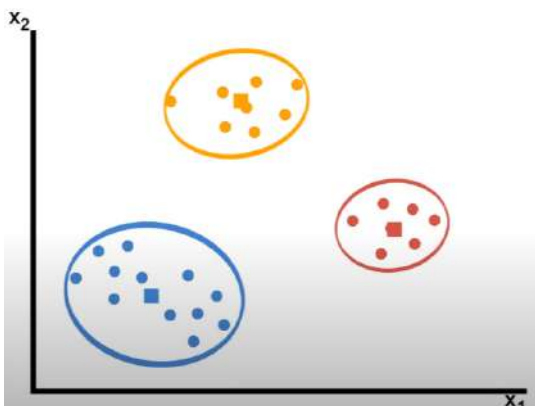
Ensuite, on affecte chaque point du Dataset au cluster du centroid le plus proche :



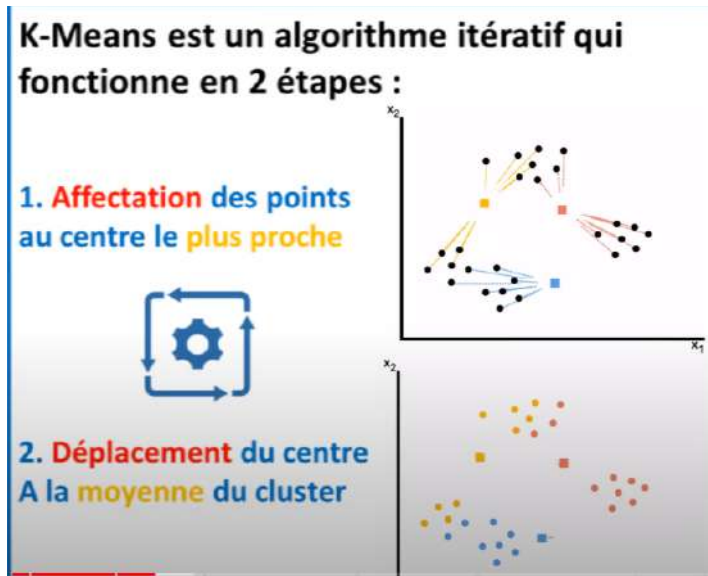
Puis, on va déplacer chaque centroid au milieu du cluster (là où se situe la moyenne des points) :



Et on répète les 3 étapes ci-dessus, jusqu'à ce que les centroids convergent vers une position d'équilibre :



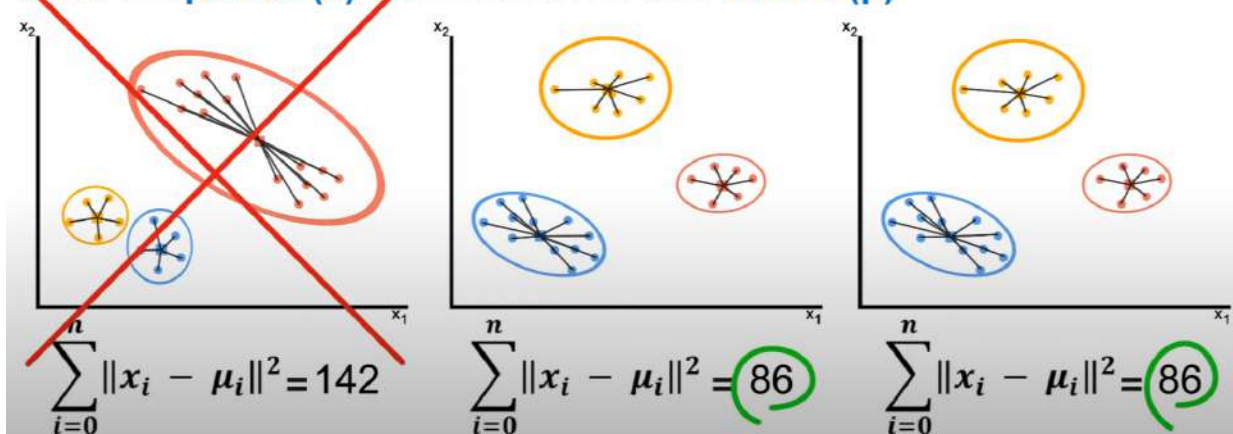
Pour résumer l'algorithme K-Means Clustering :



Après selon la position initial des centroids il est possible que ce dernier nous donne de mauvais clusters.

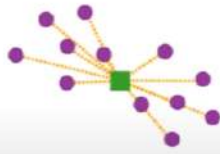
Mais il existe une **solution** pour éviter ça :

On exécute K-Mean avec **différentes positions de départ**.  
La solution retenue est celle qui **minimise** la somme des distances entre les **points (x)** d'un cluster et son **centre ( $\mu$ )**



En résumé :

K-Mean cherche la position des centres qui minimise la distance entre les points d'un cluster ( $x_i$ ) et le centre ( $\mu_j$ ) de ce dernier:



$$\sum_{i=0}^n \min_{\mu_j \in C} (\|x_i - \mu_j\|^2)$$

Cette technique équivaut à minimiser la variance des clusters.

### Le code avec Sklearn en Python :

La doc :

<https://scikit-learn.org/stable/modules/clustering.html>

#### Hyper-Paramètres :

**n\_clusters** : nombre K de clusters

**n\_init** : nombre d'exécutions (10)

**max\_iter** : nombre d'itérations (300)

**Init** : type d'initialisation (k-means++)

#### Méthodes :

**Fit(X)** : exécute l'algorithme K-Mean

**Predict(X)** : centroid le plus proche de X

**Score(X)** : calcul de l'inertia (négatif)

#### Attributs :

**cluster\_centers\_** : position des centroids

**Labels\_** : équivalent de Predict( $X_{train}$ )

**Inertia\_** : calcul de l'inertia (positif)

Par défaut **init** utilise K-Means++ qui est une méthode consistant à placer les centroids sur des points du Dataset très éloignés les uns des autres.

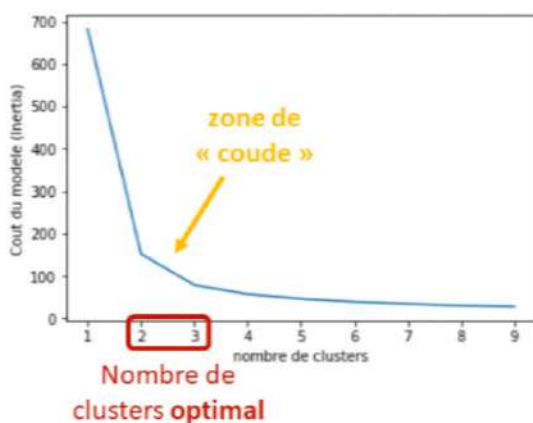
→ Cela facilite la convergence

```
from sklearn.cluster import KMeans
model = Kmeans(n_clusters=3)
model.fit(X)
model.labels_ # permet de voir comment son classé nos différents
échantillons ou on peut utiliser model.predict(X) c'est égal
model.cluster_centers_ # affiche la position finale de nos centroids
```

Pour trouver le nombre de `n_clusters` qu'on doit avoir, on utilise la technique de **Elbow** :

Pour trouver le bon nombre de clusters :

« **Elbow Method** » : Détecter une zone de « **coude** » dans la minimisation du Coût (`inertia_`)



```
inertia = []
K_range = range(1,20)
for k in K_range :
    model = Kmeans(n_clusters=k).fit(X)
    inertia.append(model.inertia)
```

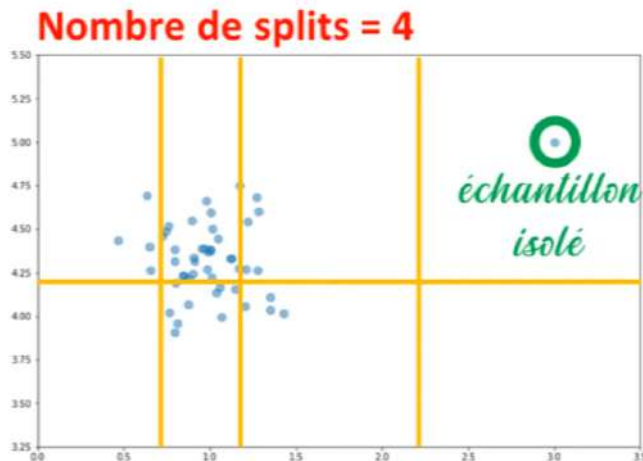
```
plt.plot(K_range, inertia)
plt.xlabel('nombre clusters')
plt.ylabel('Coût du modèle (inertia)')
```

Permet de voir la zone de coude et donc de savoir combien on doit mettre de `n_clusters` pour un Dataset donné.

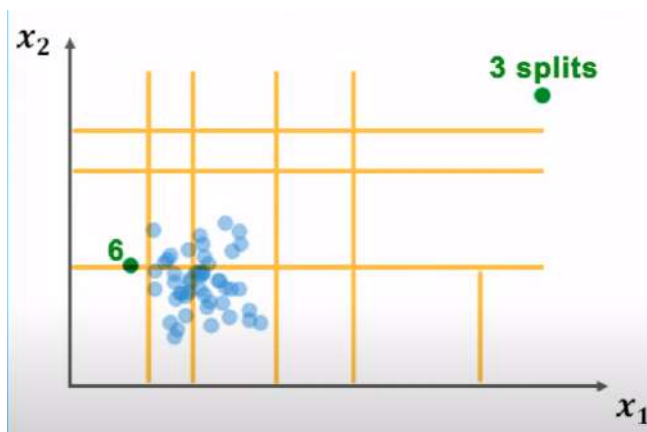


## ⇒ Isolation Forest, Anomaly Detection

Pour l'algorithme d'isolation forest, on effectue une série de **splits** aléatoire, et on **compte** le nombre de splits qu'il faut effectuer pour pouvoir **isoler** nos échantillons.



Faible nombre de **splits** → **Anomalie**



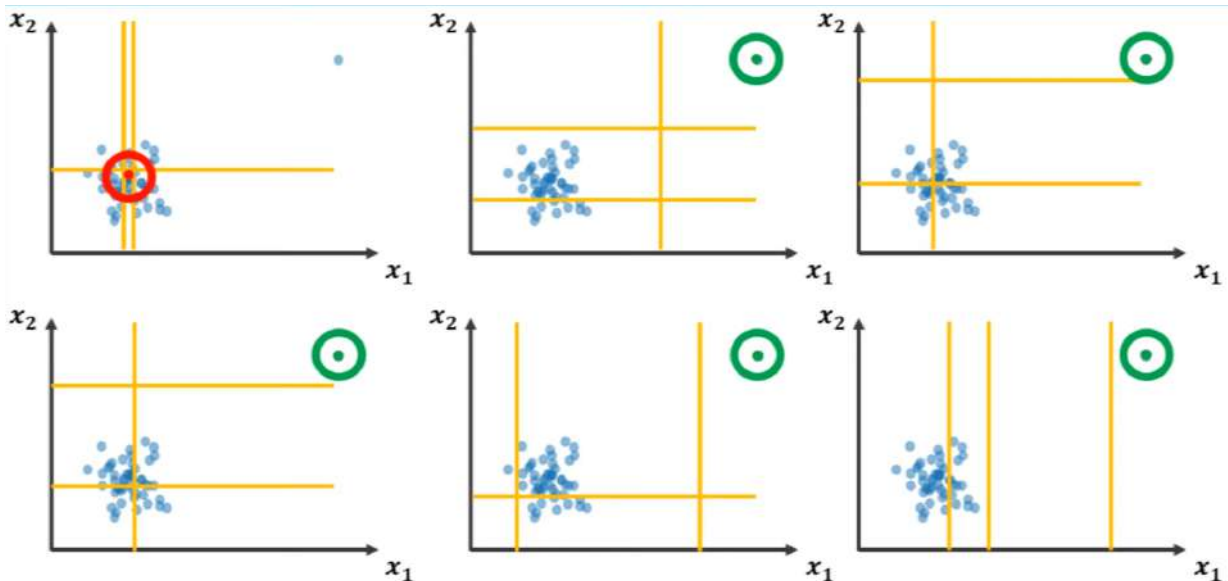
Faible nombre de splits = **Forte probabilité** d'anomalie

⇒ *Après la probabilité et faible mais possible, d'isoler un échantillon perdu dans la masse avant d'isoler une véritable anomalie.*

Mais en solution face à ça, on va demander à la machine de générer plusieurs estimateur qui vont chacun effectuer une séquence de splits aléatoires et en considérant alors l'ensemble de leurs résultats, on est capable de disqualifier les quelques erreurs qui pourraient être commises par certain estimateur.



Illustration :



On appelle ça **une technique d'ensemble**, le résultat final correspond à l'ensemble (donc la moyenne) des résultats. Et on nomme Forest, car l'algorithme Isolation Forest est un ensemble de générateur de type arbre, les splits représentent les embranchements des différents arbres.

### Le code avec Sklearn en Python :

La doc :

<https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.IsolationForest.html>

Dans sklearn, il suffit de définir le **pourcentage** de données que vous désirez filtrer (taux de **contamination**)

Exemple :

```
IsolationForest(contamination = 0.02)
```

`fit(X)` → entraîne le modèle

```
predict(X) → { +1 : normale  
               -1 : anomalie
```

```
from sklearn.ensemble import IsolationForest
model = IsolationForest(contamination=0.01)
model.fit(X)
plt.scatter(X[:, 0], X[:, 1], c=model.predict(X))
```

Il existe un algorithme efficace qui se nomme **Local Outlier Factor** qui permet aussi de trouver des anomalies mais en utilisant les voisins les plus proches et permet de faire de la **Novelty Detection** :

- **Outlier Detection** : détecte les anomalies dans le Train Set (Isolation Forest)
- **Novelty Detection** : détecte les anomalies dans les données futures (Local Outlier Factor)

## ⇒ PCA (Principal Component Analysis), Réduction de Dimension

C'est une des technique et application les plus importante en apprentissage non-supervisé.

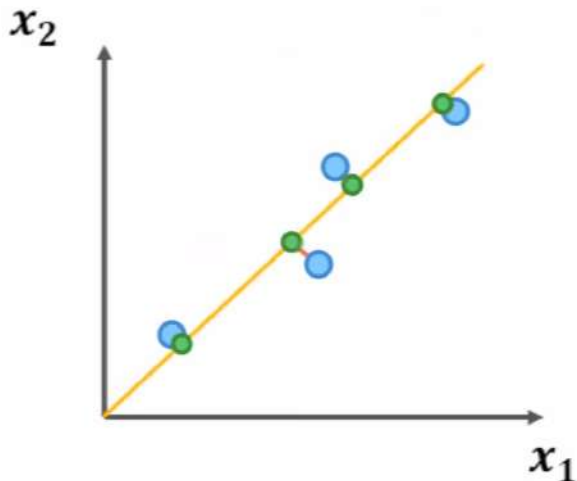
Le principe est de réduire la **complexité superflue** d'un Dataset en projetant ses données dans un espace de plus petite dimension (un espace avec **moins de variables**).



Les avantages de cette technique :

- **Accélère** l'apprentissage de la machine
- Lutte contre le **fléau de la dimension** (risque d'overfitting lié au surplus de dimensions)

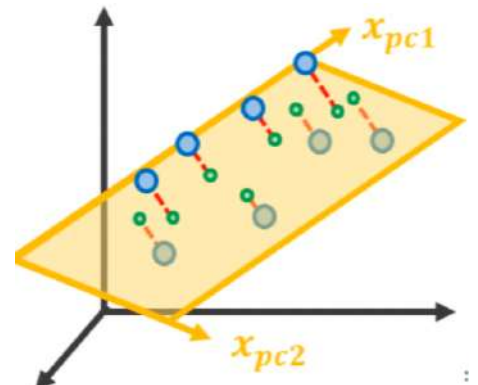
L'algorithme le plus utilisé pour réduire les dimensions est l'algorithme PCA (Analyse en Composantes Principales). Le principe est de **projeter** nos données sur des axes appelés **Composantes Principales**, en cherchant à **minimiser la distance** entre nos points et leur **projections**.



De cette manière on réduit la dimension de notre Dataset, tout en **préservant au maximum la variance** de nos données.

Pour trouver **les axes** de projection ( $x_{pc}$ ) :

1. On calcule la **matrice de covariance** des données
2. On détermine les **vecteurs propres** de cette matrice : ce sont les **Composantes Principales**
3. (On **projette** les données sur ces axes)





**PCA est un transformer !**

1. Définir le nombre de **composantes**
2. Transformer les données avec **fit\_transform()**

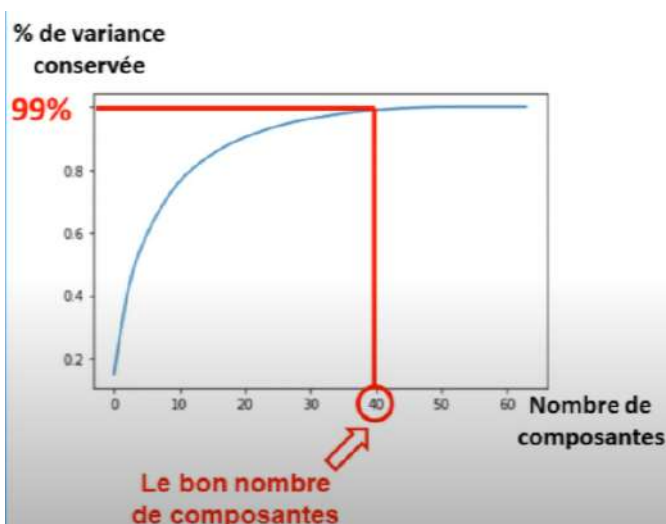
```
from sklearn.decomposition import PCA  
  
model = PCA(n_components=10)  
  
model.fit_transform(X)
```

Pour choisir le nombre de composantes il y a 2 cas possibles :

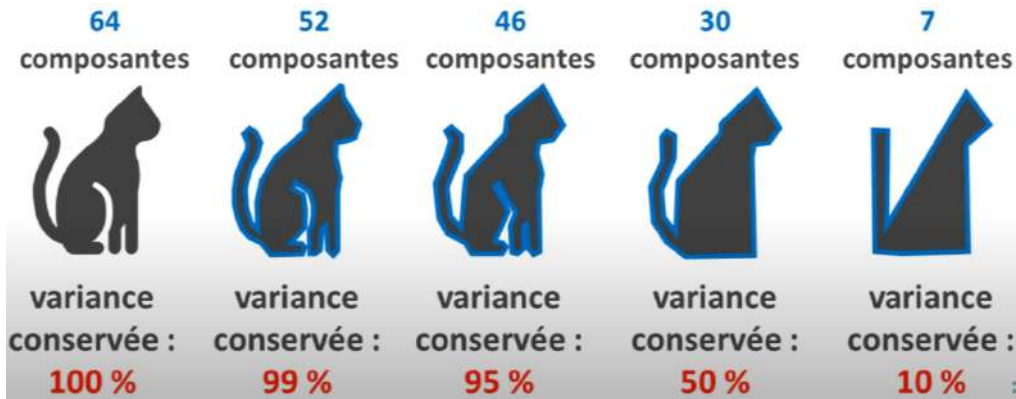
1- Visualisation des données : on projette notre Dataset dans un espace 2D ou 3D ( $n\_components = 2$  ou  $3$ )



2- Compression de données : réduire au maximum la taille du Dataset tout en conservant **95-99 % de la variance** de nos données



Plus on réduit la dimension d'un Dataset et plus on perd en qualité (donc en variance) :



Le code avec Sklearn en Python :

La doc :

<https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html>

**Visualisation de données**

Dataset digits :  
X contient 64 variables

```
model = PCA(n_components = 2)
```

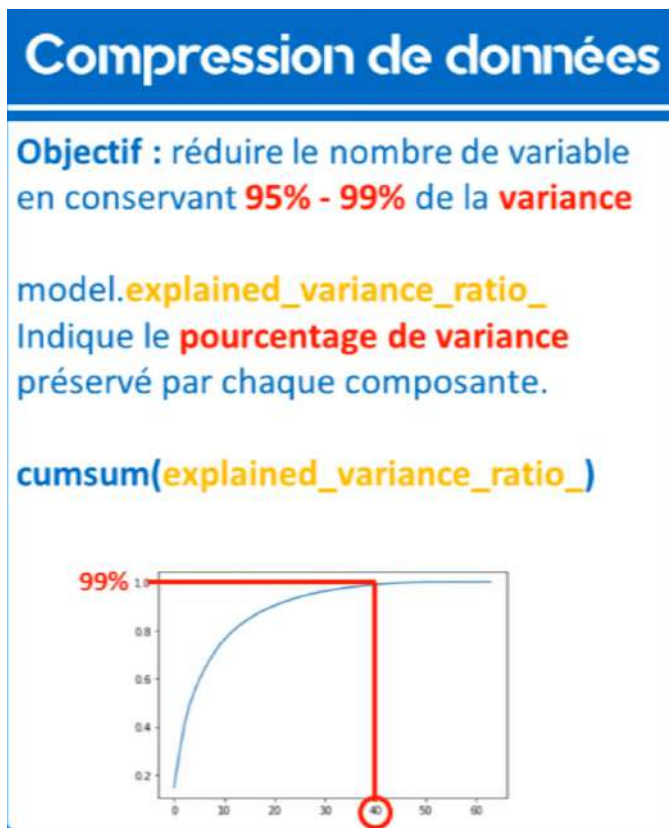
$X_{\text{reduced}}$  = model.fit\_transform(X)

(1797, 2)                      (1797, 64)

model.components\_ : Dévoile la combinaison linéaire entre les composantes et les variables de X

```
from sklearn.decomposition import PCA
model = PCA(n_components=2)
X_reduced = model.fit_transform(X)
plt.scatter(X_reduced[:, 0], X_reduced[:, 1], c=y)
plt.colorbar()
```

`model.components_` # permet de voir les combinaisons linéaire de nos points après la réduction de dimensionnalité.



```
from sklearn.decomposition import PCA
model = PCA(n_components=64) # on met le même nombre de
dimension qu'on a dans X donc 64 pour l'exemple ici
X_reduced = model.fit_transform(X)
np.cumsum(model.explained_variance_ratio_) # on va chercher le
moment où on atteint 95-99 %
plt.plot(np.cumsum(model.explained_variance_ratio_)) # on
regarde combien de composant on a besoin pour atteindre un tel
pourcentage
np.argmax(np.cumsum(model.explained_variance_ratio_) > 99)
# on va trouver l'indice qui représentera le nombre de composante
nécessaire pour atteindre ici 99 % de variance conservée
```



A partir de là on a plus qu'à ré-entraîner le modèle avec le nombre de composantes idéales :

```
model = PCA(n_components=40)
model.fit_transform(X)
```

Et si on veut observer une image compressé on fait :

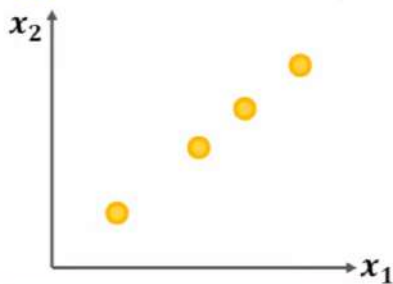
Pour observer une image **compressé** :

```
model.inverse_transform( $X_{reduced}$ )
```

$(1797, 64)$        $(1797, 2)$

Attention ! Cela ne remet pas les données à leur position d'origine !

```
 $X_{recover} = inverse\_transform(X_{reduced})$ 
```



**A Savoir** : on peut directement écrire le pourcentage de données qu'on souhaite conserver dans le **n\_components** en faisant :

```
n_components=0.99
```

```
model.n_components_ # et on va voir le nombre de composants  
nécessaire pour atteindre ce pourcentage (ici 99%)
```

### Quelques points importants :

- Il faut **Standardiser** les données avant d'utiliser PCA (**StandardScaler**)
- PCA est normalement conçu pour traiter les **variables continues**
- PCA n'est **pas efficace** sur les datasets **non-linéaires** :



Pour traiter des données non-linéaires on préfère utiliser des techniques de **Manifold Learning** comme **IsoMap**, **T-SNE** etc

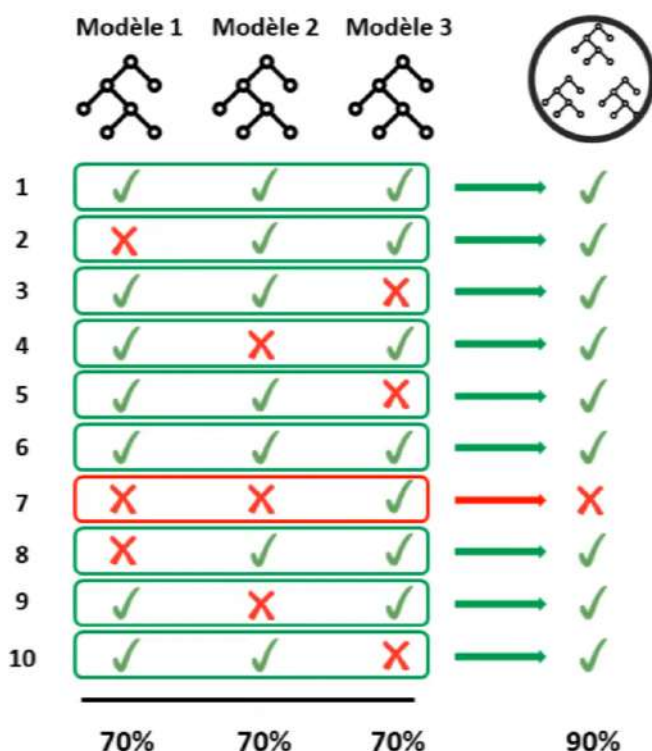
## L'Ensemble Learning :

Technique qui consiste à entraîner plusieurs modèles de ML pour ensuite considérer l'ensemble de leurs prédictions.

Il existe 3 grandes méthodes pour cela :

- Le Bagging
- Le Boosting
- Le Stacking

⇒ **Wisdom of the Crowd**



En regroupant nos modèles tous ensemble, on obtient de meilleure performance que lorsqu'on utilise nos modèles chacun de leur côté.

Ce résultat vient d'un phénomène qu'on nomme « the Wisdom of the Crowd ».

→ L'idée c'est que : une foule d'individu a plus souvent raison qu'un individu seul.

L'idée globale derrière ça c'est que :

**[L'avis d'une foule d'amateurs est meilleur que celui d'un expert seul]**

Pour nos modèles en ML c'est due à la **loi des grands nombres** : Plus le **nombre d'individus augmente** dans la foule, plus la **performance de la majorité** s'approche de **100 % de bonnes réponses**.

**Mais pour que ça fonctionne il faut au moins que chacun de nos modèles ML ait au moins 50 % de performance !**

(car si la performance est  $< 50\%$  alors la majorité converge vers une performance de  $0\%$ )

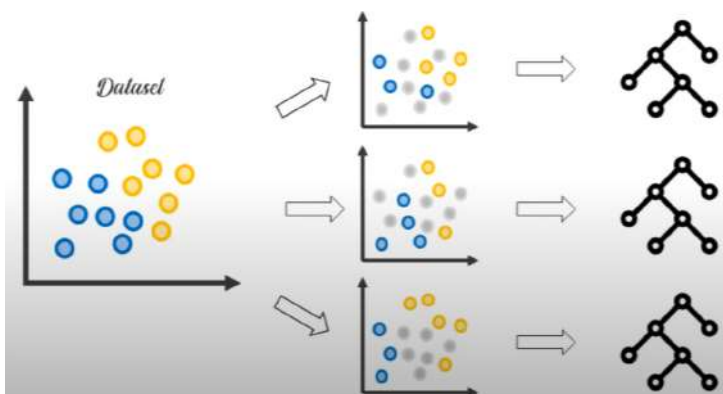
**Il faut aussi que nos modèles (foule) représentent un maximum de diversités, exemple si dans la foule toutes les personnes pensent pareil alors elles feront toutes les mêmes erreurs.**

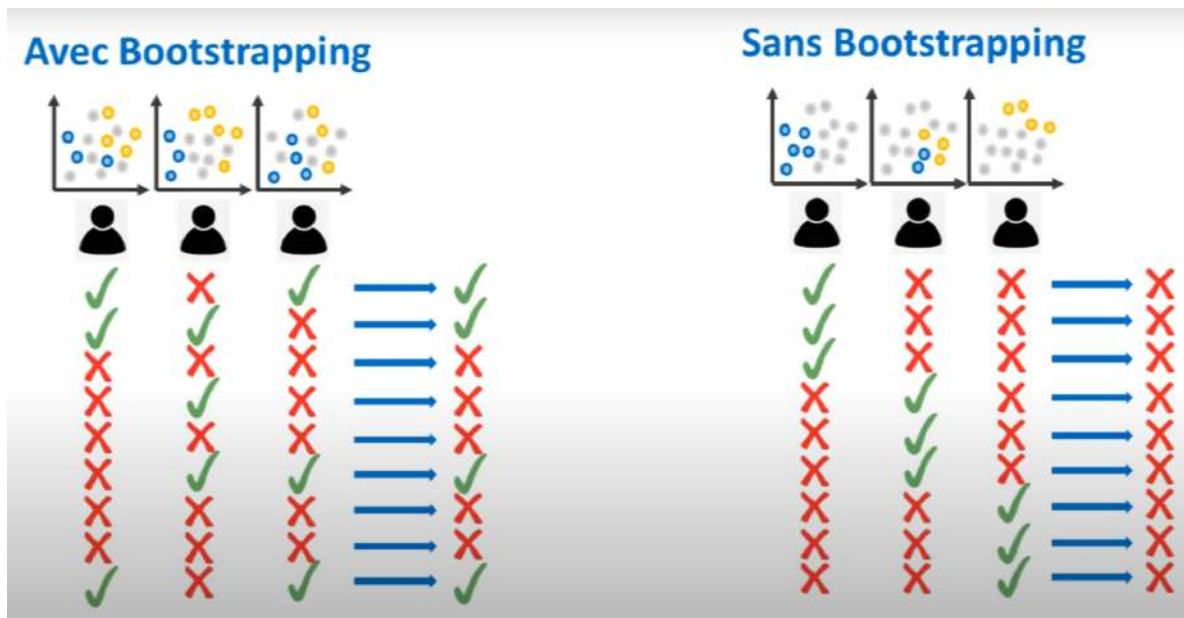
**C'est grâce à la diversité que l'union fait la force !**

## ⇒ Le Bagging

Créer plusieurs copies d'un même modèle, en entraînant chaque copie sur une partie aléatoire du Dataset.

Pour ça on utilise une technique d'échantillonnage nommé Bootstrapping qui consiste à replacer après chaque tirage au sort les données qui ont été sélectionnés.





C'est grâce à cette technique que l'on obtient la **diversité** nécessaire.

**L'algorithme le plus connu qui utilise cette technique est l'algorithme de Random Forest.**

### ⇒ Le Boosting

**Entraîne l'un après l'autre plusieurs modèles relativement faibles, en demandant à chaque modèle d'essayer de corriger les erreurs effectués par le précédant modèle.**

On va obtenir un ensemble de modèle **parfaitement complémentaire** dans lequel les **faiblesses des uns** sont compensés par les **forces des autres**.

Les algorithmes utilisant cette technique sont :

**AdaBoost et Gradient Boosting.**

## Bagging VS Boosting :

### Bagging

- Chaque modèle est **fort** mais **overfit** son sub-set.
- La foule permet de **réduire la variance**

### Boosting

- Chaque modèle est **faible**, en **underfitting**.
- La foule permet de **réduire le biais**

### ⇒ Le Stacking

Entraîne un modèle de ML par dessus les prédictions de notre foule.

**On demande à un modèle d'apprendre à reconnaître qui a tort et qui a raison dans notre foule, pour lui même prédire le résultat final.**

## Le code avec Sklearn en Python :

La doc :

<https://scikit-learn.org/stable/modules/ensemble.html>

### ⇒ Voting Classifier

```
from sklearn.linear_model import SGDClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
```

```
from sklearn.ensemble import VotingClassifier
```

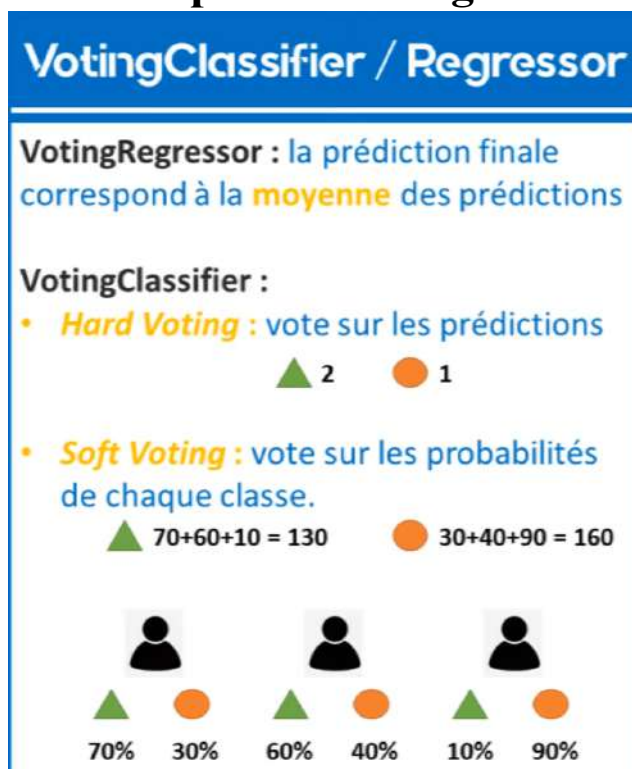
```

model1 = SGDClassifier(random_state=0)
model2 = DecisionTreeClassifier(random_state=0)
model3 = KNeighborsClassifier(n_neighbors=2)

model4 = VotingClassifier([('SGD', model1),
                           ('Tree', model2),
                           ('KNN', model3)], voting='soft')

```

**A Savoir pour le voting = :**



En général on dit que le vote **Soft** est un peu meilleur lorsque nos modèles sont bien calibrés, il vaut mieux **l'utiliser quand c'est possible**, donc quand on peut utiliser des modèles qui émettent des probabilités.

**MAIS**, en général le **VotingClassifier** n'est pas une technique très efficace car il est très difficile avec elle de respecter le critère de diversité.

**La meilleure façon de gagner en diversité c'est grâce au Bagging et au Boosting !**

### ⇒ **Bagging**

```
from sklearn.ensemble import BaggingClassifier,
RandomForestClassifier
model =
BaggingClassifier(base_estimator=KNeighborsClassifier(),
n_estimators=100) # par défaut n_estimators=100
model.fit(X_train, y_train)
model.score(X_test, y_test)
```

```
model = RandomForestClassifier(n_estimators=100)
model.fit(X_train, y_train)
model.score(X_test, y_test)
```

**En général, un RandomForest aura de meilleur résultat qu'un BaggingClassifier avec un KNeighborsClassifier en base, mais il faut bien le régler.**

### ⇒ **Boosting**

```
from sklearn.ensemble import AdaBoostClassifier,
GradientBoostingClassifier
model = AdaBoostClassifier(n_estimators=100)
model.fit(X_train, y_train)
model.score(X_test, y_test)
```

### ⇒ **Stacking**

```
from sklearn.ensemble import StackingClassifier
model = StackingClassifier([('SGD', model1),
                             ('Tree', model2),
                             ('KNN', model3)],
                           final_estimator=KNeighborsClassifier())
model.fit(X_train, y_train)
model.score(X_test, y_test)
```

## En résumé :

On va utiliser le **Bagging** quand on se rend compte que nos modèles ont tendance à faire de l'**Overfitting**, à l'inverse si nos modèles ont du mal à atteindre une bonne performance et ont tendance à faire de l'**Underfitting** alors on utilise le **Boosting**. Le **Stacking** est une technique qui peut être forte si on a passé énormément de temps à entraîner chacun de nos modèles individuellement (mais c'est une technique qui peut être très lente).

## Data Science et Démarche de Travail :

### ⇒ Démarche de Travail

#### 1- Définir un objectif mesurable :

Choisir de bons outils de mesures selon le besoin, l'accuracy est bien mais par exemple pour analyser si un patient est infecté ou non sur 100 individus avec 90 individus non infectés, si le modèle prédit pour les 100 individus qu'ils ne sont pas infectés alors le modèle aura raison à 90 % du temps, mais pour autant le modèle sera TRES TRES MAUVAIS !

C'est pour ça qu'il existe d'autres métriques comme :

## Precision et Recall

		Vrais valeurs		
		Classe 1	Classe 0	
prédictions	Classe 1	(TP) True Positive	(FP) False Positive	$Precision = \frac{TP}{TP+FP}$ → permet de réduire à maximum le taux de Faux Positifs
	Classe 0	(FN) False Negative	(TN) True Negative	
				$Recall = \frac{TP}{TP+FN}$ → permet de réduire à maximum le taux de Faux Négatifs



2- Analyser et explorer les données (EDA → Exploratory Data Analysis)

3- Pre-Processing

4- Modelling

## **Conseils & Checklist de base (non-exhaustive) des choses à faire pour un projet :**

### **⇒ Exploratory Data Analysis**

**Objectif :** Comprendre au maximum les données dont on dispose pour définir une stratégie de modélisation.

#### **Checklist de base (non-exhaustive)**

##### **Analyse de la forme :**

- ☐ Identification de la target
- ☐ Nombre de lignes et de colonnes
- ☐ Types de variables
- ☐ Identification des valeurs manquantes

##### **Analyse du fond :**

- ☐ Visualisation de la target (Histogramme / Boxplot)
- ☐ Compréhension des différentes variables (Internet)
- ☐ Visualisation des relations features – target (Histogramme / Boxplot)
- ☐ Identification des outliers

### **⇒ Pre-Processing**

**Objectif :** transformer le data pour le mettre dans un format propice au Machine Learning

#### **Checklist de base (non-exhaustive)**

- ☐ Création du Train set / Test Set
- ☐ Élimination des NaN : dropna(), imputation, colonnes
- ☐ Encodage
- ☐ Suppression des outliers néfastes au modèle
- ☐ Feature Selection
- ☐ Feature Engineering
- ☐ Feature Scaling

⇒ **Modelling**

**Objectif : Développer un modèle de ML qui réponde à l'objectif final.**

Checklist de base (non-exhaustive)

- ☐ Définir une fonction d'évaluation
- ☐ Entraînement de différents modèles
- ☐ Optimisation avec GridSearchCV
- ☐ (Optionnel) Analyse des erreurs et retour au Preprocessing / EDA
- ☐ Learning Curve et prise de décision